UNIVERSITY OF PÉCS
FACULTY OF SCIENCES
INSTITUTE OF MATHEMATICS AND INFORMATICS

ESTIMATING THE PARTIAL CHARGES IN ORGANIC MOLECULES VIA MACHINE LEARNING

Supervisor: Dr. Tamás Kégl
Full Professor

Author: Nabil Arrouss (VUZDOW)
Computer Science, BSc

Pécs, 2024

Table of Contents

I. Abstract

Understanding the partial charges within organic molecules is crucial for predicting their behavior and interactions. However, traditional methods for calculating these charges may have limitations due to reliance on software proprietary.

The goal of the thesis is to develop a machine learning model that delivers accurate estimations of partial charges in organic molecules using molecular geometries as input data.

Firstly, in the introduction and literature review chapters, the thesis introduces the significance of charge distribution in organic chemistry and presents the research goals along with a comprehensive background focusing on existing methods for expressing partial charges. Moreover, highlighting the motivation for a machine learning-based approach to overcome these limitations and deliver accessible and accurate estimations of partial charges.

The materials and development chapter will walk us through the tools utilized and provides a description of the dataset, including details on definition of geometric features, and data preparation. Also, the chapter details the process of selecting a machine learning model, and the neural network architecture. Further, it delves into training and validation steps, discussing the results obtained through the initial steps, and the hyperparameters tuning process to re-fine the model capabilities. Finally, the chapter concludes with an evaluation of the developed model.

The results chapter presents the predictions generated by the developed machine learning model and provides analysis of the accuracy and reliability of the model in estimating partial charges in organic molecules.

Finally, the conclusion chapter summarizes the key findings of the thesis.

II. Introduction and research goals

In this chapter, we will go through the definition of partial charges, their importance in chemistry and what are the methods used to calculate them precisely. Then, we will discover what are the limitations that come up when using those methods to calculate them and how machine learning models demonstrated potential to overcome these limitations. Finally, I state the goal of this thesis for developing a machine learning model.

1. Partial charges in organic molecules

“Partial charges are numerical values assigned to atoms within a molecule. These charges represent the distribution of electron density”. (Raček T et al., 2020)

“Understanding partial charge is important and essential in chemistry. Now, imagine a molecule as a central nucleus surrounded by an electron cloud. In this structure, it becomes difficult to precisely determine a specific partial charge because of the complex distribution of electrons”. (Meaning and Definition of Partial Charges, n.d.)

“We can’t physically measure these charges, but they are crucial for many chemical theories and theoretical models. That’s why they are often determined from experimental results or computed using quantum chemistry techniques”.(Uhliar, 2024)

“Basically, a molecule’s shape and its electron distribution dictate how it behaves in terms of chemical reactivity and physical properties, which is why it is important to understand them”.(Uhliar, 2024)

“Precise determination of atomic partial charges is essential in many fields of chemistry, for example accurate charges in quantum chemistry are required to calculate molecular energies, optimize structures, and predict chemical reactions”. (Brehm & Thomas, 2021) In general, charge distribution within a molecule is of highest importance in terms of both reactivity and physical properties.

2. Limitations and motivation

“Regardless of their significance, partial charges cannot be directly measured experimentally. Also, there is no universally accepted best approach for calculating these charges. Consequently, a wide range of methods to define them has been proposed. One of the most widely used technique is Natural Population Analysis (NPA)”.(Partial Atomic Charge Derivation of Small Molecule, n.d.)

“Accurate partial charge calculations are possible, but these methods can be computationally demanding and often rely on expensive software. Thus, how to predict these charges accurately and efficiently has been a challenge in computational chemistry”.(J. Wang et al., n.d.) Hence, a simplified machine learning model that results in acceptable agreements with the quantum chemical calculations would be a valuable addition to a chemist’s toolbox.

3. Estimating partial charges via machine learning

“Over the past 20 years, many machine learning approaches have been used to predict various molecular properties without needing a deep understanding of the fundamental philosophy of chemistry. During the last few years, ML models demonstrated their abilities to provide reliable predictions for partial charges by training pre-collected data of high-level quantum mechanics charges”. (J. Wang et al., n.d.)

The goal of this thesis is to develop a machine learning model, precisely a neural network model that can estimate partial charges utilizing molecular geometries solely as input.

III. Literature Review

This chapter will take us briefly through a historical overview of traditional methods used for calculating partial charges and how they can be very computationally high cost, to the age of machine learning, where new approaches offer solutions to the challenges of traditional methods. Also, we will investigate a couple of machine learning applications used for this kind of task.

1. Challenges of Traditional Partial Charge Calculation Methods

“The idea of partial charges first emerged in chemistry to help explain how atoms react with each other. Later, these charges became widely used in computational chemistry, as well as in cheminformatics and bioinformatics. Over time scientists developed several methods to calculate these charges, and the commonly used ones are Mulliken Population Analysis, Natural Population Analysis (NPA), and electrostatic potential (ESP). Quantum chemistry offers the most accurate approach for calculating partial charges. In fact, these approaches are computationally expensive in general and therefore faster methods were developed”. (Tom´ et al., 2020b) Nevertheless, even these developed approaches can face computational bottlenecks or rely on closed-source software. This motivates the exploration of machine learning for open-source and computationally efficient partial charge prediction.

2. Machine Learning for Partial Charge Prediction

"More recently, machine learning (ML) has emerged as a powerful tool for chemistry in general, including the prediction of partial charges in organic chemistry". (Artrith et al., n.d.) "However, to effectively learn, ML models require suitable representations of molecular data. Various machine learning methods, including neural networks, have been proposed based on the representation form of molecules". (GRAMMAR-INDUCED GEOMETRY FOR DATA-EFFICIENT MOLECULAR PROPERTY PREDICTION, n.d.) ”We can represent molecules as graphs, strings, precomputed feature vectors or sets of atomic coordinates". (Heid et al., 2024)
There are many applications that use machine learning methods to predict partial charges, for example:

1. ContraDRG: "A software designed for estimating partial charges in small molecules within seconds and high accuracy by applying several machine learning techniques such as: linear regression, random forests and many more".(Martin & Heider, 2019)
2. EspalomaCharge: "Employs a graph neural network architecture to predict charges of molecules rapidly and accurately. The model was trained on large datasets of molecules with partial charges calculated using quantum chemistry methods". (Y. Wang, n.d.)

The machine learning model I have developed follows the neural network approach to estimate partial charges in organic molecules using solely the various aspects of the molecular geometry data. This model can be a very useful tool for scientists and researchers in general, particularly for chemists. Furthermore, it offers additional opportunities for further development and research.

The model (codes and all development steps) can be found here: LINK

IV. Materials and methods

This chapter will discuss the tech stack used as tools for the development of the model, alongside the dataset source. I will cover a complete overview of the methods followed and the step-by-step process used to build the model. We will explore everything from data preparation and analysis to model training, refinement with hyperparameters, evaluation, and finally, saving the model.

1. Tools

For developing a good machine learning model, I choose JupyterLab which is a powerful open-source software that provides an interactive environment where you can write code, run it, see the results, modify it, and run it again quickly. "Its interactive nature is highly compatible with the iterative model building and testing processes of machine learning".(JupyterLab Documentation — JupyterLab 4.2.0rc0 Documentation, n.d.)

The programming language I used for coding was Python. "It is a powerful tool known for its simplicity, readability, and clear syntax". (About PythonTM | Python.Org, n.d.)"It provides vast libraries which offer tools for data manipulation, analysis, and developing machine learning models". (Machine Learning with Python Tutorial, n.d.)

For different tasks, I utilized appropriate libraries or frameworks, each designed for its specific purpose, here's a list of all the tools used, along with an explanation of their purpose:

For exploratory data analysis (EDA), data manipulation and scientific computing:

- ➢ pandas: "Used for data manipulation and analysis".(Pandas - Python Data Analysis Library, n.d.)
- ➢ NumPy: "Serves as the foundation for scientific computing with Python".(NumPy -, n.d.)
- ➢ SciPy: "Employed for scientific and technical computing".(SciPy -, n.d.)
- ➢ For visualization and statistical data visualization:
- ➢ Matplotlib: "Used for creating plots".(Matplotlib — Visualization with Python, n.d.)
- ➢ Seaborn: "Library for statistical data visualization".(Waskom, 2021)

For machine learning relative work:

- ➢ scikit-learn: "Used for data scaling and dataset splitting".(Scikit-Learn: Machine Learning in Python — Scikit-Learn 1.4.2 Documentation, n.d.)

- TensorFlow: “Powerful framework for machine learning, providing a lower-level interface which is easy-to-use. I chose this for building the machine learning model due to its simplicity and flexibility”.(TensorFlow, n.d.)
- Keras: “A high-level neural networks API, serving as an interface for TensorFlow. It is a user-friendly interface for building and training neural networks.”. (Keras: Deep Learning for Humans, n.d.)
- KerasTuner: “Offers hyperparameters optimization with an easy-to-use framework”.(KerasTuner, n.d.)
- tensorflow_docs: “A TensorFlow package for creating nicely formatted documentation directly from the TensorFlow code”. (API Documentation | TensorFlow v2.16.1, n.d.)

2. Dataset

The dataset is generated from a database that was developed at the Chemistry Department of the University of Pécs, through extensive quantum chemical calculations on 3427 organic molecules. The dataset has a shape of (65076, 20), representing 65076 atoms. Each atom has 20 data points describing its molecular geometry and computed partial charges. Down below I list the 20 column definitions, highlighting the columns used as features and the target columns:

- ‘name’ is the file name representing the molecule’s name (1 column).

Targets (3 columns):

- ‘cm5’, ‘esp’, ‘npa’: These are the computed charges.

Features (16 columns):

- ‘cm_d’: Is the diagonal elements of the Coulomb matrix belonging to the given atom.
- ‘cm_od1’, ‘cm_od2’, ‘cm_od3’: They are the largest off diagonal Coulomb matrix elements for the atom.
- ‘dist1’, ‘dist2’, ‘dist3’, ‘dist4’: These columns represent the smallest interatomic distances for the atom.
- ‘ang1’, ‘ang2’, ‘ang3’, ‘ang4’, ‘ang5’, ‘ang6’: The typical angles for the given atom.
- ‘ang_nb1’, ‘ang_nb2’: The two typical angles for the closest neighboring atom.

The dataset was fully prepared and organized in a Comma Separated Values format (CSV).

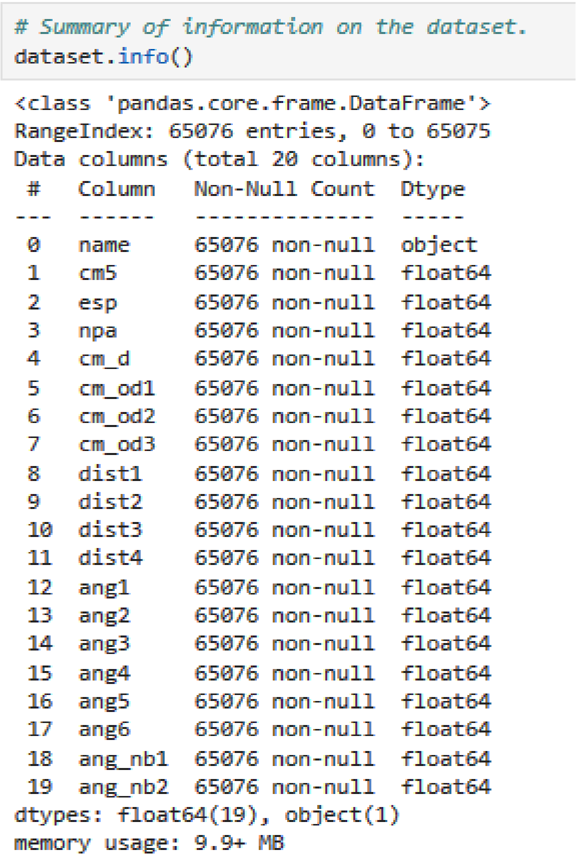
2.1 Data preparation and data analysis

After setting up the JupyterLab environment and installing all the softwares and libraries mentioned above, I imported pandas, NumPy, SciPy, Matplotlib and Seaborn as the initial libraries to work with. I wrote a simple code like this:

```
import pandas as pd
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
```

Firstly, I loaded the CSV file data into a dataframe using pandas and started with exploratory data analysis. The codes below show how to load the dataset and how to display various information about the dataset are shown in the Figure 1:

```
path = ‘Data_Charges.csv’
dataset = pd.read_csv(path)
```



```
# Summary of information on the dataset.
dataset.info()
```

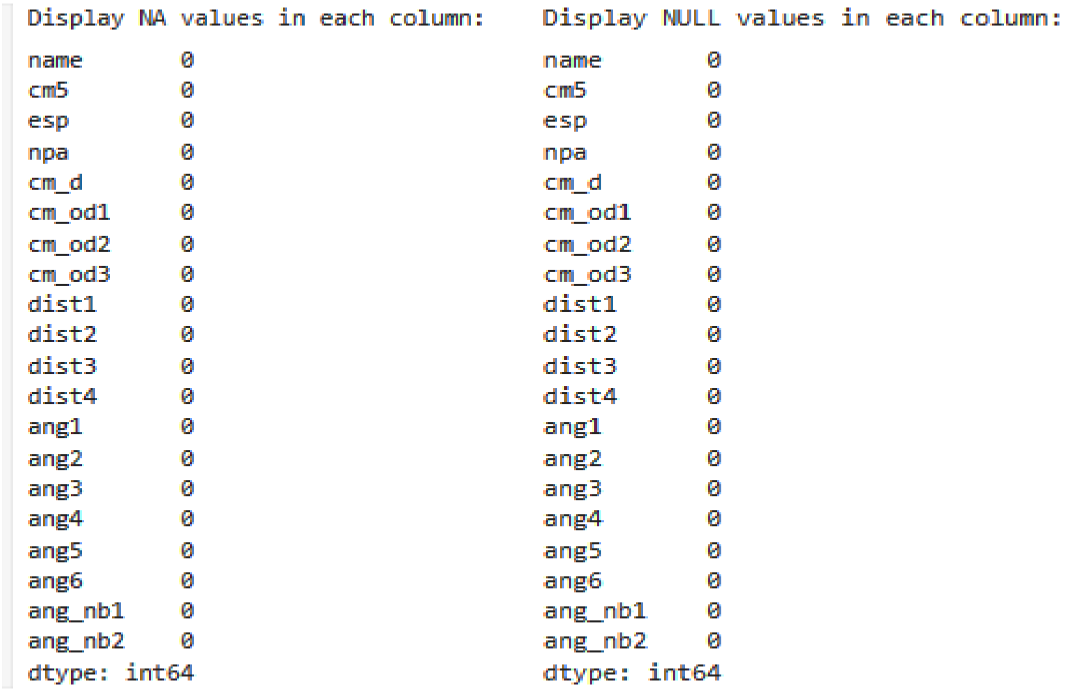
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65076 entries, 0 to 65075
Data columns (total 20 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   name     65076 non-null  object 
 1   cm5      65076 non-null  float64
 2   esp      65076 non-null  float64
 3   npa      65076 non-null  float64
 4   cm_d     65076 non-null  float64
 5   cm_od1   65076 non-null  float64
 6   cm_od2   65076 non-null  float64
 7   cm_od3   65076 non-null  float64
 8   dist1    65076 non-null  float64
 9   dist2    65076 non-null  float64
 10  dist3    65076 non-null  float64
 11  dist4    65076 non-null  float64
 12  ang1     65076 non-null  float64
 13  ang2     65076 non-null  float64
 14  ang3     65076 non-null  float64
 15  ang4     65076 non-null  float64
 16  ang5     65076 non-null  float64
 17  ang6     65076 non-null  float64
 18  ang_nb1  65076 non-null  float64
 19  ang_nb2  65076 non-null  float64
dtypes: float64(19), object(1)
memory usage: 9.9+ MB
```

Figure 1: summary of information on the dataset.

Then I checked if these columns contain NA values or NULL values, the 2 codes below used for this task:

