CM3015 Machine Learning and Neural Networks
End term Coursework
(bk112)
2023-09-05

Within this report we will be creating a neural network with the purpose of predicting housing prices based in Boston during the late 1970s. The problem is predicting a continuous variable, making this a regression problem. We will be using as many tools and libraries available to perform the task, and will evaluate how well a neural network does at this task.

This report will consist of the following sections:

- Dataset Exploration
- Baseline
- Model Development
- Finalising Model
- Conclusion

In [ ]:
```
# Lets just load all the libraries we going to use in one shot
import math
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from keras.datasets import boston_housing
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras.callbacks import ReduceLROnPlateau
```

Dataset Exploration

We have available the Boston dataset from Keras. The dataset contains 13 feature variables and 506 samples, where the dependent variable are the house prices.

- Housing prices are recorded as a factor of 1000, so if a house price is valued at 10.0, this will be the value of $10 000.
- The minimum house value is 5.0 and the maximum house value is 50.0

Below is a list of the feature variables:

1. CRIM: Per capita crime rate by town.
2. ZN: Proportion of residential land zoned for lots over 25,000 sq. ft.
3. INDUS: Proportion of non-retail business acres per town.
4. CHAS: Charles River dummy variable (1 if tract bounds river; 0 otherwise).
5. NOX: Nitric oxides concentration (parts per 10 million).
6. RM: Average number of rooms per dwelling.
7. AGE: Proportion of owner-occupied units built prior to 1940.
8. DIS: Weighted distances to five Boston employment centers.
9. RAD: Index of accessibility to radial highways.
10. TAX: Property tax rate (full-value property-tax rate per $10,000).
11. PTRATIO: Pupil-teacher ratio by town.
12. B: 1000(Bk - 0.63)^2 where Bk is the proportion of Black residents by town.
13. LSTAT: Percentage lower status of the population.

This section will be further split into the additional sections:

- Visually exploring the data
- Pre-processing the data

In [ ]:
```
# Lets load the dataset, with 20% being the test set
(train_data, train_targets), (test_data, test_targets) = boston_housing.load_data(
    path="boston_housing.npz", test_split=0.2, seed=111
)

# Lets create a full dataset for some statistical analysing
full_data = np.concatenate((train_data,test_data))
full_targets = np.concatenate((train_targets,test_targets))

# Lets state the features
feature_names = ["CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE", "DIS", "RAD", "TAX", "PTRATIO", "B", "LSTAT"]

# Lets create a data frame for the data for some quick analytics
```

```
full_data_df = pd.DataFrame(full_data, columns=feature_names)
full_data_df['PRICES'] = full_targets
```

Visually exploring the data

Lets plot the data as is, and see if we can identify any patterns and relations.

In [ ]:
```
# Lets first have a look at the dataset
full_data_df.head()
```

Out[ ]:

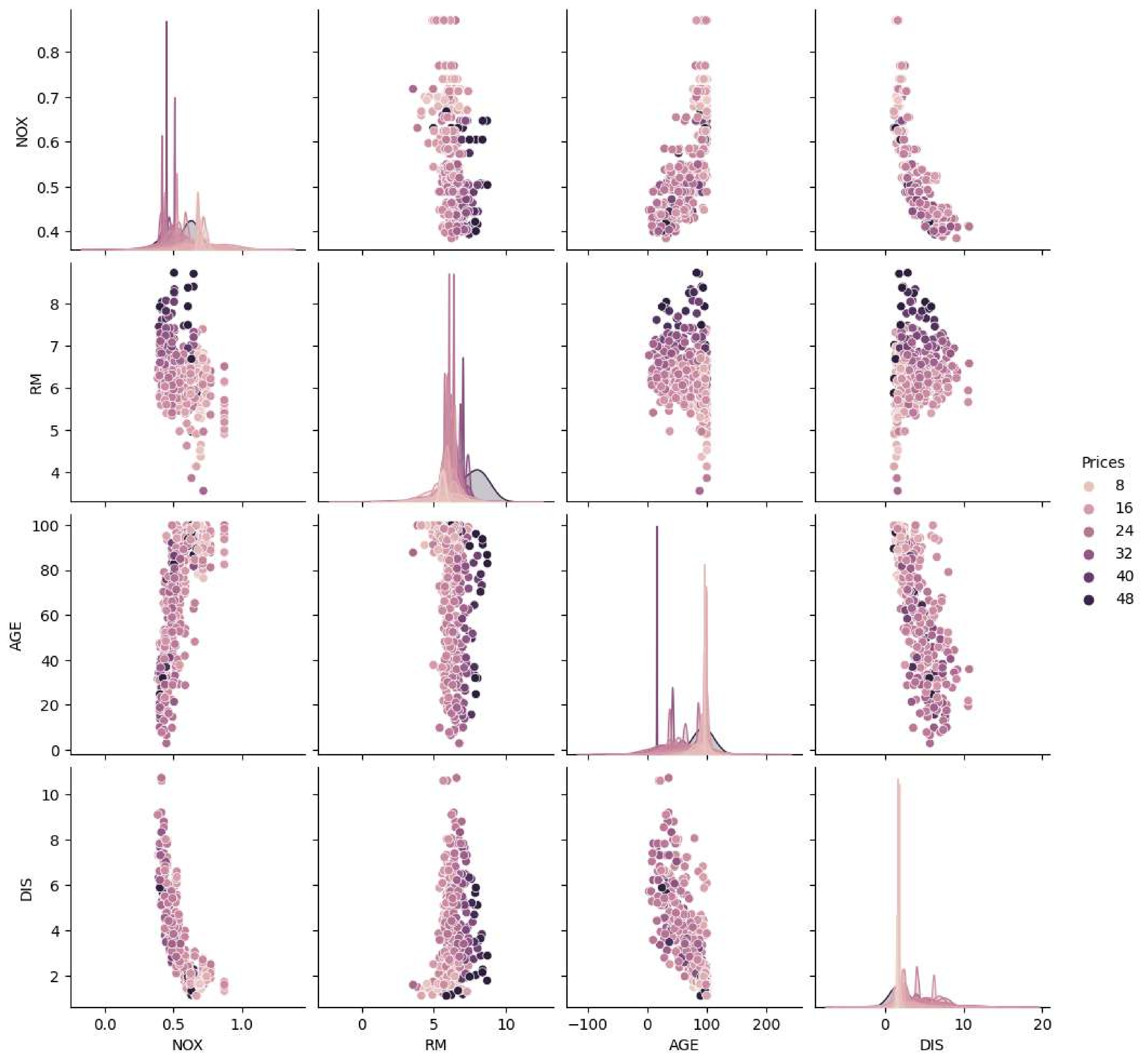
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICES
0	1.23247	0.0	8.14	0.0	0.538	6.142	91.7	3.9769	4.0	307.0	21.0	396.90	18.72	15.2
1	0.02177	82.5	2.03	0.0	0.415	7.610	15.7	6.2700	2.0	348.0	14.7	395.38	3.11	42.3
2	4.89822	0.0	18.10	0.0	0.631	4.970	100.0	1.3325	24.0	666.0	20.2	375.52	3.26	50.0
3	0.03961	0.0	5.19	0.0	0.515	6.037	34.5	5.9853	5.0	224.0	20.2	396.90	8.01	21.1
4	3.69311	0.0	18.10	0.0	0.713	6.376	88.4	2.5671	24.0	666.0	20.2	391.43	14.65	17.7

Visualize the data

Let's create a pairplot to visualize what is happening between the independent variables and the dependent variable. Though as there are 13 variables, this will look a bit messy, and as we are just trying to identify some statistical relations i have opted to select 4 of the independent variables to analyse.

In [ ]:
```
# Helper functions

# plotData
def plotData(X,y,features):
    """
    Plots a matrix pair scatter plot.

    Args:
        X (array-like): Array of independent variable vectors.
        y (array-like): Array of dependent variables.
        features (array-like): Array of feature labels

    Returns:
        Nothing
    """
    # Creates a dataframe
    df = pd.DataFrame(X, columns = features)
    # Concatenate dependent variables with independent variables
    df['Prices'] = y
    sns.pairplot(df, hue="Prices")
    plt.show()
```

In [ ]:
```
# Selecting 4 independent variables for visualising
variable_indicies_to_select = [4, 5, 6, 7] # "NOX", "RM", "AGE", "DIS"

# Splitting the dataset for plotting
selected_matrix = [[row[i] for i in variable_indicies_to_select] for row in train_data]

# Plotting the split dataset
plotData(selected_matrix,train_targets,["NOX", "RM", "AGE", "DIS"])
```

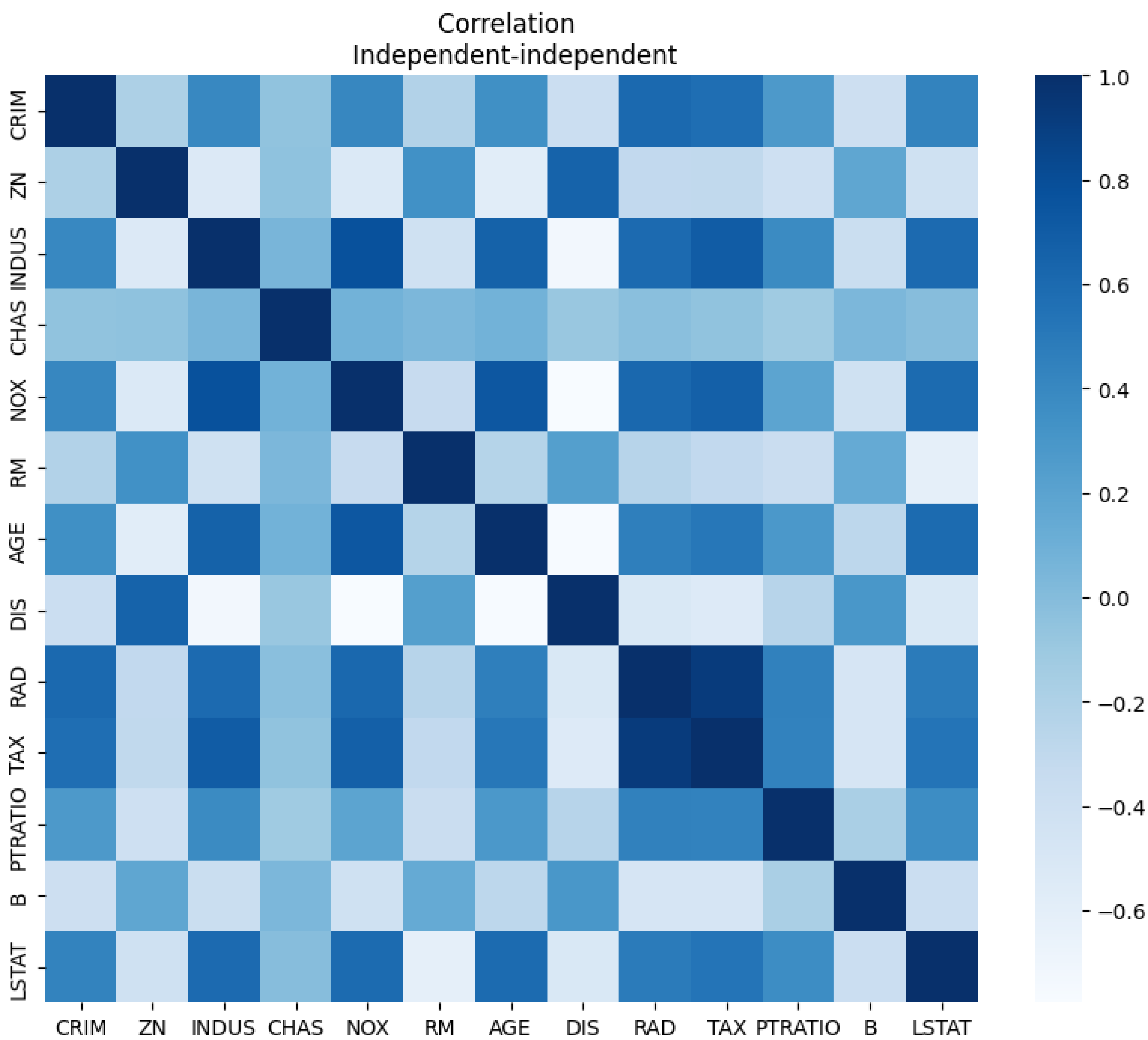


Visualize independent variable correlations

Lets get an idea how correlated the independent variables are with eachother.

In [ ]:
```
# Helper functions

# plotHeatMap
def plotHeatMap(X, features, title):
    """
    Plots a heat map of a matrix.

    Args:
        X (array-like) 2D: 2D form of matrix.
        features (array-like): The lables of the independent variables
        title (str): Title for plot

    Returns:
        Nothing
    """
    # Creates a dataframe
    df = pd.DataFrame(X, columns=features)
    # Calculating correlations
    correlations = df.corr()

    plt.figure(figsize=(10, 8))
    sns.heatmap(correlations, cmap='Blues', xticklabels=features, yticklabels=features)
    plt.title(title)
    plt.show()
```

In [ ]:
```
# Plotting the correlations
plotHeatMap(train_data, feature_names, "Correlation \n Independent-independent")
```



Visual analysis

From the first pair plot we notice, although we are only looking at 4 of the feature variables that there are some patterns evident, which is shown in the separation of the prices as well as the steady transition of the price data within the plots. So we should be able to train a model to identify these patterns.