```
print("Display NA values in each column: ")
dataset.isna().sum(axis=0)
print("Display NULL values in each column: ")
dataset.isnull().sum()
```

The Figure 2 below show the outputs of the 2 codes above respectively.



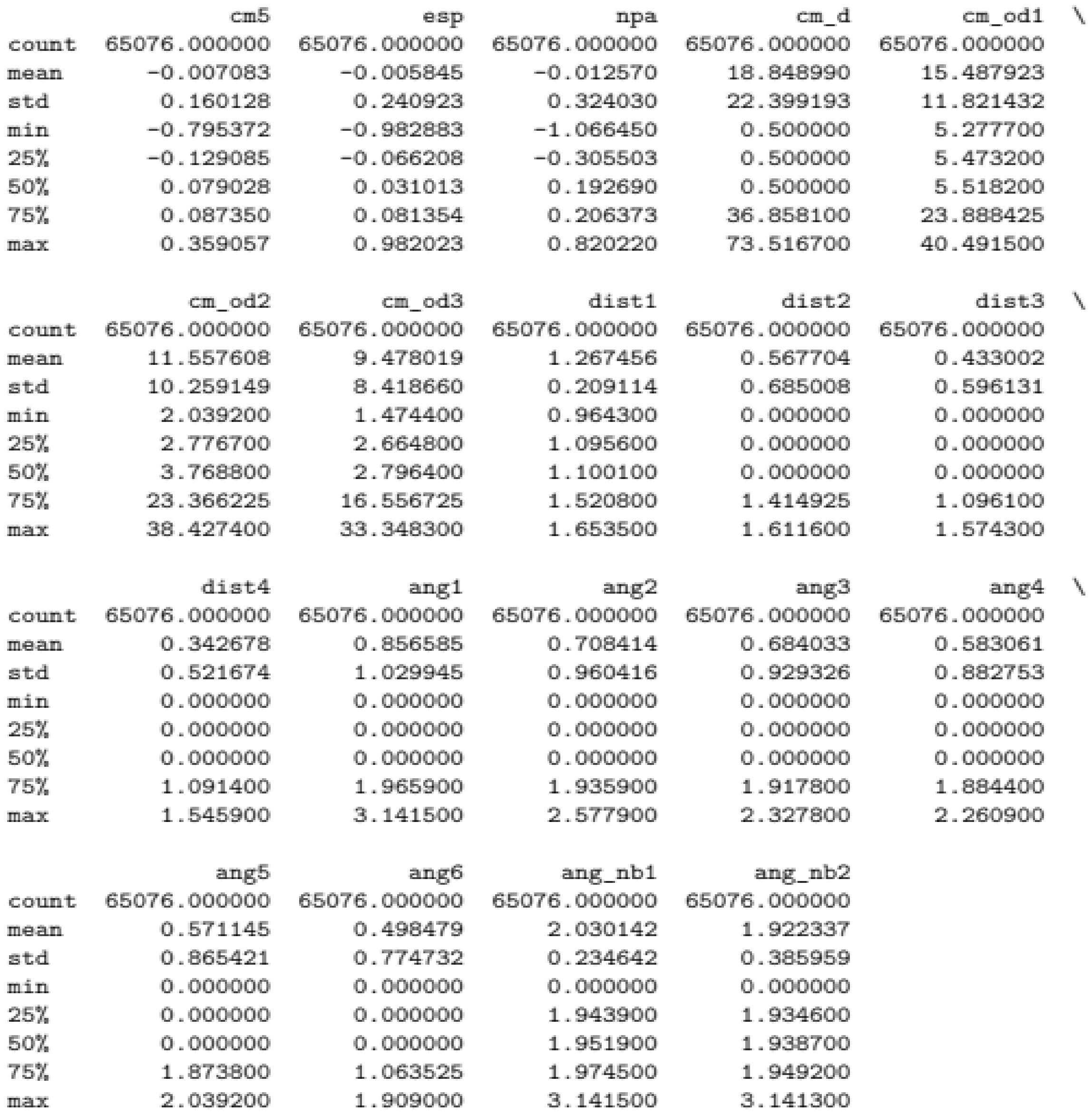
```
Display NA values in each column:    Display NULL values in each column:
name       0                         name       0
cm5        0                         cm5        0
esp        0                         esp        0
npa        0                         npa        0
cm_d       0                         cm_d       0
cm_od1     0                         cm_od1     0
cm_od2     0                         cm_od2     0
cm_od3     0                         cm_od3     0
dist1      0                         dist1      0
dist2      0                         dist2      0
dist3      0                         dist3      0
dist4      0                         dist4      0
ang1       0                         ang1       0
ang2       0                         ang2       0
ang3       0                         ang3       0
ang4       0                         ang4       0
ang5       0                         ang5       0
ang6       0                         ang6       0
ang_nb1    0                         ang_nb1    0
ang_nb2    0                         ang_nb2    0
dtype: int64                         dtype: int64
```

Figure 2: NA values and NULL values check.

During this initial analysis I explored the statistical summary of each column to gain more insights. The Figure 3 Figure 3down below demonstrates the output summary of this single line code:

```
Dataset.describe()
```

The observation I made on this summary was that the target columns can take both positive and negative values. This insight was helpful for designing the model's architecture to be able predict both negative and positive values.



```
                cm5           esp           npa          cm_d        cm_od1  \
count  65076.000000  65076.000000  65076.000000  65076.000000  65076.000000
mean      -0.007083     -0.005845     -0.012570     18.848990     15.487923
std        0.160128      0.240923      0.324030     22.399193     11.821432
min       -0.795372     -0.982883     -1.066450      0.500000      5.277700
25%       -0.129085     -0.066208     -0.305503      0.500000      5.473200
50%        0.079028      0.031013      0.192690      0.500000      5.518200
75%        0.087350      0.081354      0.206373     36.858100     23.888425
max        0.359057      0.982023      0.820220     73.516700     40.491500

             cm_od2        cm_od3         dist1         dist2         dist3  \
count  65076.000000  65076.000000  65076.000000  65076.000000  65076.000000
mean      11.557608      9.478019      1.267456      0.567704      0.433002
std       10.259149      8.418660      0.209114      0.685008      0.596131
min        2.039200      1.474400      0.964300      0.000000      0.000000
25%        2.776700      2.664800      1.095600      0.000000      0.000000
50%        3.768800      2.796400      1.100100      0.000000      0.000000
75%       23.366225     16.556725      1.520800      1.414925      1.096100
max       38.427400     33.348300      1.653500      1.611600      1.574300

              dist4          ang1          ang2          ang3          ang4  \
count  65076.000000  65076.000000  65076.000000  65076.000000  65076.000000
mean       0.342678      0.856585      0.708414      0.684033      0.583061
std        0.521674      1.029945      0.960416      0.929326      0.882753
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%        0.000000      0.000000      0.000000      0.000000      0.000000
50%        0.000000      0.000000      0.000000      0.000000      0.000000
75%        1.091400      1.965900      1.935900      1.917800      1.884400
max        1.545900      3.141500      2.577900      2.327800      2.260900

               ang5          ang6       ang_nb1       ang_nb2
count  65076.000000  65076.000000  65076.000000  65076.000000
mean       0.571145      0.498479      2.030142      1.922337
std        0.865421      0.774732      0.234642      0.385959
min        0.000000      0.000000      0.000000      0.000000
25%        0.000000      0.000000      1.943900      1.934600
50%        0.000000      0.000000      1.951900      1.938700
75%        1.873800      1.063525      1.974500      1.949200
max        2.039200      1.909000      3.141500      3.141300
```

Figure 3: Statistical summary of each column.

After the general analysis of the dataset, I moved to investigate our features and target values. Firstly, I selected the features from the dataset using indexing method, stored them in a variable named ‘features’, and displayed them as a list with all these features names. Then I plotted a heatmap to find correlation between these features. See code under, which was used to display the features selected:

```
features = dataset.columns[4:]
print("Features ==>\n", features.tolist())
```

The output of this code is a list of the feature columns names which I used further to plot the heatmap. The resulted list is like this:

```
Features ==>
['cm_d', 'cm_od1', 'cm_od2', 'cm_od3', 'dist1', 'dist2', 'dist3', 'dist4',
'ang1', 'ang2', 'ang3', 'ang4', 'ang5', 'ang6', 'ang_nb1', 'ang_nb2']
```

For the plotting part, this code below is used:

```
plt.figure(figsize = (8,5))
correlation_matrix = dataset.iloc[:, 4:].corr()
sns.heatmap(correlation_matrix, cbar = True, annot = True,
fmt=".1f", cmap='coolwarm')
plt.title('Features Autocorrelation Heatmap')
plt.show()
```

This code simply specifies the plot size, calculates the correlation matrix of the selected features, and plots a heatmap. The Figure 4 below shows the heatmap.

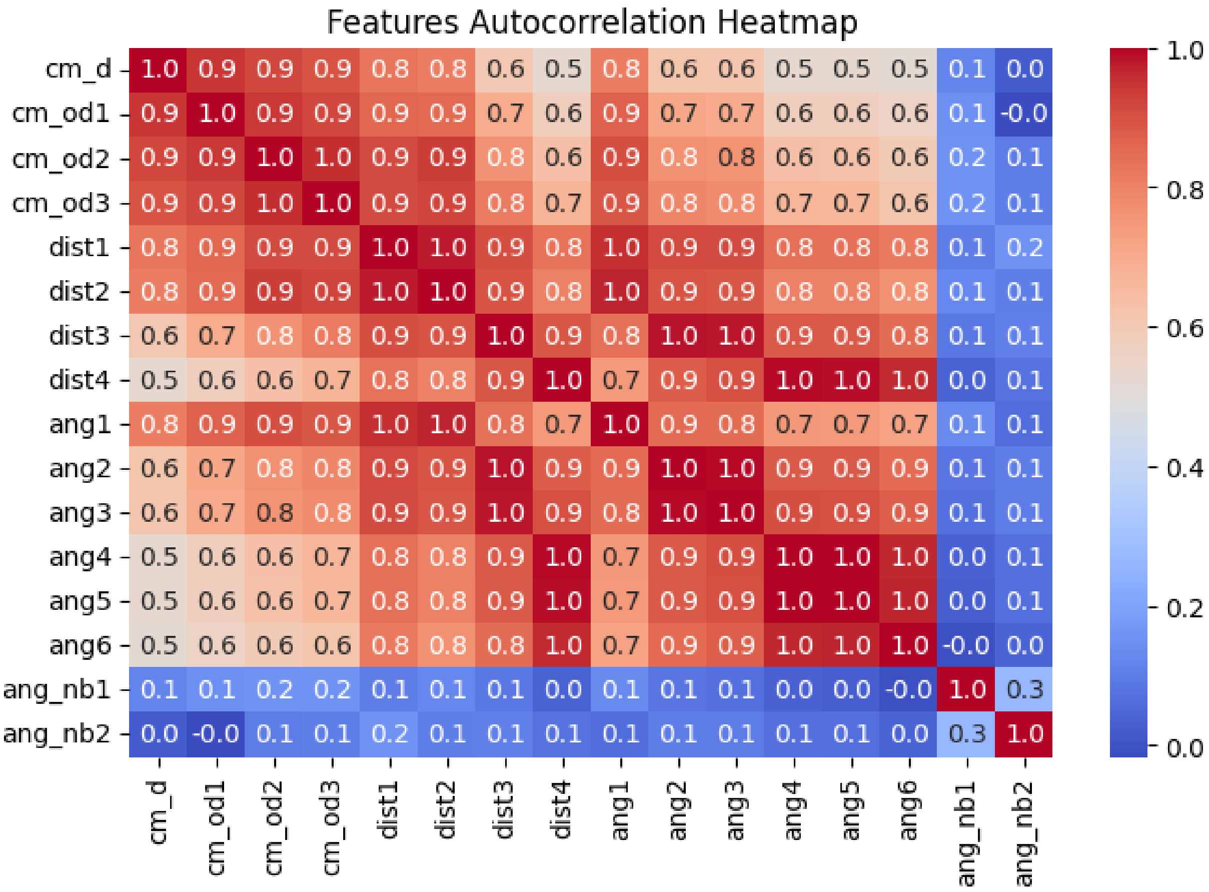


Figure 4: Features Correlation Heatmap

The heatmap displays correlations between the features. It reveals some strong positive correlations among feature pairs like ‘cm_od1’ and ‘cm_od2’, as well as several moderately positive correlations. No very strong negative correlations are immediately apparent.

Further analysis on the features was conducted by creating histograms with probability density function curve (PDF) to visualize the distributions of these features. I used this code for that:

```
n_rows=4
n_cols=4
fig, axes = plt.subplots(nrows=n_rows, ncols=n_cols)
fig.set_size_inches(14, 12)
for i, column in enumerate(dataset.iloc[:,4:].columns):
    sns.histplot(dataset[column], ax=axes[i//n_cols,
      i % n_cols], kde=True, color='skyblue')
plt.tight_layout()
plt.show()
```

The code basically plots a histogram for each feature, so we would generate 16 histograms to visualize the distribution of each feature.

The Figure 5 in the next page represents the distributions of the features using histograms and probability density functions plots. These histograms reveal that all the features, excluding ‘ang_nb1’ and ‘ang_nb2’, predominantly have a high concentration of counts at very low values with notable spikes, with fewer instances occurring as values increase. The patterns observed here are indicative of highly skewed distributions. ’ang_nb1’ shows a spread of data points across a wider range, while ‘ang_nb2’ displays an almost uniform distribution among specific values.

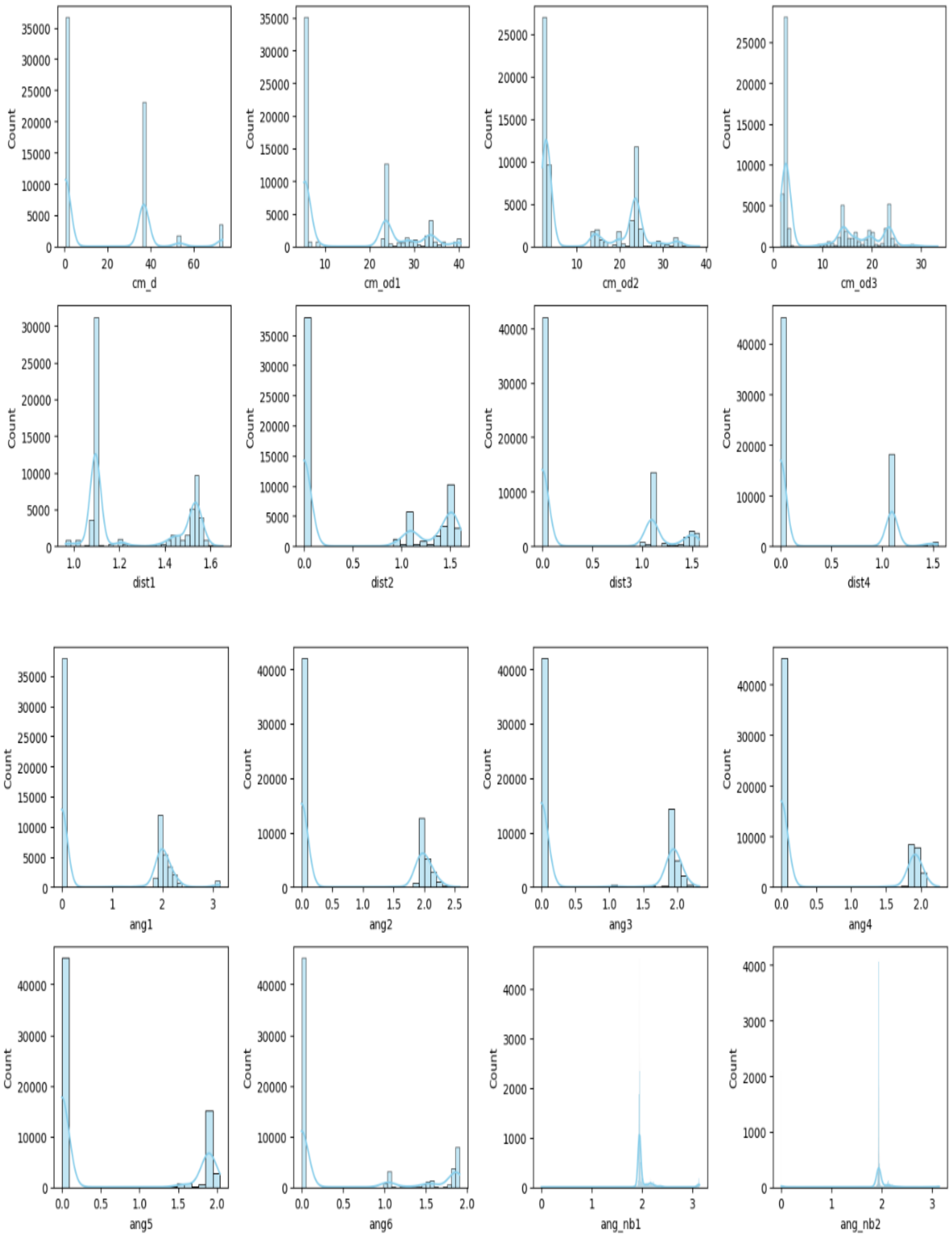


Figure 5: Histograms with (PDF) curve of the features.

After analyzing the features, I started analyzing the targets, the partial charges in our case. Again, I extracted them from the dataset using indexing, stored them in a variable called ‘targets’, and printed them as a list of names.

```
 targets = dataset.columns[1:4]
 print("Target molecule charges ==> ", targets.tolist())
```

The code above shows how to extract the target values, store them, and print them. the output is this: `Target molecule charges ==>  ['cm5', 'esp', 'npa']`

I analyzed them by plotting the distributions of each molecule charge. For that, I defined a function called `plot_target_distribution` which takes 2 input arguments, namely a “dataframe” and an integer index variable named “moleculecharge”. The function takes the dataset and gets the name of the partial charge column based on the provided index to store it in a variable, then it extracts the data of that charge column, sorts it, and converts it to a NumPy array. It calculates the mean and the standard deviation for the charge to use them for generating a probability density function modeled as a normal distribution. Finally, it creates the plot based on the previously gathered information . To create this function, see the code below:

```
def plot_target_distribution(dataframe:pd.DataFrame,moleculecharge=1):
    # Get name of the partial charge based on its index
    molecule_charge_name = dataframe.columns[moleculecharge]
    # Extract the values for the specified charge
     values = dataframe.iloc[:, moleculecharge].sort_values().values
    # Calculate mean and standard deviation for the molecule Charge
    mean = values.mean()
    std = values.std()
    # Create a probability density function
    pdf = scipy.stats.norm.pdf(values, loc=mean, scale=std)
    # Plotting
    plt.plot(values, pdf , label="Distribution curve")
    plt.title(f'Distribution of "{molecule_charge_name}"',weight='bold')
    plt.xlabel('Values')
    plt.ylabel('Probability Density Function')
    plt.xlim([-1.2,1.2])
    plt.grid(True, alpha=0.5, linestyle="--")
    plt.vlines(x = mean, ymin = 0, ymax = max(pdf),
            colors = 'red',
            label = 'Mean Value',
            linestyles='dashed')
    plt.legend(loc="upper left")
    plt.show()
    pass
```

This way, this custom function was used effectively to generate 3 plots for the distribution of target molecule charges, only by passing the dataset and setting the index number of the desired column target. Here is how I used it by code:

```
# Plotting the distribution for each target molecule charge
plot_target_distribution(dataframe=dataset, moleculecharge=1)
plot_target_distribution(dataframe=dataset, moleculecharge=2)
plot_target_distribution(dataframe=dataset, moleculecharge=3)
```

The Figure 6 below represents distribution of ‘cm5’ charge. The mean value of the distribution, indicated by the red dashed line, is centrally located at 0, suggesting that the data is normally distributed with no skewness.

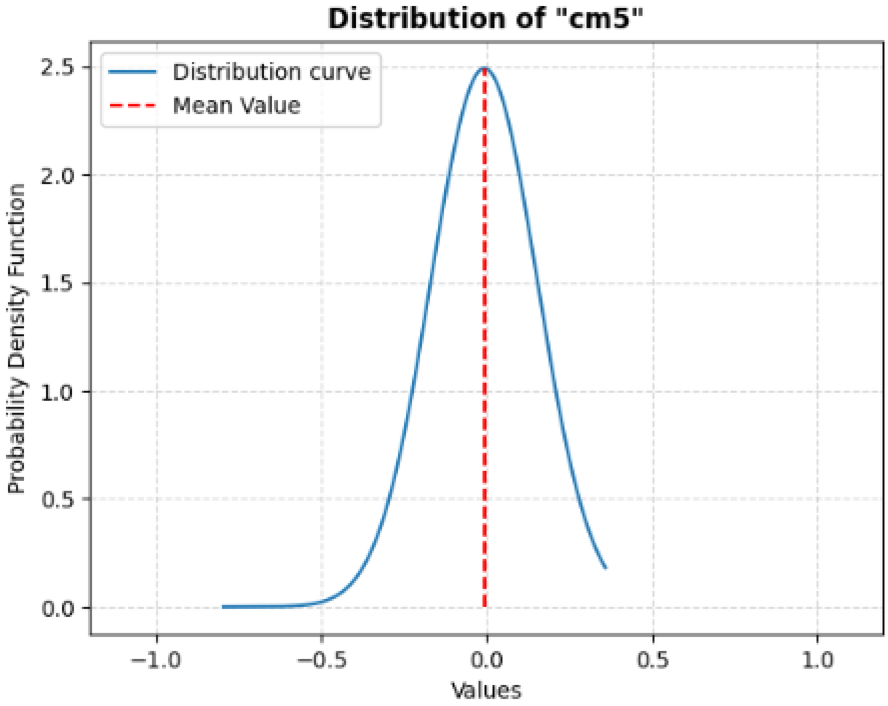


Figure 6: Distribution of 'cm5' charge.

The Figure 7 below illustrates the probability density function for the charge ‘esp’. The mean value is prominently marked by the red dashed line at 0, suggesting that the distribution is symmetric and centered at this point.

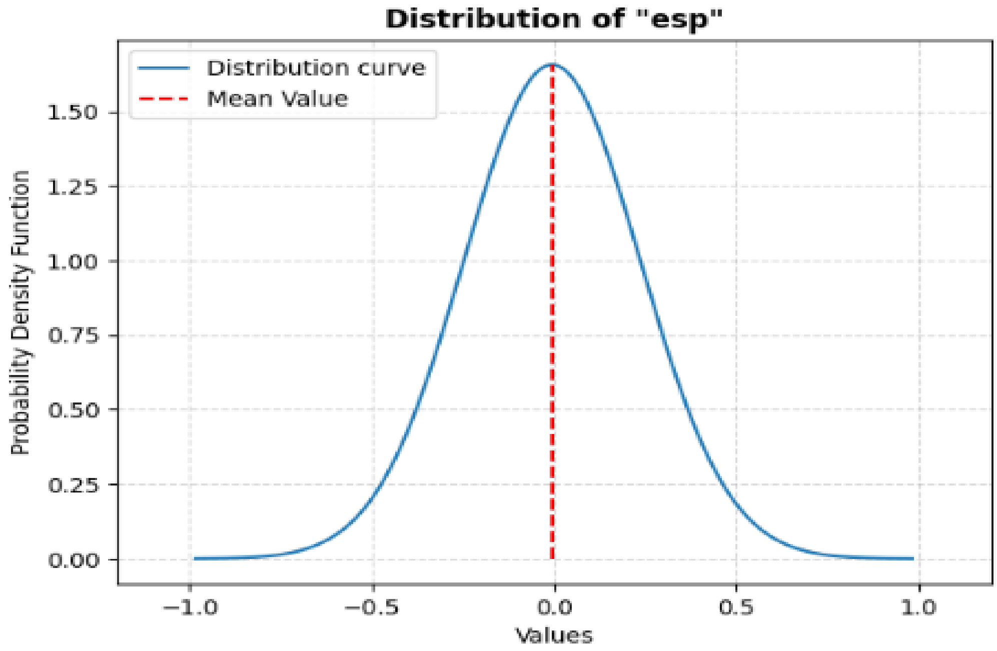


Figure 7: Distribution of 'esp' charge.

Figure 8 underneath is a graph representing the probability density function for the charge ‘npa’, characterized by a symmetric curve. The mean of the distribution is indicated by the red dashed line at the center, precisely at 0, reflecting the balance and symmetry in the data around this central value.

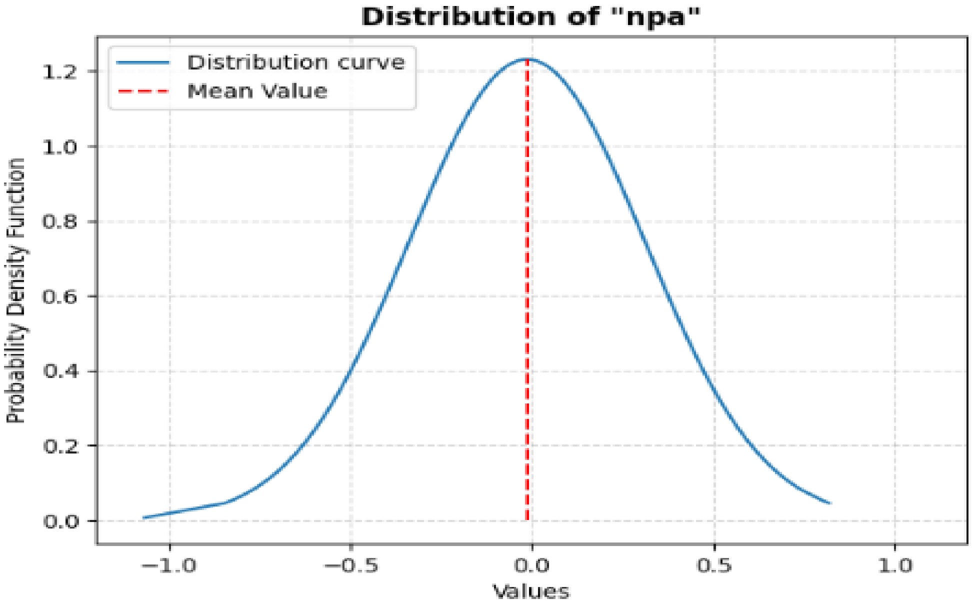


Figure 8: Distribution of 'npa' charge.

2.2 Data normalization and splitting

In this section, we will go through how I normalized the data and how the dataset splitting was done. However, before that I randomly shuffled the data. Shuffling helps reduce variance and support model generalization by preventing overfitting. (Neural Network - Why Should the Data Be Shuffled for Machine Learning Tasks - Data Science Stack Exchange, n.d.) The code below was used for shuffling the dataset:

```
RANDOM_SEED = 42
shuffled_dataset = dataset
shuffled_dataset=shuffled_dataset.sample(frac=1,random_state=RANDOM_S
EED)
```

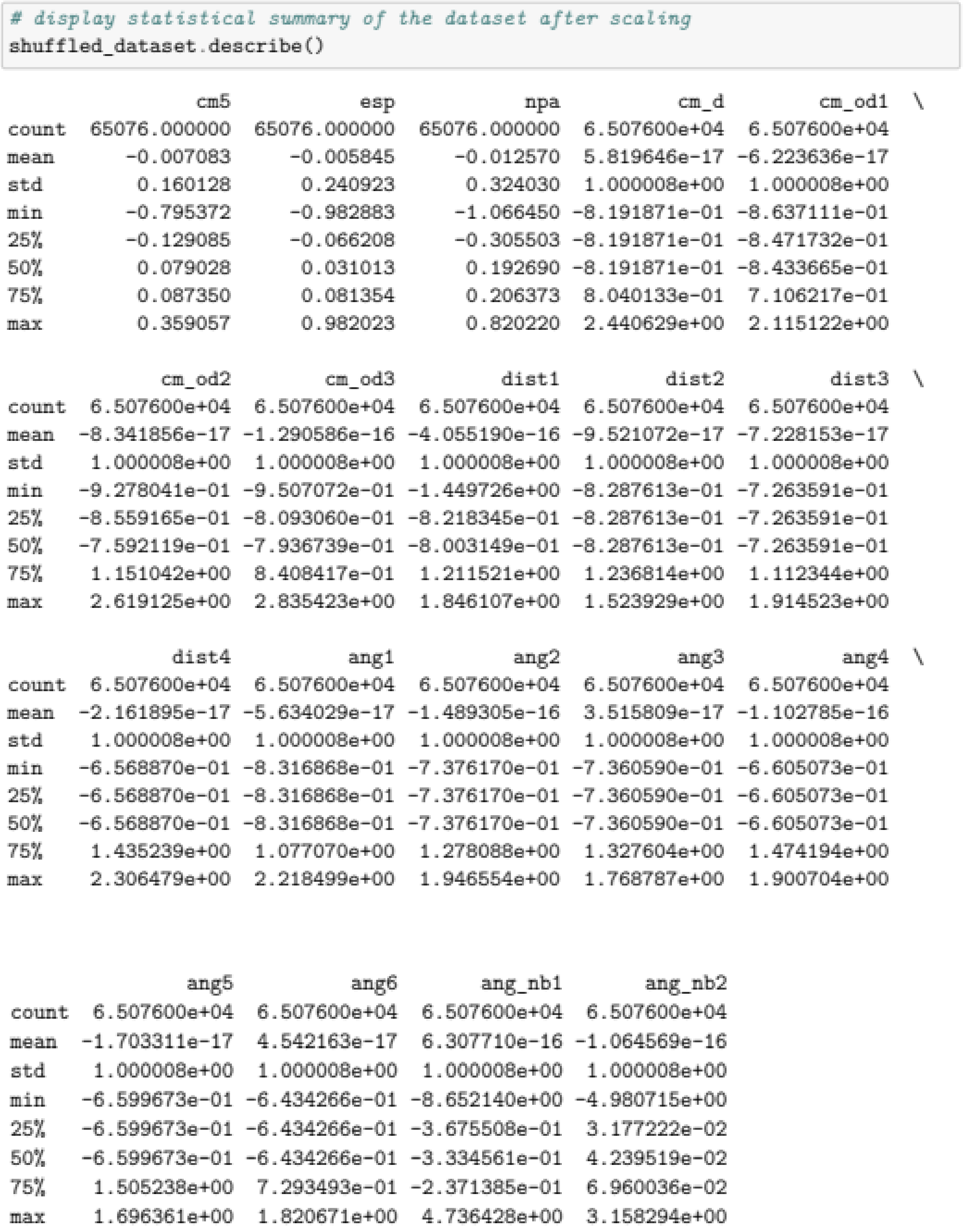
Now the whole dataset is shuffled, I proceed to use machine learning libraires such as scikit-learn to scale and split the dataset. For scaling and splitting, I used the code below to import the modules needed for this task:

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

“Feature scaling is the process of normalizing the features to make sure the machine learning model will process them in an effective way”. (What Is Feature Scaling and Why Is It Important, n.d.) These techniques can be helpful to improve model performance. I wrote this code for scaling the features:

```
# Create a standard scaler object
 scaler = StandardScaler()
# select the features and scale them
shuffled_dataset.iloc[:,4:]=scaler.fit_transform(shuffled_dataset.ilo
c[:,4:].values)
# statistical summary
 shuffled_dataset.describe()
```

The code above is used for the normalization process. I created a scaler object from the StandardScaler module, and I selected the features columns using indexing method then I called my scaler to make the transformation on the selected columns. To make sure that my features are scaled, I print a statistical summary of the dataset. See the summary in Figure 9 down below.