From the second heat map we notice there is an abundant of highly correlated independent variables. If we were working on some type of linear regression algorithm we would need to do some dimensional reduction to sort out these highly correlated variables, though as our aim is to develop a neural network, highly correlated features are less of a problem as the network will perform its own feature extractions.

Pre-processing the data

Lets process our data. As we would like to prevent the range difference of independent variable values taking higher precedence in the models we will be normalising the independent variable dataset. We will also identify if there are any null values to handle.

With regards to normalisation, and due to our independent variables values being continuous we will be normalising the values around 0 utilising their standard deviations.

In [ ]:

```
# Helper functions

# normaliseData <Referenced: From course work>
def normaliseData(data, mean, std):
    """
    Normalises data around 0 according to the sets standard deviation.

    Args:
        data (array-like): array of doubles.
        mean (double): Mean of the data
        std (double): Standard deviation of the data

    Returns:
        Normalised data: Array-like
    """
    data = data.copy()
    data -= mean
    data /= std
    return data
```

In [ ]:
```
# Check if there are any missing values in any of the columns
full_data_df.isnull().sum().sum()
```

Out[ ]: 0

In [ ]:
```
# Lets get the mean of each of the variables within the dataset
mean = train_data.mean(axis=0)

# Lets get the standard diviation of each of the variables in the dataset
std = train_data.std(axis=0)

# Lets generate normalised instances of the data
train_data_normalised = normaliseData(train_data, mean, std)
test_data_normalised = normaliseData(test_data, mean, std)
```

Baseline

Before we start developing our neural network model, we need to evaluate the data and check there is some statistical power. Since this is a regression problem, lets use a simple linear regressor as our baseline model.

We will split this section up as:

- Baseline model
- Evaluation
- Conclusion

Baseline model

Lets create a baseline simple linear regression model using the sklearn library.

In [ ]:
```
# Creating our model instance
simple_linear_regression_model = LinearRegression(fit_intercept=True)
# Training our model
simple_linear_regression_model.fit(train_data_normalised, train_targets)
```

Out[ ]:
```
▾ LinearRegression
LinearRegression()
```

Evaluation

The metrics we will be using for the evaluation of our baseline model will be **Mean Average Error**, which will determine how far off on average we are from our true target value.