```
# display statistical summary of the dataset after scaling
shuffled_dataset.describe()
```

```
                cm5           esp           npa          cm_d        cm_od1  \
count  65076.000000  65076.000000  65076.000000  6.507600e+04  6.507600e+04
mean      -0.007083     -0.005845     -0.012570  5.819646e-17 -6.223636e-17
std        0.160128      0.240923      0.324030  1.000008e+00  1.000008e+00
min       -0.795372     -0.982883     -1.066450 -8.191871e-01 -8.637111e-01
25%       -0.129085     -0.066208     -0.305503 -8.191871e-01 -8.471732e-01
50%        0.079028      0.031013      0.192690 -8.191871e-01 -8.433665e-01
75%        0.087350      0.081354      0.206373  8.040133e-01  7.106217e-01
max        0.359057      0.982023      0.820220  2.440629e+00  2.115122e+00

             cm_od2        cm_od3         dist1         dist2         dist3  \
count  6.507600e+04  6.507600e+04  6.507600e+04  6.507600e+04  6.507600e+04
mean  -8.341856e-17 -1.290586e-16 -4.055190e-16 -9.521072e-17 -7.228153e-17
std    1.000008e+00  1.000008e+00  1.000008e+00  1.000008e+00  1.000008e+00
min   -9.278041e-01 -9.507072e-01 -1.449726e+00 -8.287613e-01 -7.263591e-01
25%   -8.559165e-01 -8.093060e-01 -8.218345e-01 -8.287613e-01 -7.263591e-01
50%   -7.592119e-01 -7.936739e-01 -8.003149e-01 -8.287613e-01 -7.263591e-01
75%    1.151042e+00  8.408417e-01  1.211521e+00  1.236814e+00  1.112344e+00
max    2.619125e+00  2.835423e+00  1.846107e+00  1.523929e+00  1.914523e+00

              dist4          ang1          ang2          ang3          ang4  \
count  6.507600e+04  6.507600e+04  6.507600e+04  6.507600e+04  6.507600e+04
mean  -2.161895e-17 -5.634029e-17 -1.489305e-16  3.515809e-17 -1.102785e-16
std    1.000008e+00  1.000008e+00  1.000008e+00  1.000008e+00  1.000008e+00
min   -6.568870e-01 -8.316868e-01 -7.376170e-01 -7.360590e-01 -6.605073e-01
25%   -6.568870e-01 -8.316868e-01 -7.376170e-01 -7.360590e-01 -6.605073e-01
50%   -6.568870e-01 -8.316868e-01 -7.376170e-01 -7.360590e-01 -6.605073e-01
75%    1.435239e+00  1.077070e+00  1.278088e+00  1.327604e+00  1.474194e+00
max    2.306479e+00  2.218499e+00  1.946554e+00  1.768787e+00  1.900704e+00


               ang5          ang6       ang_nb1       ang_nb2
count  6.507600e+04  6.507600e+04  6.507600e+04  6.507600e+04
mean  -1.703311e-17  4.542163e-17  6.307710e-16 -1.064569e-16
std    1.000008e+00  1.000008e+00  1.000008e+00  1.000008e+00
min   -6.599673e-01 -6.434266e-01 -8.652140e+00 -4.980715e+00
25%   -6.599673e-01 -6.434266e-01 -3.675508e-01  3.177222e-02
50%   -6.599673e-01 -6.434266e-01 -3.334561e-01  4.239519e-02
75%    1.505238e+00  7.293493e-01 -2.371385e-01  6.960036e-02
max    1.696361e+00  1.820671e+00  4.736428e+00  3.158294e+00
```

Figure 9: statistical summary after feature scaling.

The features are scaled, and the dataset needs to be split into training, validation, and testing datasets. Here I state the code I wrote for this with explanation for each part under the code:

```
features = shuffled_dataset.drop(['name', 'cm5', 'esp','npa'],axis=1)
```

By dropping the target columns and the 'name' column, the remaining columns now are assigned to a dataframe called 'features' and it contains only the 16 features.

```
labels = shuffled_dataset[['cm5', 'esp', 'npa']]
```

This selects only the columns 'cm5', 'esp', 'npa' from the dataset and assigns them to a dataframe I named 'labels' and it contains only the 3 targets.

```
x_train, x_temp, y_train, y_temp = train_test_split(features, labels,
train_size=0.80, random_state=42)
```

The code above shows the first step of splitting. This specifies that 80% of the original dataset will be allocated for the training set ('x_train' and 'y_train'), and the remaining 20% are basically temporary datasets ('x_temp' and 'y_temp') which will be used for further splitting.

```
x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp,
test_size=0.5, random_state=42)
```

Here, we take the temporary data ('x_temp' and 'y_temp') and split it into half. 50% will become the validation set ('x_val', 'y_val') and 50% will become the testing set ('x_test', 'y_test'). Each of these sets will represent 10% of the original dataset.

Overall, after all the splitting, we must now have:

80% of the original dataset used as the training data

10% of the original dataset used as the validation data

10% of the original dataset used as the testing data

I wrote this code below to verify the shape of each dataset.

```
print("Training Features Shape:", x_train.shape)
print("Training Labels Shape:", y_train.shape)
print("Validation Features Shape:", x_val.shape)
print("Validation Labels Shape:", y_val.shape)
print("Test Features Shape:", x_test.shape)
print("Test Labels Shape:", y_test.shape)
```

The output of the code above is this:

```
Training Features Shape: (52060, 16)
Training Labels Shape: (52060, 3)
Validation Features Shape: (6508, 16)
```

```
 Validation Labels Shape: (6508, 3)
 Test Features Shape: (6508, 16)
 Test Labels Shape: (6508, 3)
```

The shapes of the datasets prove that the splitting was done correctly and now we can move to the next step which is building the neural network.

3. Machine learning model development

“There are several machine learning categories such as supervised machine learning, unsupervised machine learning and reinforcement machine learning. Our focus would be on supervised machine learning because our data is labeled. This algorithm learns the complex pattern from the input to the output”.((PDF) Machine Learning: A Review of Learning Types, n.d.)

To examine the relationship between the 16 features and the 3 targets (partial charges), a multivariate regression analysis was conducted.

“Among the numerous algorithmic methods employed in supervised machine learning are neural networks.”.(What Is Supervised Learning? | IBM, n.d.) “They are at the core of deep learning, and they mimic the brain’s structure with layers of interconnected nodes. Nodes use inputs, weights, biases, and outputs to process data. Neural networks through supervised learning and gradient descent, adjust their calculations to minimize errors and increase accuracy”. (What Is Supervised Learning? | IBM, n.d.) The Figure 10 below represent a neural network architecture.

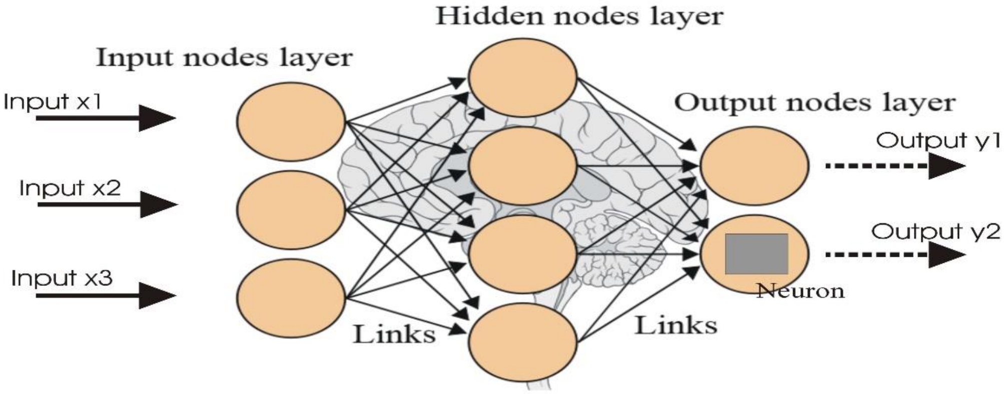


Figure 10: Neural Network

This section will discuss the machine learning methodology used and the architecture of the neural network model. The initial training and results subsection will cover the parameters I used for experimenting to gain in-depth observations, along with different analysis and results for discovering how the training went over the epochs.

The hyperparameters subsection will cover various approaches and methods I implemented to refine the model based on the initial results, and how I run the tuner with different hyperparameters using techniques such as early stopping to prevent overfitting during the search.

The method to train the neural network model with the best hyperparameters and the model check point technique I used to save the model will be discussed later in this section. Also, the final model loading and evaluating will be shown after with the results and analysis obtained.

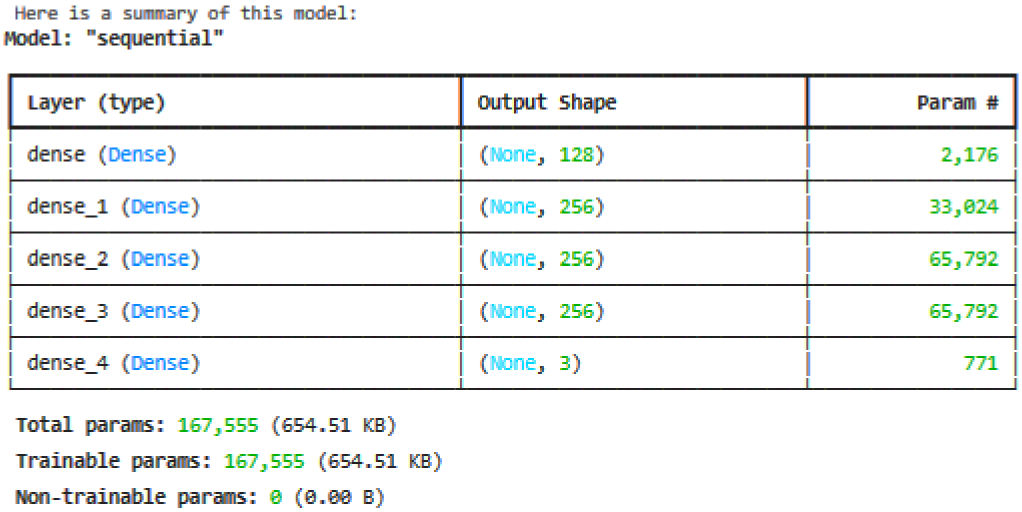
To build the model, I imported several libraries using the code below:

```
# The main machine learning framework used to build the neural network
import tensorflow as tf
from tensorflow import keras

#The components used for developing the model
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam

# Tensorflow untils packages for better documentation
import tensorflow_docs as tfdocs
import tensorflow_docs.plots
import tensorflow_docs.modeling
```

3.1 Neural network architecture

```
 # define the architecture of the neural network
 def build_neural_network_model():
    model = Sequential([
        # Input layer
        Input(shape=(x_train.shape[1], )),
        # hidden layer 1
        Dense(128,activation='relu',kernel_initializer='he_uniform'),
        # hidden Layer 2
        Dense(256,activation='relu',kernel_initializer='he_uniform'),
        # hidden Layer 3
        Dense(256,activation='relu',kernel_initializer='he_uniform'),
        # hidden layer 4
        Dense(256,activation='relu',kernel_initializer='he_uniform'),
        # output layer.
        Dense(3, activation='linear')
    ])
    optimizer = Adam(learning_rate=0.0001)
    model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae', 'mse'])
    return model
# displaying the model summary
model = build_neural_network_model()
print('Here is a summary of this model: ')
model.summary()
```

The code above was used to define the neural network architecture which contains of an input layer with 16 nodes according to the shape of the training dataset (that's the number of features inputs), 4 hidden layers each with 'relu' activation function, and an output layer with 3 nodes representing the targets with a linear activation function because the output value can be negative. The hidden layers 2, 3, and 4 contain 256 neurons each, while the first hidden layer has 128 neurons. Adam optimizer was employed with a 0.0001 learning rate. I used Mean Squared Error (MSE) and Mean Absolute Error (MAE) metrices to evaluate my model's accuracy. MSE focuses on larger errors, while MAE gives a general picture of error size. Using both metrics helped me understand how my model was learning. Lastly, I displayed a summary of the model to check the total parameters. The Figure 11 represents the summary after compiling the model.



```
Here is a summary of this model:
Model: "sequential"

┌─────────────────────────────┬──────────────────────┬─────────────┐
│ Layer (type)                │ Output Shape         │     Param # │
├─────────────────────────────┼──────────────────────┼─────────────┤
│ dense (Dense)               │ (None, 128)          │       2,176 │
├─────────────────────────────┼──────────────────────┼─────────────┤
│ dense_1 (Dense)             │ (None, 256)          │      33,024 │
├─────────────────────────────┼──────────────────────┼─────────────┤
│ dense_2 (Dense)             │ (None, 256)          │      65,792 │
├─────────────────────────────┼──────────────────────┼─────────────┤
│ dense_3 (Dense)             │ (None, 256)          │      65,792 │
├─────────────────────────────┼──────────────────────┼─────────────┤
│ dense_4 (Dense)             │ (None, 3)            │         771 │
└─────────────────────────────┴──────────────────────┴─────────────┘

 Total params: 167,555 (654.51 KB)
 Trainable params: 167,555 (654.51 KB)
 Non-trainable params: 0 (0.00 B)
```

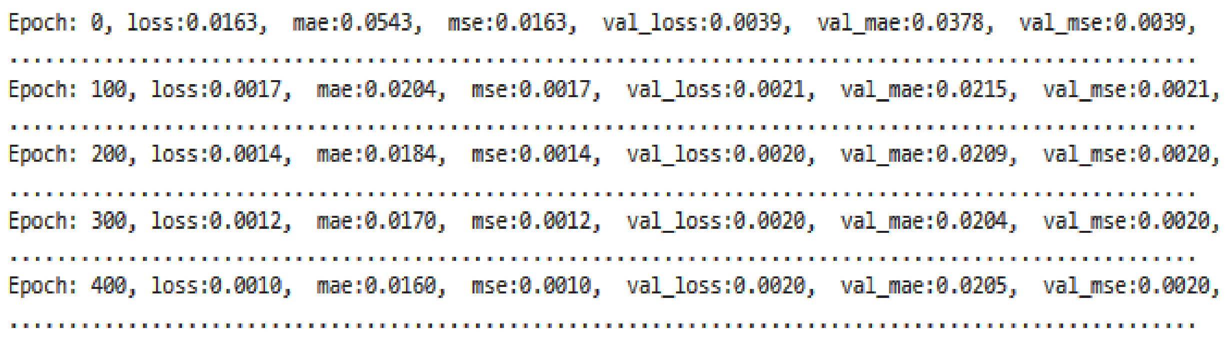
Figure 11: Summary of the model

3. 2 Model training, initial evaluation and results

I compiled the model and configured training for 500 epochs with a batch size of 34. “This batch size promotes faster, computationally efficient training”. (Differences Between Epoch, Batch, and Mini-Batch | Baeldung on Computer Science, n.d.) Training utilized the 'x_train' and 'y_train' datasets, while 'x_val' and 'y_val' served for validation. The training code is as follows:

```
EPOCHS = 500
batch_size = 34
history = model.fit(
        x_train,
        y_train,
        batch_size = batch_size,
        epochs=EPOCHS,
        verbose=0,
        steps_per_epoch = x_train.shape[0] // batch_size,
        validation_data = (x_val, y_val),
        callbacks=[tfdocs.modeling.EpochDots()])
```

Using 'tfdocs', I formatted the training results for clarity, displaying progress updates every 100 epochs. See Figure 12 below for a visual example:



```
Epoch: 0, loss:0.0163,  mae:0.0543,  mse:0.0163,  val_loss:0.0039,  val_mae:0.0378,  val_mse:0.0039,
....................................................................................................
Epoch: 100, loss:0.0017,  mae:0.0204,  mse:0.0017,  val_loss:0.0021,  val_mae:0.0215,  val_mse:0.0021,
....................................................................................................
Epoch: 200, loss:0.0014,  mae:0.0184,  mse:0.0014,  val_loss:0.0020,  val_mae:0.0209,  val_mse:0.0020,
....................................................................................................
Epoch: 300, loss:0.0012,  mae:0.0170,  mse:0.0012,  val_loss:0.0020,  val_mae:0.0204,  val_mse:0.0020,
....................................................................................................
Epoch: 400, loss:0.0010,  mae:0.0160,  mse:0.0010,  val_loss:0.0020,  val_mae:0.0205,  val_mse:0.0020,
....................................................................................................
```

Figure 12: Training results over 500 epochs

The training process demonstrated promising results. Both training loss and validation loss decreased steadily over the epochs, indicating that the model successfully learned from the data without significant overfitting. This suggests the model's potential to generalize well to unseen examples. For better analysis, with the help of ‘tfdocs’, I plotted the Mean Absolute Error over the 500 epochs to discover more details, the code below was used for that:

```
plotter = tfdocs.plots.HistoryPlotter(smoothing_std=2)
plotter.plot({'Basic': history}, metric = "mae")
plt.ylim([0, 0.05])
plt.ylabel('MAE')
```

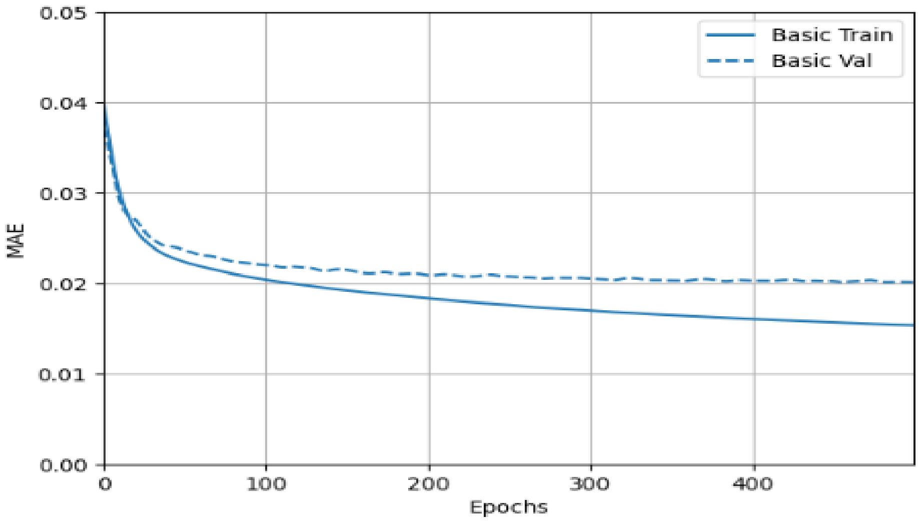


Figure 13: MAE over 500 epochs

According to Figure 13 above, It appears that both the training MAE (Basic_mae) and validation MAE (val_mae) decrease over the epochs, which suggests that the model is learning and improving on both the training and validation data.

For more deeper insights I plotted the MSE using the code below:

```
plotter.plot({'Basic': history}, metric = "mse")
plt.ylim([0, 0.006])
plt.ylabel('MSE')
```

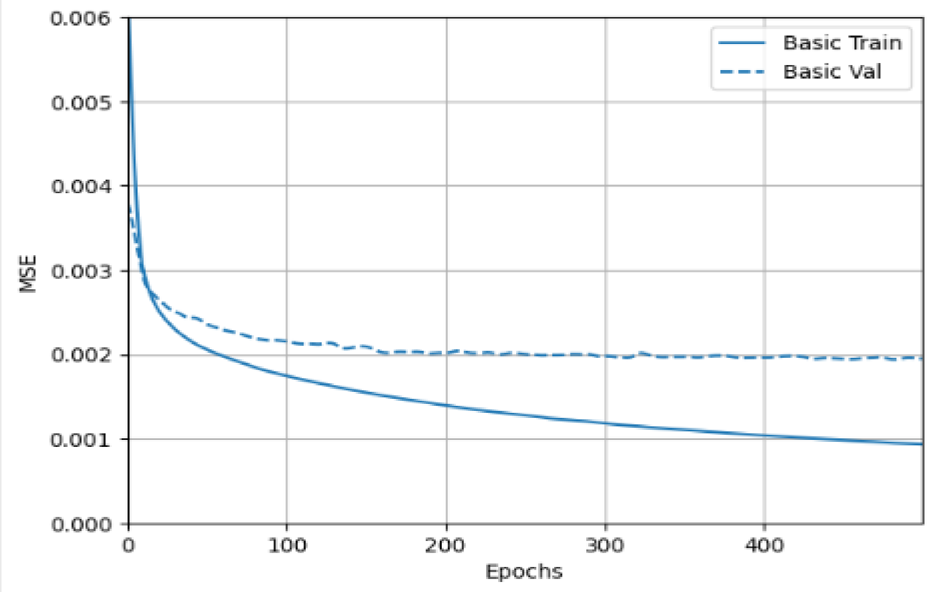


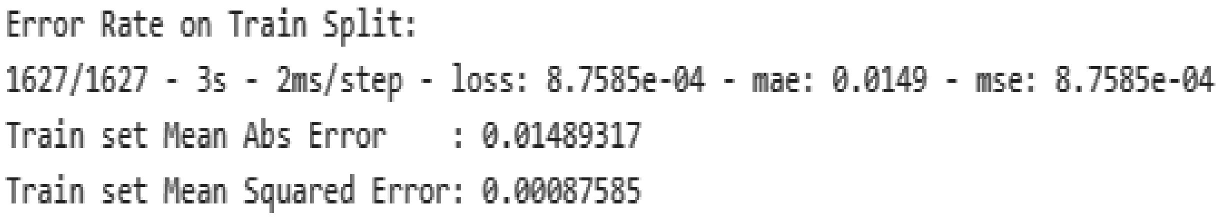
Figure 14: MSE over 500 epochs

The Figure 14 above shows that the model's MSE is decreasing on both the training and validation data in general, suggesting it is generalizing well, but we can also notice that the MSE of the validation data does not improve after 200 epochs unlike the MSE of the training data which keeps improving over the 500 epochs, suggesting that the model could be memorizing the pattern instead of learning it.

Down below is the code I used for evaluating the errors on training set:

```
print('Error Rate on Train Split: ')
loss, mae, mse = model.evaluate(x_train, y_train, verbose=2)
print("Train set Mean Abs Error     : {:.8f}".format(mae))
print("Train set Mean Squared Error: {:.8f}".format(mse))
```

The results of the code above are mentioned in Figure 15 down:



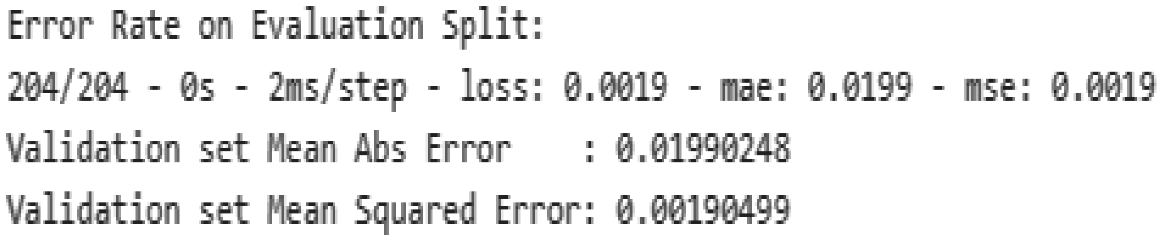
```
Error Rate on Train Split:
1627/1627 - 3s - 2ms/step - loss: 8.7585e-04 - mae: 0.0149 - mse: 8.7585e-04
Train set Mean Abs Error    : 0.01489317
Train set Mean Squared Error: 0.00087585
```

Figure 15: Error rate on training set

Down below is the code I used for evaluating the errors on validation set:

```
print('Error Rate on Evaluation Split: ')
loss, mae, mse = model.evaluate(x_val, y_val, verbose=2)
print("Validation set Mean Abs Error    : {:.8f}".format(mae))
print("Validation set Mean Squared Error: {:.8f}".format(mse))
```

The results of the code above are shown in the Figure 16 below:



```
Error Rate on Evaluation Split:
204/204 - 0s - 2ms/step - loss: 0.0019 - mae: 0.0199 - mse: 0.0019
Validation set Mean Abs Error    : 0.01990248
Validation set Mean Squared Error: 0.00190499
```

Figure 16: Error rate on validation set

The model demonstrated excellent performance on the training dataset, achieving a low error rate with a Mean Absolute Error (MAE) of 0.0149 and a Mean Squared Error (MSE) of 0.000875. Furthermore, the evaluation on a separate validation set confirmed the model's ability to generalize well to unseen data. It showed a slightly increased MAE of 0.0199 and an MSE of 0.0019 on the validation set, suggesting that the model might be memorizing specific patterns in the training data instead of learning generalizable representations. This indicated potential room for improvement through hyperparameters tuning.

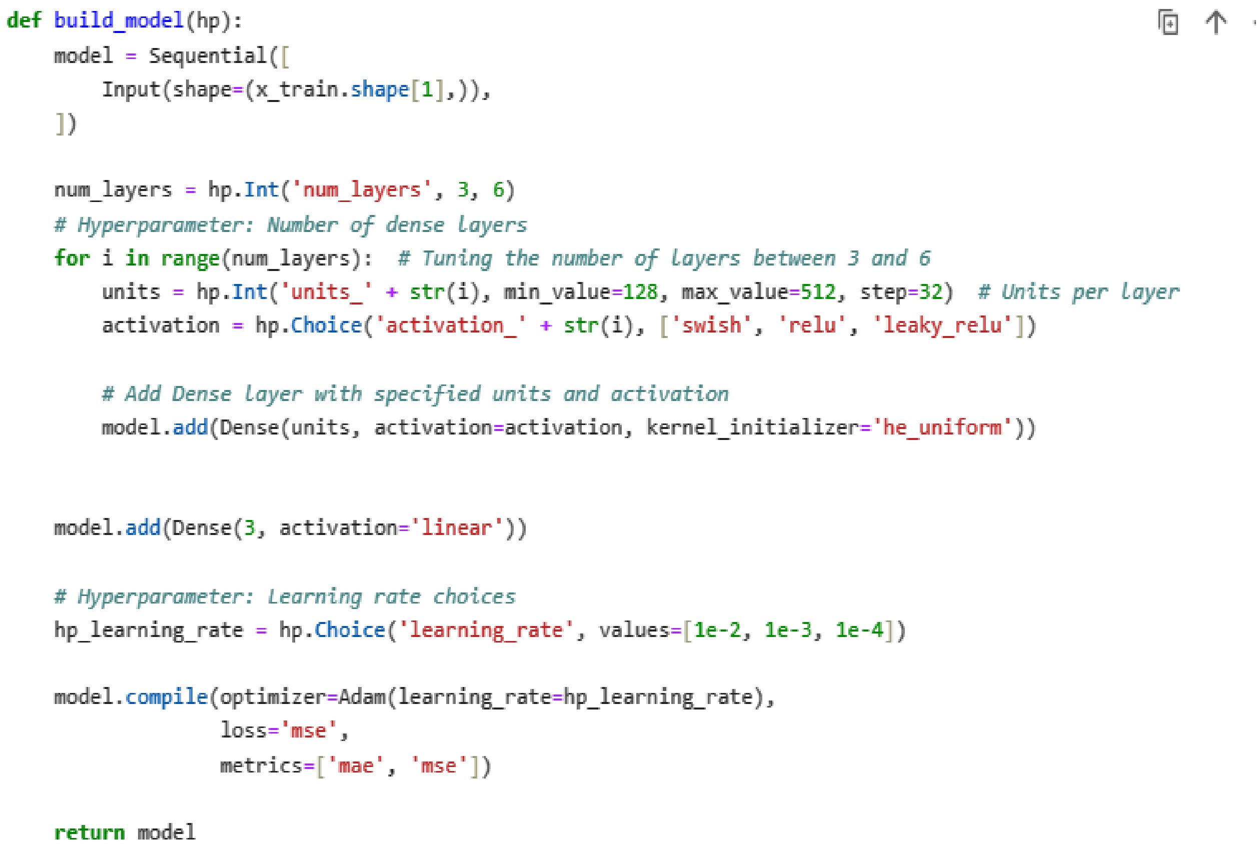
4. Hyperparameters tuning

The first thing I did was import the ‘BayesianOptimization’ from ‘Keras_tuner’ and ‘EarlyStopping’ from TensorFlow. With the combination of these 2 techniques, I was able to efficiently explore different hyperparameters for the model. “BayesianOptimization is nothing but a method to find the best combination of hyperparameters”. (Frazier, 2018) The code was used for this task:

```
from keras_tuner import BayesianOptimization
from tensorflow.keras.callbacks import EarlyStopping
```

4.1 Model with various parameters

Based on the previous model and its evaluation result, I defined a neural network architecture where several critical hyperparameters can be optimized with the Bayesian method. The design provides a way to explore different model structures and tune values within specific ranges, aiming to find the best configurations. The code used to define the model is shown in the Figure 17 underneath for better representation:



```
def build_model(hp):
    model = Sequential([
        Input(shape=(x_train.shape[1],)),
    ])

    num_layers = hp.Int('num_layers', 3, 6)
    # Hyperparameter: Number of dense Layers
    for i in range(num_layers):  # Tuning the number of Layers between 3 and 6
        units = hp.Int('units_' + str(i), min_value=128, max_value=512, step=32)  # Units per Layer
        activation = hp.Choice('activation_' + str(i), ['swish', 'relu', 'leaky_relu'])

        # Add Dense Layer with specified units and activation
        model.add(Dense(units, activation=activation, kernel_initializer='he_uniform'))


    model.add(Dense(3, activation='linear'))

    # Hyperparameter: Learning rate choices
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(optimizer=Adam(learning_rate=hp_learning_rate),
                  loss='mse',
                  metrics=['mae', 'mse'])

    return model
```

Figure 17: Neural Network with different hyperparameters

The components of this model are as follows:

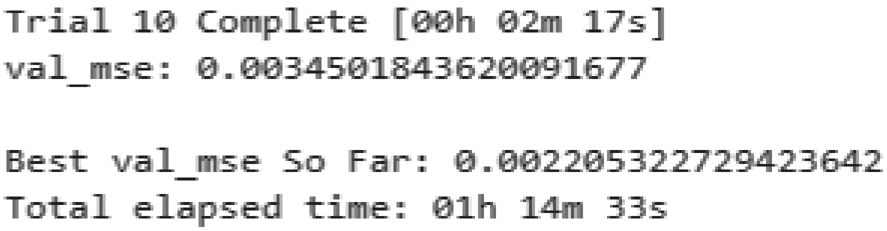
- Sequential model is the core of the model which indicates that layers are added one after another
- Input layer defines the input shape of the data, which is 16 in our case
- num_layers: this hyperparameter controls the number of hidden dense layers in the model (between 3 and 6).
- units: for each layer, this hyperparameter determines the number of neurons/units in that layer (values from 128 to 512 in steps of 32).
- activation: each layer's activation function can be selected from 'swish', 'relu', or 'leaky_relu'.
- output layer: the final dense layer which has 3 output neurons and uses 'linear' activation.
- The learning rate for the Adam optimizer is also a tunable hyperparameter with possible values of 1e-2, 1e-3, and 1e-4 (0.01, 0.001, and 0.0001).

```
tuner = BayesianOptimization(
    build_model,
    objective='val_mse',
    num_initial_points=30,
    max_trials=10,
    directory='directory_dnn_building_model_on_tuner',
    project_name='hyperparameters_tuning_dnn')
```

The code above was used to guide the search process. The tuner was configured to minimize the validation Mean Squared Error ('val_mse') over a maximum of 10 trials. Initial exploration included 30 random hyperparameter configurations. Results and models were saved within a dedicated directory ('directory_dnn_building_model_on_tuner') for further analysis and use.