In [ ]:
```
# Using the model to predict
predicted_values = simple_linear_regression_model.predict(test_data_normalised)
# Calculating errors
absolute_errors = np.abs(predicted_values - test_targets)
# Printing the mean of the errors
np.mean(absolute_errors)
```

Out[ ]: 3.4641858124067175

Conclusion

From our linear regressor we had gotten an average error, is this any good though? let's just compare that to a somewhat random value generator

In [ ]:
```
# Lets create random values around the mean and standard diviation of the test data
random_targets = np.random.normal(test_targets.mean(), test_targets.std(), size=len(test_targets))
# Calculating errors
random_absolute_errors = np.abs(random_targets - test_targets)
# Printing the mean of the errors
np.mean(random_absolute_errors)
```

Out[ ]: 10.699916673709504

The linear regressor does do better than just random values. This tells us that there is some statistical power in this problem. We will use the linear regressors MAE as our baseline for the neural network.

Model Development

In this section we will be training our neural network. We will be iteratively training models based on different hyper-peramiters, then evaluate the models based on a few metrics. From here we will be able to decide on an optimal set of hyper-parameters for our model.

This section is split up as follows:

- Iterations setup
- Iterations run
- Iterations evaluation

Iterations Setup

Here we will be creating the iterative steps. Within these steps we will be checking different hyper parameters, we will be recording the performance results for each permutation for further analyses.

Below are the hyper parameters we will be tuning

- Layer quantities and densities
- Activation functions
- Optimizer
- Loss function
- Batch-size

In [ ]:

```
# Helper functions

# buildRegressionModel
def buildRegressionModel(_layers, optimizer='rmsprop', loss='mse', metrics = ['mae']):
    """
    Creates a Sequential neural network.

    Args:
        _layers (array-like): Keras layers
        optimizer (str): Back proporation optimizer : Default = 'rmsprop'
        loss (str): Loss function : Default = 'mse'
        metrics (array-like): Evaluation metrics : Default = ['mae']

    Returns:
        Instance of a sequential model
    """
    # Create model
    model = models.Sequential()
    # Append model layers
    for layer in _layers:
        model.add(layer)
    #  Compile per parameters
    model.compile(optimizer = optimizer, loss = loss, metrics = metrics)
    return model

# crossValidate <Referenced: From course work>
def crossValidate(train_data, train_targets, model_builder, model_attributes, dynamic_learning_rate=False, epochs=1
    """
    Performs cross validation

    Args:
        train_data (array-like) 2D: 2D form of matrix independent variables.
        train_targets (array-like): Dependent variables
        model_builder (function): Function instance of a model builder
        model_attributes (object): Key value pairs of model parameters
        dynamic_learning_rate (boolean): State whether to use the dynamic learning rate, default = False
        epochs (int): Number of epochs to perform, default = 100
        folds (int): Will be 5 fold, this indicates the number of folds to perform, default = 1
        batch_size (int): Batch size for forward propogation, default = 128

    Returns:
        History data: Array-like
    """
    # Factoring the training datas length by the folds
    num_val_samples = len(train_data) // 5
    # Keeping track of all recorded MAEs
    all_history = []
    for i in range(5):
        # Creating the validation index set
        a, b = i * num_val_samples, (i + 1) * num_val_samples
        # Splitting the validation data
        validation_data = train_data[a : b]
        validation_targets = train_targets[a : b]
        # Splitting the training data
        partial_train_data = np.concatenate([train_data[:a], train_data[b:]], axis=0)
```

```
        partial_train_targets = np.concatenate([train_targets[:a], train_targets[b:]], axis=0)
        # Initialising model
        model = model_builder(**model_attributes)
        # Initialise the dynamic learning rate if there is one
        model_callbacks = []
        if dynamic_learning_rate:
            dynamically_reduce_learning_rate = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr
            model_callbacks.append(dynamically_reduce_learning_rate)
        # Train the model and record results
        history = model.fit(partial_train_data,
                        partial_train_targets,
                        validation_data=(validation_data, validation_targets),
                        epochs=epochs,
                        batch_size=batch_size,
                        verbose=0,
                        callbacks=model_callbacks)
        # Recording the MAE
        all_history.append(history)
        if i == (folds - 1):
            break
    return all_history

# layerCreator
def layerCreator(layer_combination, activation):
    """
    Creates a array of keras layers per the layer combination object

    Args:
        layer_combination (array-like): Array of layer combination object.
        activation (str): The activation function for ea layer.

    Returns:
        Array of keras layers: Array-like
    """
    # Creating the layer object
    selected_layers = [
        layers.Dense(layer_combination['nodes'][layer_index], activation = activation, input_shape = (train_data.sh
        if layer_index == 0 else
        layers.Dense(layer_combination['nodes'][layer_index], activation = activation)
        for layer_index in range(layer_combination['layers'])
    ]
    # Adding dropout regularisation per the brief
    selected_layers.insert(1, layers.Dropout(0.2))
    # Adding the output layer
    selected_layers.append(layers.Dense(1))

    return selected_layers

# scoreHistory
def scoreHistory(history, epochs):
    """
    Performs a scoring metric for history objects

    Custom score: Here we identify the distance between the MAE and validation MAE taking into consideration.
    We also weight this value against the lowest valuation MAE

    Args:
        history (array-like): Returned instance from a keras compile.
        epochs (int): Number of epochs

    Returns:
        Scoring object: {
            "min_validation_mae": This relates to the lowest MAE
            "validation_mae_index": This relates to the index of the lowest MAE
            "best_custom_score": This is the custom score
            "best_custom_index": This is the index of the custom score
        }
    """
    # Identifying MAEs
    mae = [np.mean([x[i] for x in [e.history['mae'] for e in history]])for i in range(epochs)]
    validation_mae = [np.mean([x[i] for x in [e.history['val_mae'] for e in history]])for i in range(epochs)]

    # Calculating a custom scoring metric
    best_custom_score = 999
    best_custom_index = 0
    for epoch_index in range(epochs):
        score = abs(mae[epoch_index] - validation_mae[epoch_index]) + validation_mae[epoch_index]
        if score < best_custom_score:
            best_custom_score = score
            best_custom_index = epoch_index

    # Identifying the lowest MAE
    min_validation_mae = min(validation_mae)
    # Stating the index / epoch of the lowest MAE
    validation_mae_index = validation_mae.index(min_validation_mae)

    return {
```

```
        "min_validation_mae": min_validation_mae,
        "validation_mae_index": validation_mae_index,
        "best_custom_score": best_custom_score,
        "best_custom_index": best_custom_index
    }
```

Let's initialise our iterative permutation parameters

In [ ]:
```
# Lets initialise our iterative permutation parameters
iterative_optimizers = ['rmsprop']
iterative_losses = ['mse']
iterative_activations = ['relu', 'tanh']
iterative_batch_sizes = [1,8,16,32]
iterative_layer_combinations = [
    {'layers': 2, 'nodes': [32,32]},
    {'layers': 3, 'nodes': [32,32,32]},
    {'layers': 2, 'nodes': [64,64]},
    {'layers': 3, 'nodes': [64,64,64]},
    {'layers': 2, 'nodes': [128,128]},
    {'layers': 3, 'nodes': [128,128,128]},
]
iterative_epochs = 500
```

Iterations Run

We will be using the results from the iterative evaluations to choose our hyper-parameters, we will also be including a dropout layer for regularisation at this stage, per the brief.

In [ ]:
```
# Identifying the total permutations
total_permutations = (
    len(iterative_optimizers) *
    len(iterative_losses) *
    len(iterative_activations) *
    len(iterative_batch_sizes) *
    len(iterative_layer_combinations)
)
permutation = 0
# Keep records from the iterations
reports = []
# Perform the iterations
for optimizer in iterative_optimizers:
    for loss in iterative_losses:
        for activation in iterative_activations:
            for batch_size in iterative_batch_sizes:
                for layer_combination in iterative_layer_combinations:
                    # Creating the layer object
                    selected_layers = layerCreator(layer_combination, activation)

                    # Creating the attributes object
                    model_attributes = {
                        "_layers": selected_layers,
                        'optimizer': optimizer,
                        'loss': loss
                    }

                    # Print iteration while running
                    permutation += 1
                    print(f"Permutation: {permutation} / {total_permutations}")

                    # Training
                    history = crossValidate(
                        train_data_normalised,
                        train_targets,
                        buildRegressionModel,
                        model_attributes,
                        epochs=iterative_epochs,
                        folds=1,
                        batch_size=batch_size)

                    # Calculating the scorings
                    mae_scoring = scoreHistory(history,iterative_epochs)

                    # Recording the report
                    reports.append({
                        'history': history,
                        'optimizer': optimizer,
                        'loss': loss,
                        'activation': activation,
                        'batch_size': batch_size,
                        'layer_combination': layer_combination,
                        **mae_scoring
                    })
```

```
Permutation: 1 / 48
Permutation: 2 / 48
Permutation: 3 / 48
Permutation: 4 / 48
Permutation: 5 / 48
Permutation: 6 / 48
Permutation: 7 / 48
Permutation: 8 / 48
Permutation: 9 / 48
Permutation: 10 / 48
Permutation: 11 / 48
Permutation: 12 / 48
Permutation: 13 / 48
Permutation: 14 / 48
Permutation: 15 / 48
Permutation: 16 / 48
Permutation: 17 / 48
Permutation: 18 / 48
Permutation: 19 / 48
Permutation: 20 / 48
Permutation: 21 / 48
Permutation: 22 / 48
Permutation: 23 / 48
Permutation: 24 / 48
Permutation: 25 / 48
Permutation: 26 / 48
Permutation: 27 / 48
Permutation: 28 / 48
Permutation: 29 / 48
Permutation: 30 / 48
Permutation: 31 / 48
Permutation: 32 / 48
Permutation: 33 / 48
Permutation: 34 / 48
Permutation: 35 / 48
Permutation: 36 / 48
Permutation: 37 / 48
Permutation: 38 / 48
Permutation: 39 / 48
Permutation: 40 / 48
Permutation: 41 / 48
Permutation: 42 / 48
Permutation: 43 / 48
Permutation: 44 / 48
Permutation: 45 / 48
Permutation: 46 / 48
Permutation: 47 / 48
Permutation: 48 / 48
```

Iterations evaluation

The metrics we will be using for the evaluation of our model will again be the **Mean Average Error**, we will also be evaluating the valuation MAE compared to the training MAE as well as the epochs it took the model to get to its best score.