```
early_stopping_callback=EarlyStopping(monitor='val_mse',patience=10)

tuner.search(x_train,
             y_train,
             epochs=500,
             validation_data=(x_val, y_val),
                     callbacks=[early_stopping_callback,
                     tfdocs.modeling.EpochDots()])
```

The code above was used to run the search. I employed ‘EarlyStopping’ with a patience of 10 epochs to avoid overfitting. This technique halts training if validation Mean Squared Error fails to improve for 10 consecutive epochs, preventing the model from memorizing training data and losing generalizability. The ‘EarlyStopping’ callback, coupled with ‘BayesianOptimization’, allowed for an efficient hyperparameter search. The Bayesian Optimization completed its 10 trials, with the lowest validation MSE (0.0022) achieved in an earlier trial. This suggests that the tuner successfully explored the hyperparameter space within a total elapsed time of 1 hour and 14 min and 33 seconds. See the Figure 18 below:



```
Trial 10 Complete [00h 02m 17s]
val_mse: 0.003450184362009167 7

Best val_mse So Far: 0.002205322729423642
Total elapsed time: 01h 14m 33s
```

Figure 18: Bayesian Optimization output result

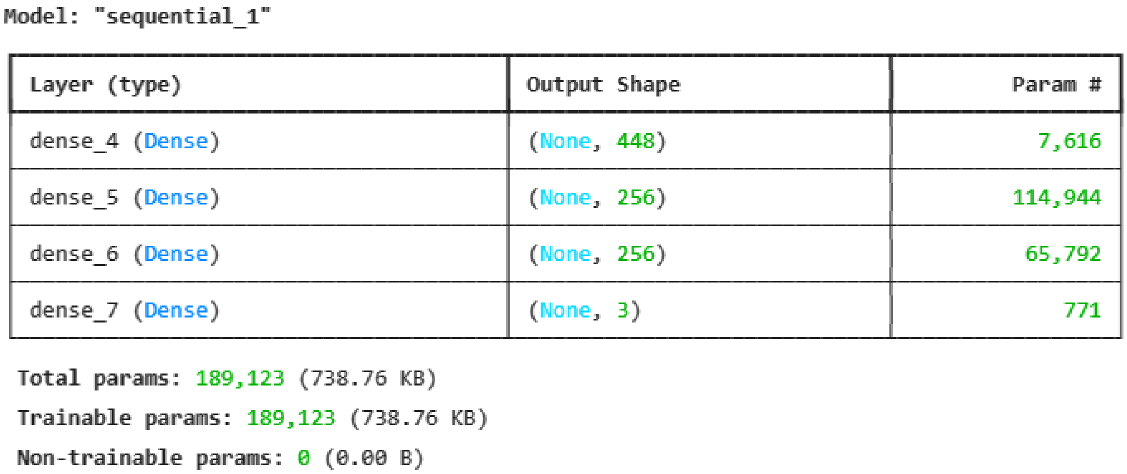
4.2 Training the model with the best hyperparameters and saving the model

After the search was done, I extracted the best hyperparameters and saved them in a variable called ‘best_hps’, then I re-built the model with these preferred configurations. Underneath is the code I used to do these steps:

```
 # Extract the best hyperparameters
 best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
 # Build the model with the best hyperparameters
 best_hps_model = tuner.hypermodel.build(best_hps)
```

I displayed the summary of the new built model before I started training the model, I used this code: `print(best_hps_model.summary())`

The output of the summary is shown under in the Figure 19:



```
Model: "sequential_1"
┌─────────────────────────────────┬──────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape             │       Param # │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_4 (Dense)                 │ (None, 448)              │         7,616 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_5 (Dense)                 │ (None, 256)              │       114,944 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_6 (Dense)                 │ (None, 256)              │        65,792 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_7 (Dense)                 │ (None, 3)                │           771 │
└─────────────────────────────────┴──────────────────────────┴───────────────┘
 Total params: 189,123 (738.76 KB)
 Trainable params: 189,123 (738.76 KB)
 Non-trainable params: 0 (0.00 B)
```

Figure 19: Summary of the model with best hyperparameters

Before I trained the model with the best hyperparameters, I set up a ModelCheckPoint callback to save the best model during the training. I wrote the code below to define the function:

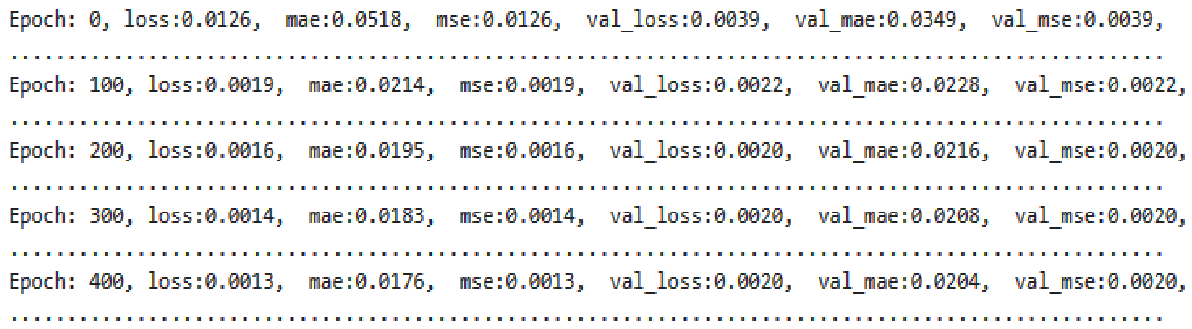
```
# file path for saving the best model
checkpoint_path = "./models/best_model.keras"
# ModelCheckpoint callback to save the best model
ckpt_callback = ModelCheckpoint(
    filepath=checkpoint_path,
    monitor='val_mse',
    save_best_only=True,  # Save only the best model
    save_weights_only=False,  # Save the entire model
    verbose=0)
```

'checkpoint_path' variable defines the file path where the best model will be saved. 'ckpt_callback' initializes a ModelCheckPoint callback object to save the best model during training. 'monitor' determines the metric to monitor for deciding the best model, and it is set to monitor the mean squared error on the validation dataset. 'save_best_only' is set to true to save the best model only based on the monitored metric, and if a new best model is found, it will overwrite the previously saved one. 'save_weight_only' is set to false to save the entire model, including its architecture and weights.

I proceed to training this model by using the code below:

```
history = best_hps_model.fit(
    x_train,
    y_train,
    batch_size=34,
    epochs=500,
    verbose=0,
    validation_data=(x_val, y_val),
    callbacks=[ckpt_callback, tfdocs.modeling.EpochDots()])
```

The results of the training over the 500 epochs are mentioned below in Figure 20:



```
Epoch: 0, loss:0.0126,  mae:0.0518,  mse:0.0126,  val_loss:0.0039,  val_mae:0.0349,  val_mse:0.0039,
....................................................................................................
Epoch: 100, loss:0.0019,  mae:0.0214,  mse:0.0019,  val_loss:0.0022,  val_mae:0.0228,  val_mse:0.0022,
....................................................................................................
Epoch: 200, loss:0.0016,  mae:0.0195,  mse:0.0016,  val_loss:0.0020,  val_mae:0.0216,  val_mse:0.0020,
....................................................................................................
Epoch: 300, loss:0.0014,  mae:0.0183,  mse:0.0014,  val_loss:0.0020,  val_mae:0.0208,  val_mse:0.0020,
....................................................................................................
Epoch: 400, loss:0.0013,  mae:0.0176,  mse:0.0013,  val_loss:0.0020,  val_mae:0.0204,  val_mse:0.0020,
....................................................................................................
```

Figure 20: Results of training the model with best hyperparameters

These results indicate a consistent improvement in both training and validation metrics over the epochs, which is a positive sign the neural network model is performing well and is effectively learning from the data.

Plotting the metrics errors would give us more insights, using the code below I started with Mean Absolut Error on both training and validation sets.

```
plotter.plot({'Basic': history}, metric = "mae")
plt.ylim([0, 0.05])
plt.ylabel('MAE')
```

To visualize the plot, see the Figure 21 under:

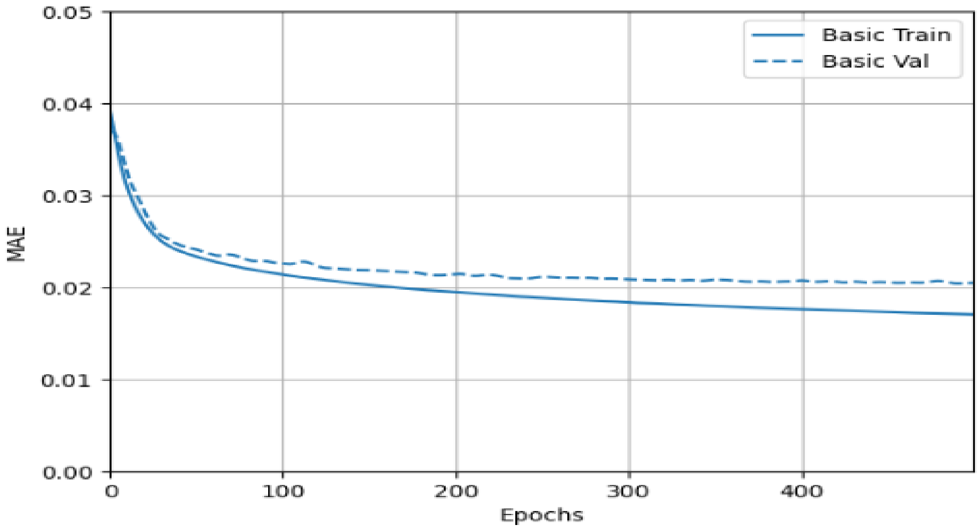


Figure 21: Best model MAE error over 500

The graph represents the performance on the training data and the validation data. Their downward trends suggest that the model is learning and generalizing well.

The code below plots Mean Squared Error:

```
plotter.plot({'Basic': history}, metric = "mse")
plt.ylim([0, 0.006])
plt.ylabel('MSE')
```

See the Figure 22 which represents the plot of Mean Square Error:

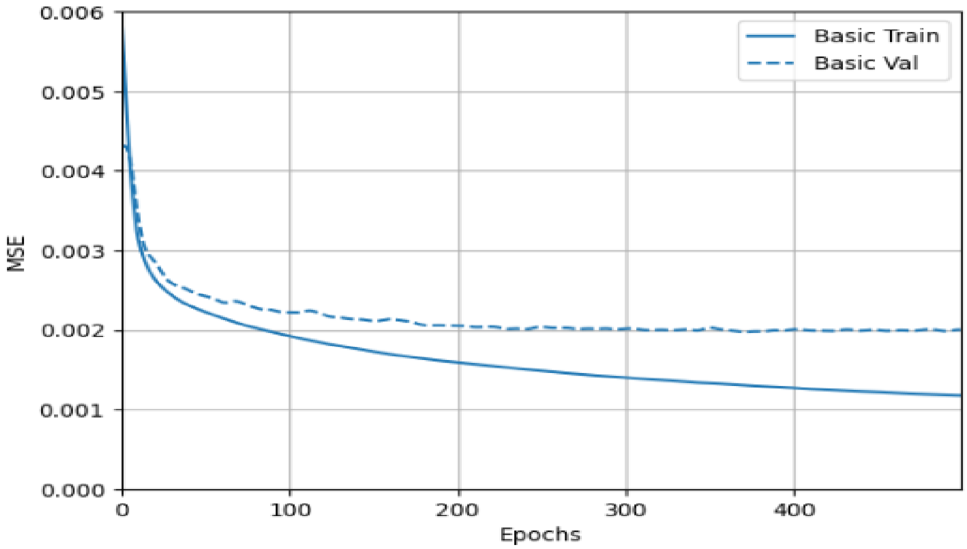


Figure 22: Best Model MSE over 500 epochs

This graph shows that the basic train is slightly higher than the basic validation, which opens door to question if the model may be overfitting the training data or it will actually do good on unssen data.

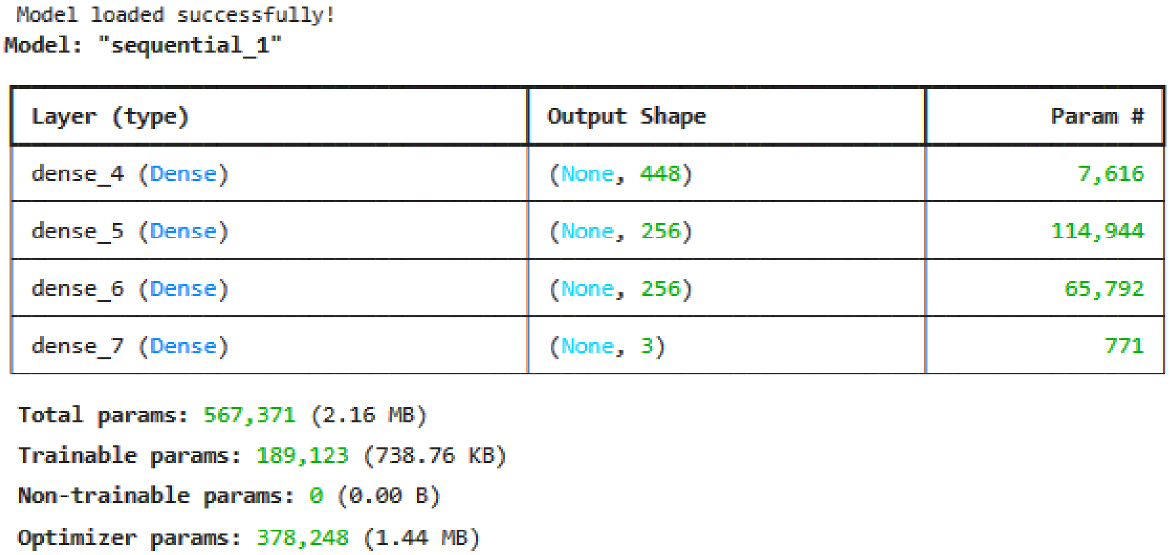
5. Final model loading and evaluating

This chaprter covers the steps of loading the final model that was previously saved and its evaluation resutls on training, validation datasets and also testing dataset this time to see how it generalizes on unseen data.

5.1 Final best model loading

```
# Load the saved model
try:
    # Load the  complete model
    final_model = load_model(checkpoint_path)
    print("Model loaded successfully!")
    final_model.summary()  # Display the model's summary
except Exception as e:
    print(f"Error loading model: {e}")
```

The code above loads the final model I saved during the training, if the loading was done successfully, the summary will be displayed otherwise the exception error will be displayed. In this case, the summary was displayed, see Figure 23 down:



```
Model loaded successfully!
Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━┓
┃ Layer (type)                  ┃ Output Shape           ┃      Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━┩
│ dense_4 (Dense)               │ (None, 448)            │        7,616 │
├───────────────────────────────┼────────────────────────┼──────────────┤
│ dense_5 (Dense)               │ (None, 256)            │      114,944 │
├───────────────────────────────┼────────────────────────┼──────────────┤
│ dense_6 (Dense)               │ (None, 256)            │       65,792 │
├───────────────────────────────┼────────────────────────┼──────────────┤
│ dense_7 (Dense)               │ (None, 3)              │          771 │
└───────────────────────────────┴────────────────────────┴──────────────┘

 Total params: 567,371 (2.16 MB)
 Trainable params: 189,123 (738.76 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 378,248 (1.44 MB)
```

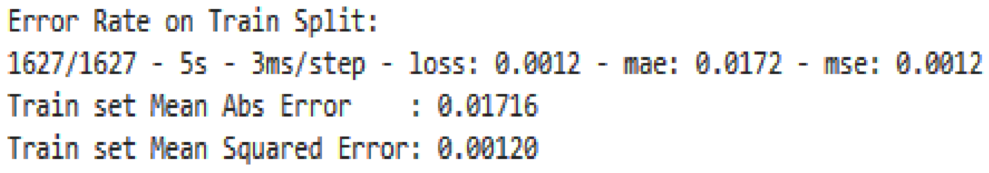
Figure 23: Final model summary

5.2 Final model evaluating

For evaluating the final model on training, validation, and testing sets, I used the codes below:

```
print('Error Rate on Train Split: ')
loss, mae, mse = final_model.evaluate(x_train,y_train,verbose=2)
print("Train set Mean Abs Error     : {:.5f}".format(mae))
print("Train set Mean Squared Error: {:.5f}".format(mse))
```

The error rate on training set is shown in Figure 24 down:



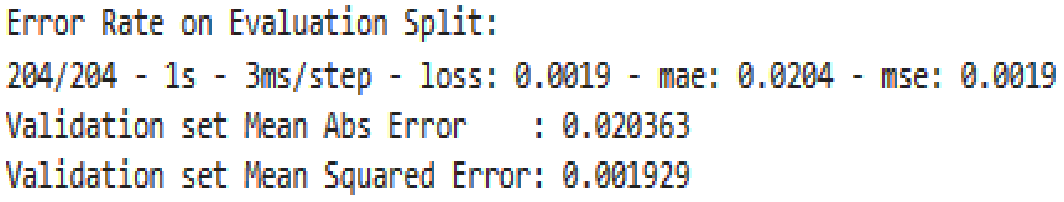
```
Error Rate on Train Split:
1627/1627 - 5s - 3ms/step - loss: 0.0012 - mae: 0.0172 - mse: 0.0012
Train set Mean Abs Error    : 0.01716
Train set Mean Squared Error: 0.00120
```

Figure 24: Error rate on training set

The code below to print the error rate on validation set

```
print('Error Rate on Evaluation Split: ')
loss, mae, mse = final_model.evaluate(x_val, y_val, verbose=2)
print("Validation set Mean Abs Error     : {:.6f}".format(mae))
print("Validation set Mean Squared Error: {:.6f}".format(mse))
```

The error rate on validation set in shown down in the Figure 25:



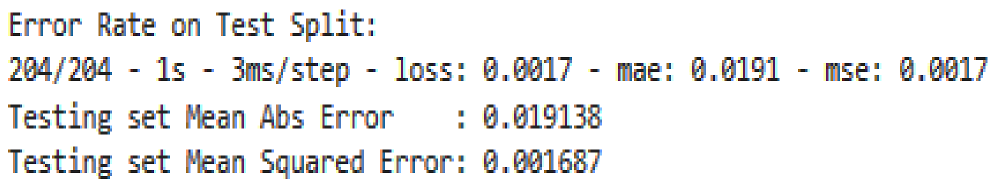
```
Error Rate on Evaluation Split:
204/204 - 1s - 3ms/step - loss: 0.0019 - mae: 0.0204 - mse: 0.0019
Validation set Mean Abs Error    : 0.020363
Validation set Mean Squared Error: 0.001929
```

Figure 25: Error rate on validation set

The code below to print the error rate on testing set

```
print('Error Rate on Test Split: ')
loss, mae, mse = final_model.evaluate(x_test, y_test, verbose=2)
print("Testing set Mean Abs Error     : {:.6f}".format(mae))
print("Testing set Mean Squared Error: {:.6f}".format(mse))
```

The error rate on validation set in shown down in the Figure 26:



```
Error Rate on Test Split:
204/204 - 1s - 3ms/step - loss: 0.0017 - mae: 0.0191 - mse: 0.0017
Testing set Mean Abs Error    : 0.019138
Testing set Mean Squared Error: 0.001687
```

Figure 26: Error rate on testing set

V. Results

Down I list the results from my final model:

- Mean Absolute Error (MAE): Across the training, validation, and testing sets, the MAE values were pretty low. This means the model's predictions were generally close to the real partial charge values.
- Mean Squared Error (MSE): Just like the MAE, the MSE values across all the datasets were also low. This is another good sign that the model is doing a solid job.
- Comparing sets: The model performed slightly better on the training and testing sets compared to the validation set, for both MAE and MSE. This tells us that the model is not just memorizing the training data but can generalize well to unseen data (testing set) too. This indicates that it's not overfitting.

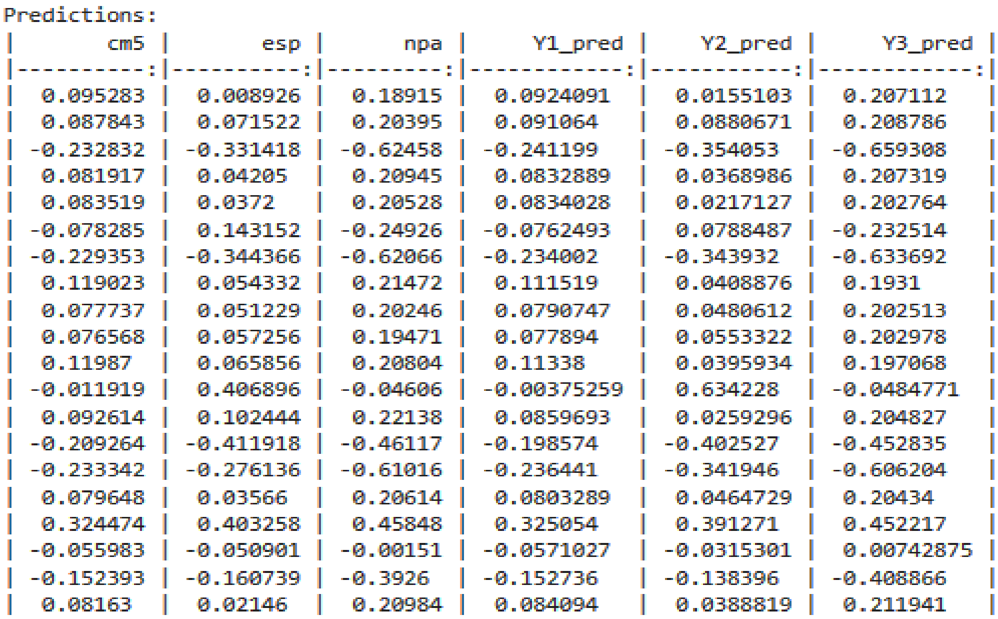
1. Predictions and visualization analysis

The code below is designed to display both the actual and predicted values for a subset of samples (20 sample) from the testing dataset

```
def print_predictions(y_true, y_pred, count):

    # DataFrame to display the actual and predicted values
    prediction_df = y_true.copy()
    # Assign predicted values to new columns
    prediction_df['Y1_pred'] = y_pred[:, 0]
    prediction_df['Y2_pred'] = y_pred[:, 1]
    prediction_df['Y3_pred'] = y_pred[:, 2]
    # Display the predictions
    print("Predictions:")
    print(prediction_df.head(count).to_markdown(index=False))
```

```
predictions = final_model.predict(x_test)
print()
print_predictions(y_test,predictions, count=20)
```

The 20 sample predictions are shown in the Figure 27 below:



```
Predictions:
|       cm5 |       esp |      npa |     Y1_pred |    Y2_pred |     Y3_pred |
|----------:|----------:|---------:|------------:|-----------:|------------:|
|  0.095283 |  0.008926 |  0.18915 |  0.0924091  |  0.0155103 |  0.207112   |
|  0.087843 |  0.071522 |  0.20395 |  0.091064   |  0.0880671 |  0.208786   |
| -0.232832 | -0.331418 | -0.62458 | -0.241199   | -0.354053  | -0.659308   |
|  0.081917 |  0.04205  |  0.20945 |  0.0832889  |  0.0368986 |  0.207319   |
|  0.083519 |  0.0372   |  0.20528 |  0.0834028  |  0.0217127 |  0.202764   |
| -0.078285 |  0.143152 | -0.24926 | -0.0762493  |  0.0788487 | -0.232514   |
| -0.229353 | -0.344366 | -0.62066 | -0.234002   | -0.343932  | -0.633692   |
|  0.119023 |  0.054332 |  0.21472 |  0.111519   |  0.0408876 |  0.1931     |
|  0.077737 |  0.051229 |  0.20246 |  0.0790747  |  0.0480612 |  0.202513   |
|  0.076568 |  0.057256 |  0.19471 |  0.077894   |  0.0553322 |  0.202978   |
|  0.11987  |  0.065856 |  0.20804 |  0.11338    |  0.0395934 |  0.197068   |
| -0.011919 |  0.406896 | -0.04606 | -0.00375259 |  0.634228  | -0.0484771  |
|  0.092614 |  0.102444 |  0.22138 |  0.0859693  |  0.0259296 |  0.204827   |
| -0.209264 | -0.411918 | -0.46117 | -0.198574   | -0.402527  | -0.452835   |
| -0.233342 | -0.276136 | -0.61016 | -0.236441   | -0.341946  | -0.606204   |
|  0.079648 |  0.03566  |  0.20614 |  0.0803289  |  0.0464729 |  0.20434    |
|  0.324474 |  0.403258 |  0.45848 |  0.325054   |  0.391271  |  0.452217   |
| -0.055983 | -0.050901 | -0.00151 | -0.0571027  | -0.0315301 |  0.00742875 |
| -0.152393 | -0.160739 | -0.3926  | -0.152736   | -0.138396  | -0.408866   |
|  0.08163  |  0.02146  |  0.20984 |  0.084094   |  0.0388819 |  0.211941   |
```

Figure 27: Actual values vs predicted values

The code below I used to create scatter plots that provide a visual representation of how closely the model's predictions align with the actual values for each target variable:

```
if not isinstance(y_test, pd.DataFrame):
    y_test = pd.DataFrame(y_test, columns=['Y1', 'Y2', 'Y3'])
# Iterate over each target variable and create plots
for i, column in enumerate(y_test.columns):
    plt.figure(figsize=(8, 8))
    true_values = y_test[column]
    predicted_values = predictions[:, i]
    plt.scatter(true_values,predicted_values,s=1,color='green')
    plt.title(f"Model Predictions vs True Values for {column}")
    plt.xlabel('True Values')
    plt.ylabel('Predicted Values')
    plt.xlim([min(true_values.min(), predicted_values.min()),
max(true_values.max(), predicted_values.max())])
    plt.ylim([min(true_values.min(), predicted_values.min()),
max(true_values.max(), predicted_values.max())])
    plt.plot([true_values.min(), true_values.max()],
[true_values.min(), true_values.max()], 'r', lw=2)
    plt.grid(True)
    plt.show()
```

Down below in the Figure 28 represents the plot of model predictions vs true values for ‘cm5’ charge:

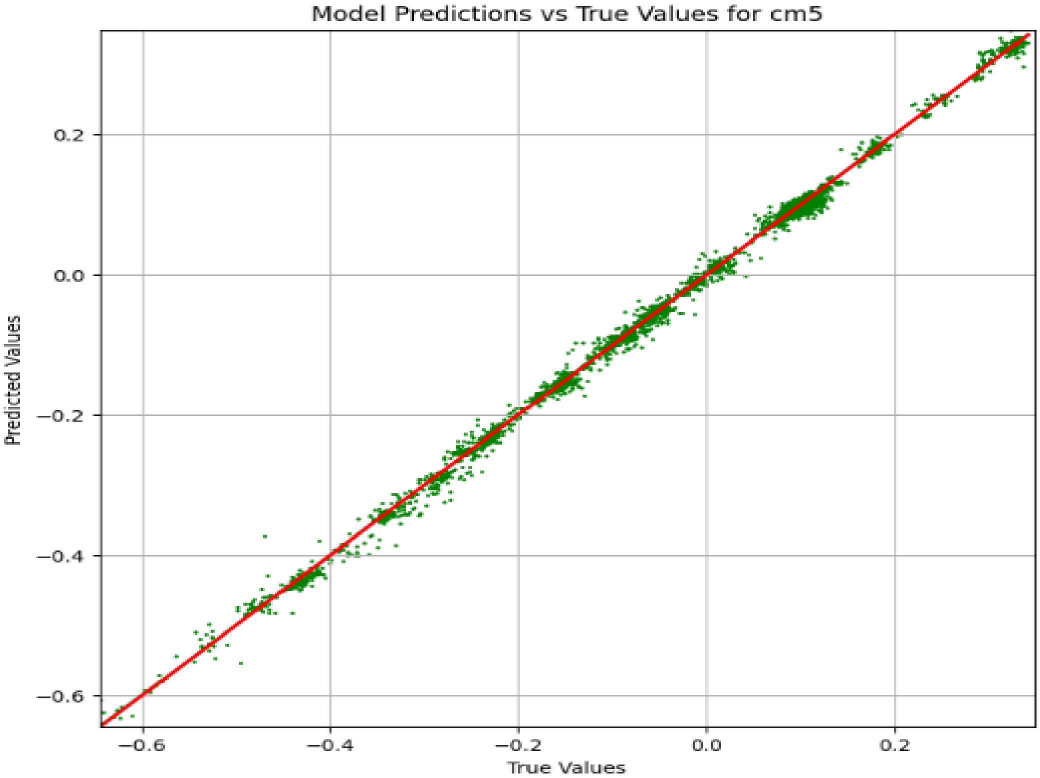


Figure 28: model prediction vs true value for cm5

The scatter plot displays a comparison between predicted values of ‘cm5’ charge and the actual true values from the final model, indicating a strong linear relationship, which suggests that the model's predictions and the actual values agree quite well.

Down below in the Figure 29 represents the plot of model predictions vs true values for’ esp’ charge:

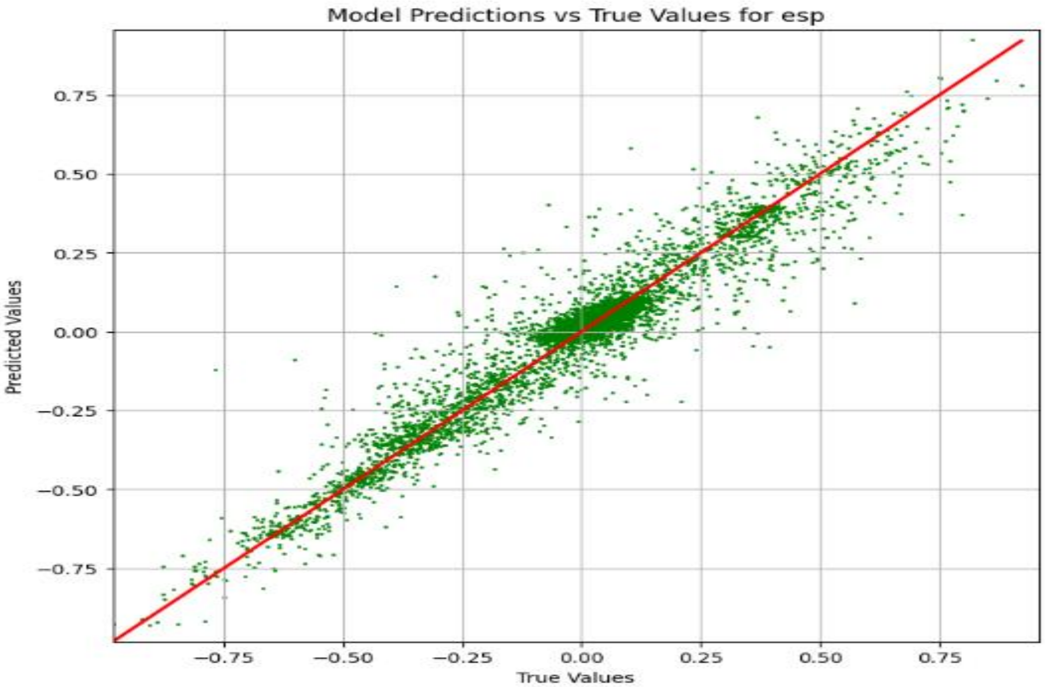


Figure 29: model predictions vs true values for esp

This plot represents the relationship between the predicted and true values of charge ‘esp’. Generally, we observe that the data points are generally clustered around the red line, indicating a good model fit, although there is some scatter.

Down below in the Figure 30 represents the plot of model predictions vs true values for ‘npa’ charge:

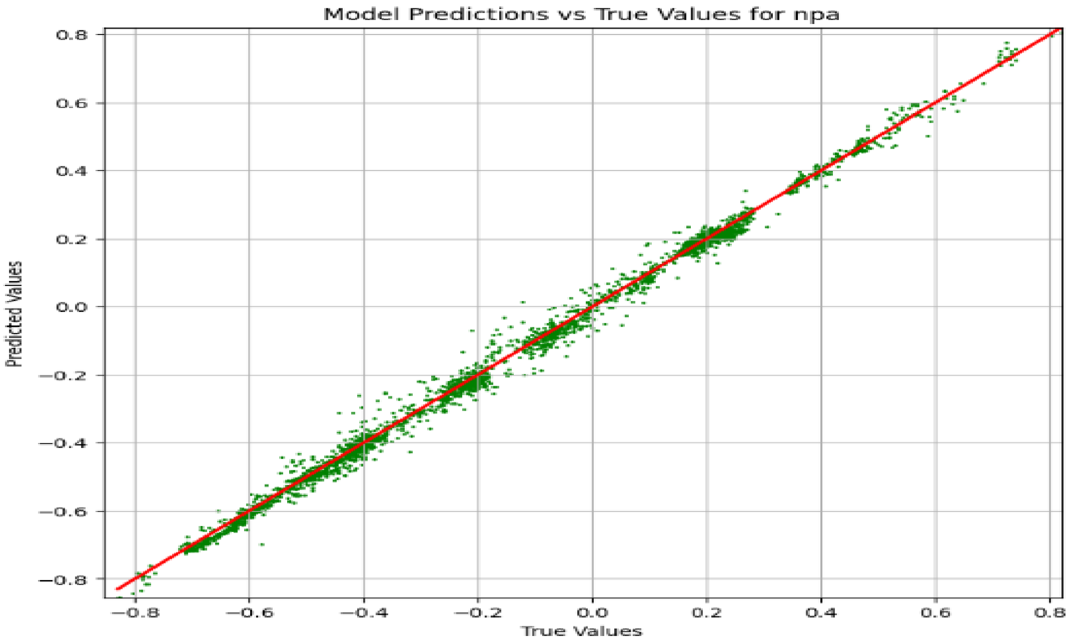


Figure 30: model predictions vs true values for npa

This scatter above also illustrates the relationship between predicted and true values of ‘npa’ for the final model. It shows a high correlation as the data points closely align with the red diagonal line, suggesting the final model’s predictions are highly accurate across the value range.

2. Error distribution analysis

The code below I used to print histograms for analyzing the error distribution:

```
# error distribution.
if not isinstance(y_test, pd.DataFrame):
    y_test = pd.DataFrame(y_test, columns=['Y1', 'Y2', 'Y3'])
# Calculate errors and plot histograms for each target variable
for i, column in enumerate(y_test.columns):
    plt.figure(figsize=(10, 4))
    # Calculate the error
    true_values = y_test[column]
    predicted_values = predictions[:, i]
    errors = predicted_values - true_values
    # Plotting the error distribution
    plt.hist(errors, bins=25, color='skyblue', edgecolor='black')
    plt.title(f"Prediction Error Distribution for {column}")
    plt.xlabel(f"Prediction Error [{column}]")
    plt.ylabel("Count")
    plt.grid(True)
    plt.show()
```

The Figure 31 below represents prediction error distribution for target ‘cm5’:

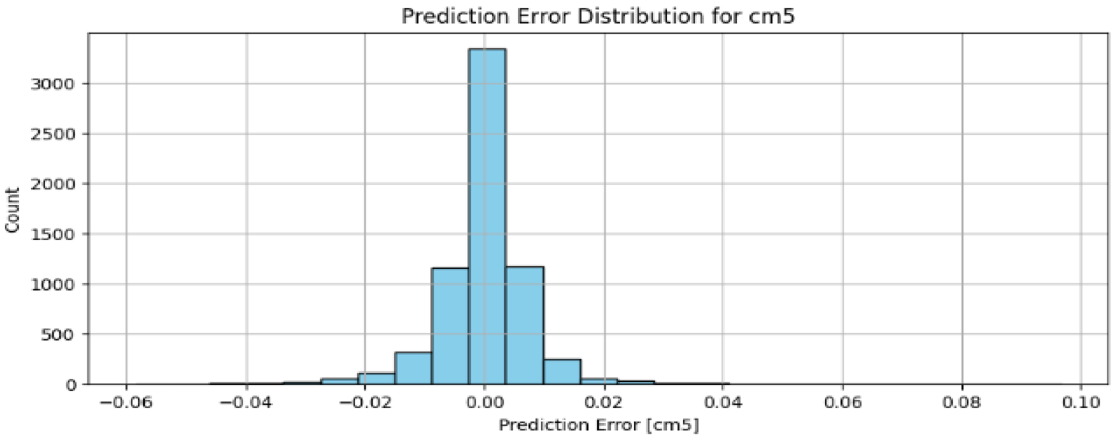


Figure 31: Error distribution for cm5

The histogram above shows the prediction error distribution for the charge ‘cm5’. We notice that the data is centered around zero, indicating that most predictions are very close to the true values.

The Figure 32 below shows prediction error distribution for target esp:

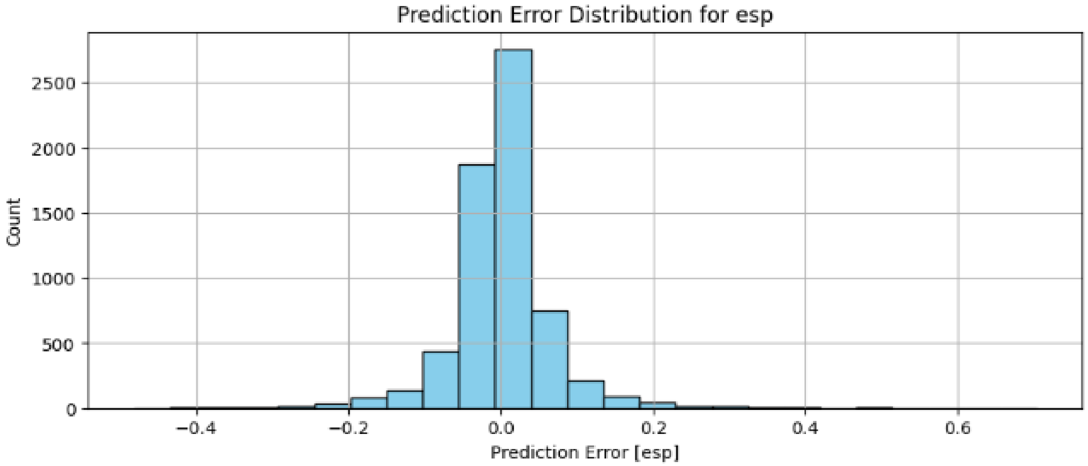


Figure 32: Error distribution for esp

The histogram above represents the prediction error distribution for the charge ‘esp’. The data forms symmetric distribution around zero, highlighting that most prediction errors are small and centered around zero. This indicates that the model generally performs well, but it overestimates frequently.

The Figure 33 below shows prediction error distribution for target ‘npa’:

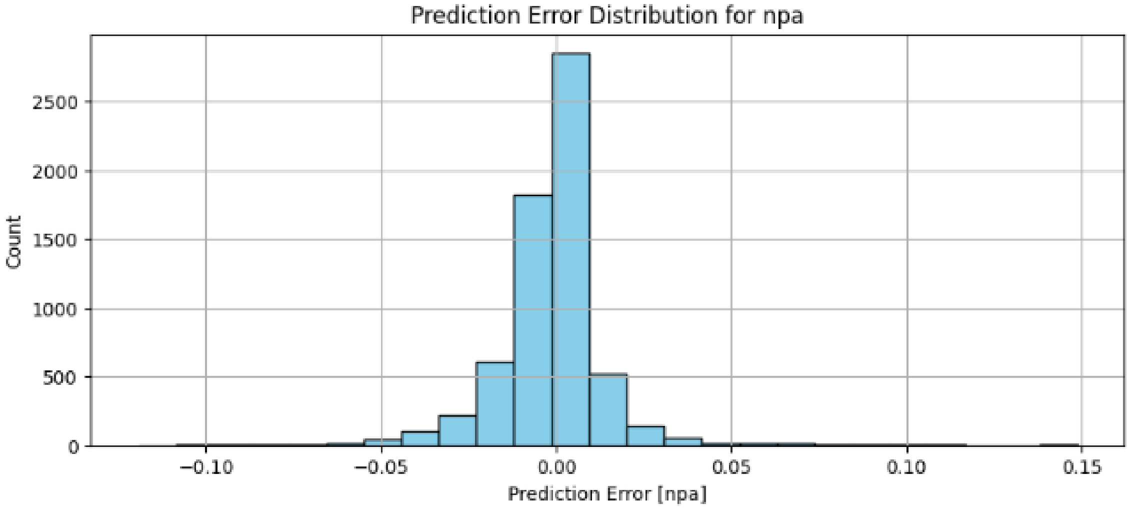


Figure 33: Error distribution for npa

The last histogram illustrates the prediction error distribution for the charge ‘npa’. The errors are centered closely around zero, indicating that the model typically makes very small errors in its predictions.

3. Initial model and final model comparison

It was interesting to see how the initial model and the final model, obtained after refining the hyperparameters, performed differently. Although, both models demonstrated good accuracy in predicting partial charges with low Mean Absolute Error (MAE) and Mean Squared Error (MSE) throughout training and validation sets, the first model showed a hint of overfitting.

During training the first model, this last appeared to be slightly overfitting the data at first, and this was observed in its lower error on the training dataset compared to the validation dataset.

However, after the hyperparameters process, the final model displays a more balanced and consistent performance across training and validation data. This suggests that the model's ability to generalize to new data has improved and it is less likely to memorize specific patterns in the training dataset. Overall, while resulting in similar error metrics, the hyperparameters tuning process appears to have made the model more robust and reliable.

VI. Conclusion

This thesis demonstrated the development of a neural network model capable of predicting partial charges in organic molecules, offering a computationally efficient alternative to traditional quantum chemistry approaches.

Initial results were highly encouraging, with the model learning complex relationships between molecular geometry and partial charges. This was reflected in low Mean Absolute Error (MAE) and Mean Squared Error (MSE) values across training and validation sets. While a slight tendency towards overfitting was initially observed, hyperparameter tuning successfully addressed this issue. The refined model exhibited improved generalization with consistent performance, indicating a greater potential for reliable partial charge prediction on new molecules.

Overall, this thesis demonstrates the power of machine learning for chemical property prediction. It highlights the value of iterative model development and paves the way for continued advancements in computationally efficient and accurate partial charge estimation.

VII. Acknowledgements

I would like to express my gratitude to the Chemistry Department of the University of Pécs, and especially to my supervisor, Dr. Tamás Kégl, for conducting the quantum chemistry calculations and providing the essential dataset that formed the foundation of this work.

For my family, especially for my mother and father, this work would not have been possible without you.

VIII. References list

About Python^{TM} | Python.org. (n.d.). Retrieved April 30, 2024, from https://www.python.org/about/

API Documentation | TensorFlow v2.16.1. (n.d.). Retrieved April 30, 2024, from https://www.tensorflow.org/api_docs

Artrith, N., Butler, K. T., Coudert, F.-X., Han, S., Isayev, O., Jain, A., & Walsh, A. (n.d.). Best practices in machine learning for chemistry. https://doi.org/10.1038/s41557-021-00716-z

Brehm, M., & Thomas, M. (2021). molecules Optimized Atomic Partial Charges and Radii Defined by Radical Voronoi Tessellation of Bulk Phase Simulations. https://doi.org/10.3390/molecules26071875

Differences Between Epoch, Batch, and Mini-batch | Baeldung on Computer Science. (n.d.). Retrieved May 2, 2024, from https://www.baeldung.com/cs/epoch-vs-batch-vs-mini-batch

Frazier, P. I. (2018). A Tutorial on Bayesian Optimization.

GRAMMAR-INDUCED GEOMETRY FOR DATA-EFFICIENT MOLECULAR PROPERTY PREDICTION. (n.d.).

Heid, E., Greenman, K. P., Chung, Y., Li, S. C., Graff, D. E., Vermeire, F. H., Wu, H., Green, W. H., & McGill, C. J. (2024). Chemprop: A Machine Learning Package for Chemical Property Prediction. Journal of Chemical Information and Modeling, 64(1), 9–17. https://doi.org/10.1021/acs.jcim.3c01250

JupyterLab Documentation — JupyterLab 4.2.0rc0 documentation. (n.d.). Retrieved April 30, 2024, from https://jupyterlab.readthedocs.io/en/latest/

Keras: Deep Learning for humans. (n.d.). Retrieved April 30, 2024, from https://keras.io/

KerasTuner. (n.d.). Retrieved April 30, 2024, from https://keras.io/keras_tuner/

Machine Learning with Python Tutorial. (n.d.). Retrieved April 30, 2024, from https://www.geeksforgeeks.org/machine-learning-with-python/

Martin, R., & Heider, D. (2019). Contradrg: Automatic partial charge prediction by machine learning. Frontiers in Genetics, 10(OCT). https://doi.org/10.3389/FGENE.2019.00990

Matplotlib — Visualization with Python. (n.d.). Retrieved April 30, 2024, from https://matplotlib.org/

Meaning and Definition of Partial Charges. (n.d.).

neural network - Why should the data be shuffled for machine learning tasks - Data Science Stack Exchange. (n.d.). Retrieved May 2, 2024, from https://datascience.stackexchange.com/questions/24511/why-should-the-data-be-shuffled-for-machine-learning-tasks

NumPy -. (n.d.). Retrieved April 30, 2024, from https://numpy.org/

pandas - Python Data Analysis Library. (n.d.). Retrieved April 30, 2024, from https://pandas.pydata.org/

Partial Atomic Charge Derivation of small molecule. (n.d.).

(PDF) Machine Learning: A Review of Learning Types. (n.d.). Retrieved May 2, 2024, from https://www.researchgate.net/publication/342890321_Machine_Learning_A_Review_of_Learning_Types

scikit-learn: machine learning in Python — scikit-learn 1.4.2 documentation. (n.d.). Retrieved April 30, 2024, from https://scikit-learn.org/stable/

SciPy -. (n.d.). Retrieved April 30, 2024, from https://scipy.org/

TensorFlow. (n.d.). Retrieved April 30, 2024, from https://www.tensorflow.org/

Raček T, Schindler O, Toušek D, Horský V, Berka K, Koča J, Svobodová R. (2020). Atomic Charge Calculator II: web-based tool for the calculation of partial atomic charges. Nucleic Acids Research, 48, 591–596. https://doi.org/10.1093/nar/gkaa367

Uhliar, M. (2024). Atomic partial charge model in chemistry: chemical accuracy of theoretical approaches for diatomic molecules. Acta Chimica Slovaca, 17(1), 1–11. https://doi.org/10.2478/ACS-2024-0001

Wang, J., Cao, D., Tang, C., Chen, X., Sun, H., & Hou, T. (n.d.). Structural bioinformatics Fast and accurate prediction of partial charges using Atom-Path-Descriptor-based machine learning. https://doi.org/10.1093/bioinformatics/btaa566

Wang, Y. (n.d.). EspalomaCharge: Machine learning-enabled ultra-fast partial charge assignment. Retrieved April 30, 2024, from https://github.com/choderalab/espaloma_charge.

Waskom, M. (2021). seaborn: statistical data visualization. Journal of Open Source Software, 6(60), 3021. https://doi.org/10.21105/JOSS.03021

What is Feature Scaling and Why is it Important. (n.d.). Retrieved May 2, 2024, from https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

What Is Supervised Learning? | IBM. (n.d.). Retrieved May 2, 2024, from https://www.ibm.com/topics/supervised-learning