As we have the records lets evaluate the performance of each record.

We will split this section up as:

- Extract optimal permutations
- Visually analyse
- The best permutation

In [ ]:

```python
# Helper functions

# customScoringMetric
def customScoringMetric(custom_mae, custom_mae_index):
    """
    Calculates a scoring metric for a model, which takes into consideration the lowest MAE, and how many epochs it

    Args:
        custom_mae (double): The mean squared error
        custom_mae_index (int): The index of the lowest MAE

    Returns:
        score : double
    """
    return custom_mae + (custom_mae_index/100)

# smooth_curve <Referenced: From course work>
def smooth_curve(points, factor = 0.9):
    """
    Smooths values of points, by creating a smooth transition between each point.

    Args:
```

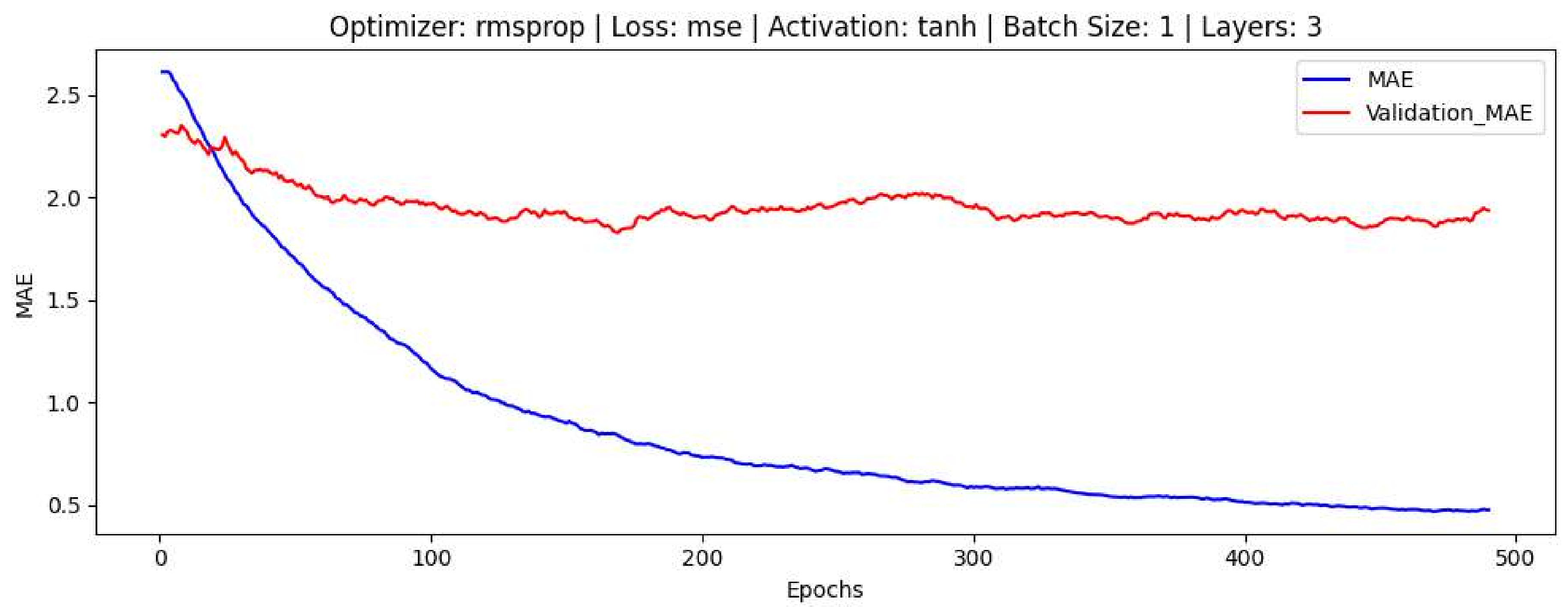
```
        points (array-like): array of doubles.
        factor (double): the smoothing factor

    Returns:
        Smoothed data: Array-like
    """
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

# plotReport <Referenced: From course work>
def plotReport(report, epochs):
    """
    Plots a report objects MAE and validation MAE

    Args:
        report (object): A report object.
        epochs (int): The number of epochs used

    Returns:
        Nothing
    """
    plt.figure(figsize=(10, 4))

    # Averaging MAE
    average_mae_history = [np.mean([x[i] for x in [e.history['mae'] for e in report['history']]])for i in range(epo
    average_validation_mae_history = [np.mean([x[i] for x in [e.history['val_mae'] for e in report['history']]])for
    # Smoothing MAE
    smooth_average_mae_history = smooth_curve(average_mae_history[10:])
    smooth_average_validation_mae_history = smooth_curve(average_validation_mae_history[10:])

    # Plotting the graph
    plt.plot(range(1, len(smooth_average_mae_history) + 1), smooth_average_mae_history, label='MAE', color='blue')
    plt.plot(range(1, len(smooth_average_validation_mae_history) + 1), smooth_average_validation_mae_history, label
    plt.title(f"Optimizer: {report['optimizer']} | Loss: {report['loss']} | Activation: {report['activation']} | Ba
    plt.ylabel('MAE')
    plt.xlabel('Epochs')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

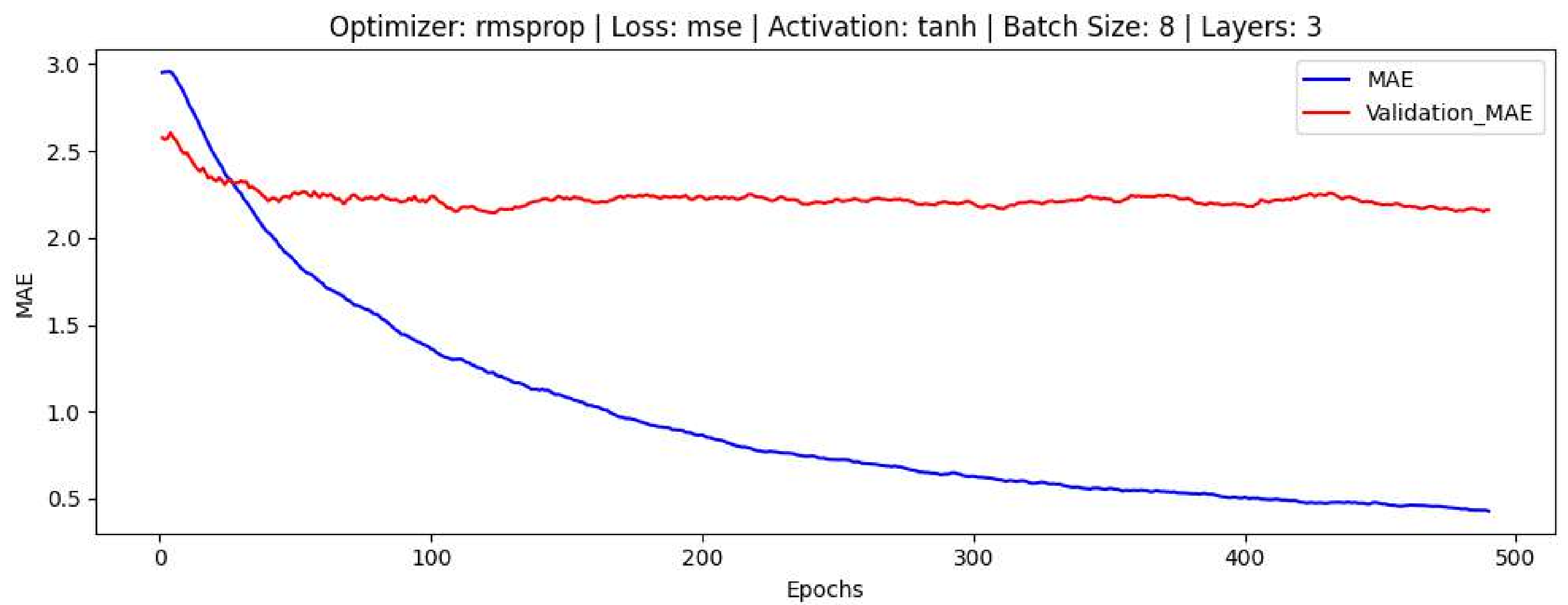
Extract optimal permutations

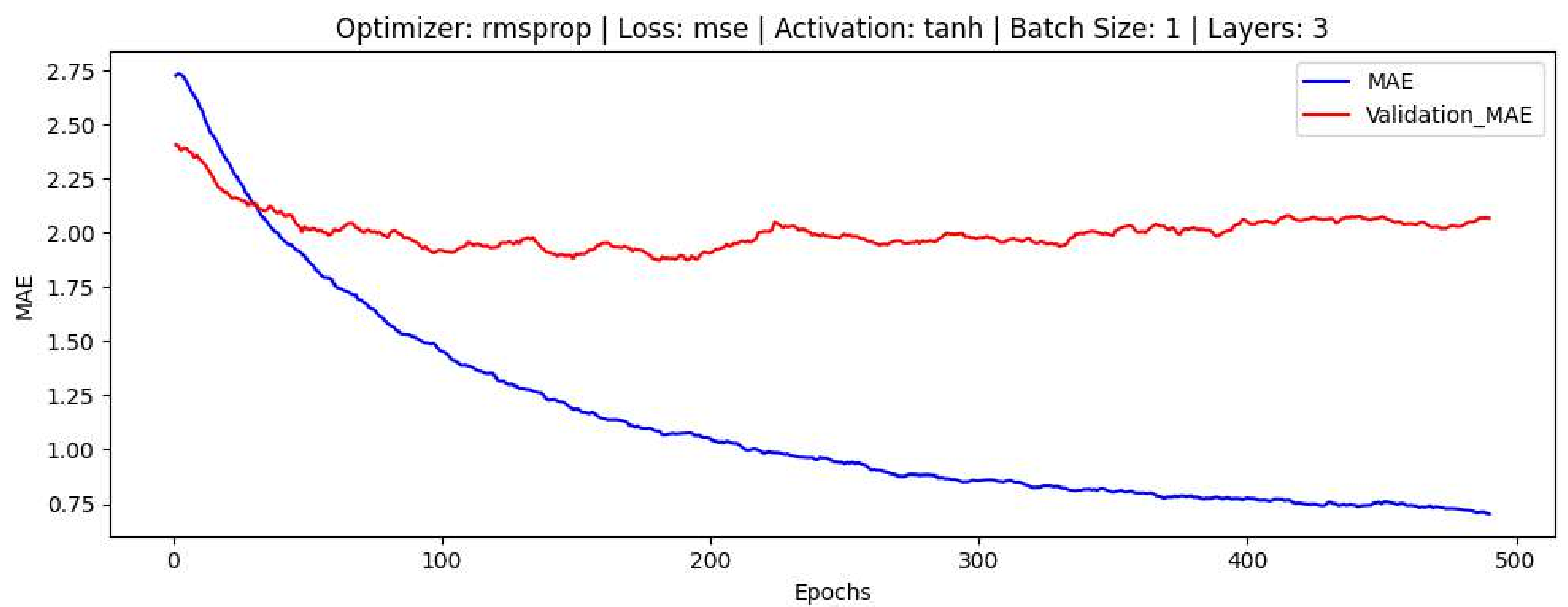
Lets utilise our evaluation metrics to select 3 of the most optimal permutations

In [ ]:
```
# Record the optimal permutations
optimal_permutations = []
optimal_permutation_count = 3
# Analyse each of the iterations reports
for report in reports:
    # Check if the optimal permutations arent full
    if not len(optimal_permutations) == optimal_permutation_count:
        optimal_permutations.append(report)
    else:
        highest_score = 0
        highest_score_index = 0
        for current_optimal_index in range(len(optimal_permutations)):
            score = customScoringMetric(optimal_permutations[current_optimal_index]['best_custom_score'], optimal_p
            if score > highest_score:
                highest_score = score
                highest_score_index = current_optimal_index
        # Replace the record if the current one is better
        if customScoringMetric(report['best_custom_score'], report['best_custom_index']) < highest_score:
            optimal_permutations[highest_score_index] = report
```

In [ ]:
```
# Lets plot the 3 most optimal permutations
for permutation in optimal_permutations:
    plotReport(permutation,iterative_epochs)
```







The best permutation

As per our analyses we can note that the most optimal hyper-parameters for our model will be as below:

In [ ]:
```
# Lets obtain the best scoring record from the regularised models
best_score = 999
best_permutation = None

for permutation in optimal_permutations:
    permutation_score = customScoringMetric(permutation['best_custom_score'], permutation['best_custom_index'])
    if permutation_score < best_score:
        best_score = permutation_score
        best_permutation = permutation

print(
    "Best hyper-parameters\n"+
    "====================================\n"+
    f"Activation: {best_permutation['activation']}\n"+
    f"Batch Size: {best_permutation['batch_size']}\n"+
    f"Layer Combination: {best_permutation['layer_combination']}\n"+
    f"Loss: {best_permutation['loss']}\n"+
```

```
    f"Optimizer: {best_permutation['optimizer']}\n"+
    f"Min validation MAE: {best_permutation['min_validation_mae']}\n"+
    f"Min validation MAE index: {best_permutation['validation_mae_index']}\n"+
    f"Best Custom Score: {best_permutation['best_custom_score']}\n"+
    f"Best Custom Score Index: {best_permutation['best_custom_index']}\n"
)
```

```
Best hyper-parameters
======================================
Activation: tanh
Batch Size: 1
Layer Combination: {'layers': 3, 'nodes': [128, 128, 128]}
Loss: mse
Optimizer: rmsprop
Min validation MAE: 1.6980584859848022
Min validation MAE index: 176
Best Custom Score: 1.9087151288986206
Best Custom Score Index: 41
```

Finalizing Model

Now that we have determined which are the best hyper parameters for our model, lets train our model on the entire dataset.

This section is split up as follows:

- Creating our model
- Train and evaluate our model

Creating our model

We will be using the hyper-parameters from the previous section while creating this model.

Here we will be creating our model and training our model on the full training set.

In [ ]:
```
# Creating the layer object
optimal_selected_layers = layerCreator(best_permutation['layer_combination'], best_permutation['activation'])

# Lets build our model
final_model = buildRegressionModel(optimal_selected_layers, optimizer=best_permutation['optimizer'], loss=best_perm
```

In [ ]:
```
# Lets train our model
final_model.fit(train_data_normalised,
                train_targets,
                epochs=best_permutation['best_custom_index'],
                batch_size=best_permutation['batch_size'],
                verbose=0)
```

Out[ ]:
```
<keras.src.callbacks.History at 0x7f2dc79ec0d0>
```

In [ ]:
```
# Lets evaluate our model
test_mse_score, test_mae_score = final_model.evaluate(test_data_normalised, test_targets, verbose=0)

# Final MAE
print(test_mae_score)
```

```
2.782836675643921
```

Conclusion

Lets compare our results to the linear regression model.

- Linear regressor = $3464
- Neural network = $2782

The neural network did better, but not by much. And lets not forget if the linear regression model was worked on and had its independent variables dimensionallity reduced for the highly correlated variables then the linear regression model would do even better.

Training the neural network had less steps involved, though at a cost of higher computational resources. So linear regression models arent out of the picture yet.