Perceptrons Assignment Report

ÖZYEĞİN UNIVERSITY

Assignment #3:

Lecture: CS454/554 - Introduction to Machine Learning and Artificial Neural Networks

Instructor: Prof. Ethem Alpaydın

Tuna Tuncer – S018474

1. Introduction

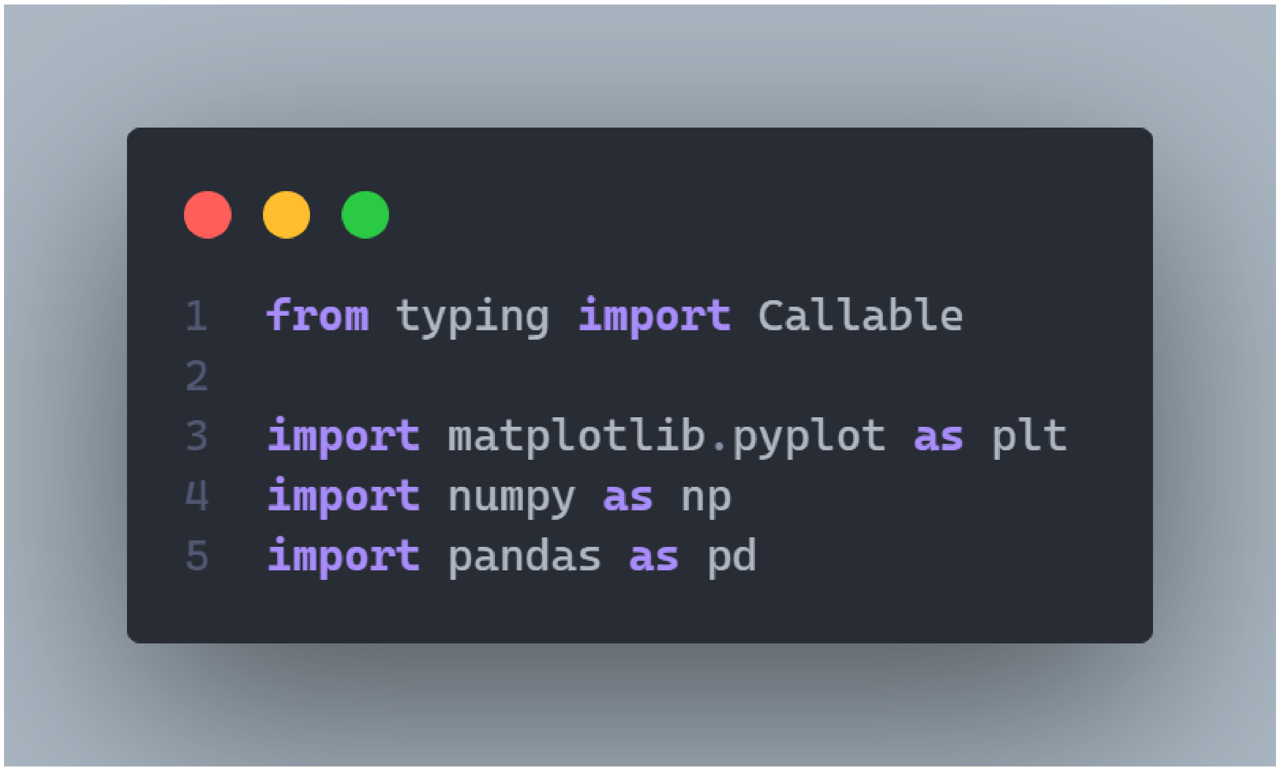
I was given one test data set with 100 instances and ten training data sets, each having 25 instances, for this assignment. I was expected to create single layer perceptron (SLP) and multi-layer perceptron (MLP) regression models and train them using the training data sets, and then measure and compare the mean squared errors of the models using the test data set. It was desired that the MLP models I would develop had 2, 3, 5, and 10 neurons (H) in their hidden layers. The data was generated using a hidden f(x) function and made more random by adding gaussian noise to it.

2. Methodology

For each H value, 10 different models were created, and the mean squared errors of each of these 10 models were calculated. Afterwards, a single value was obtained by averaging 10 different mean squared error values. When this process was repeated for each H value, a total of 6 different average mean square error values were obtained. Thus, it has been empirically found that the models with which H value explains the data more successfully.

3. Implementation Details

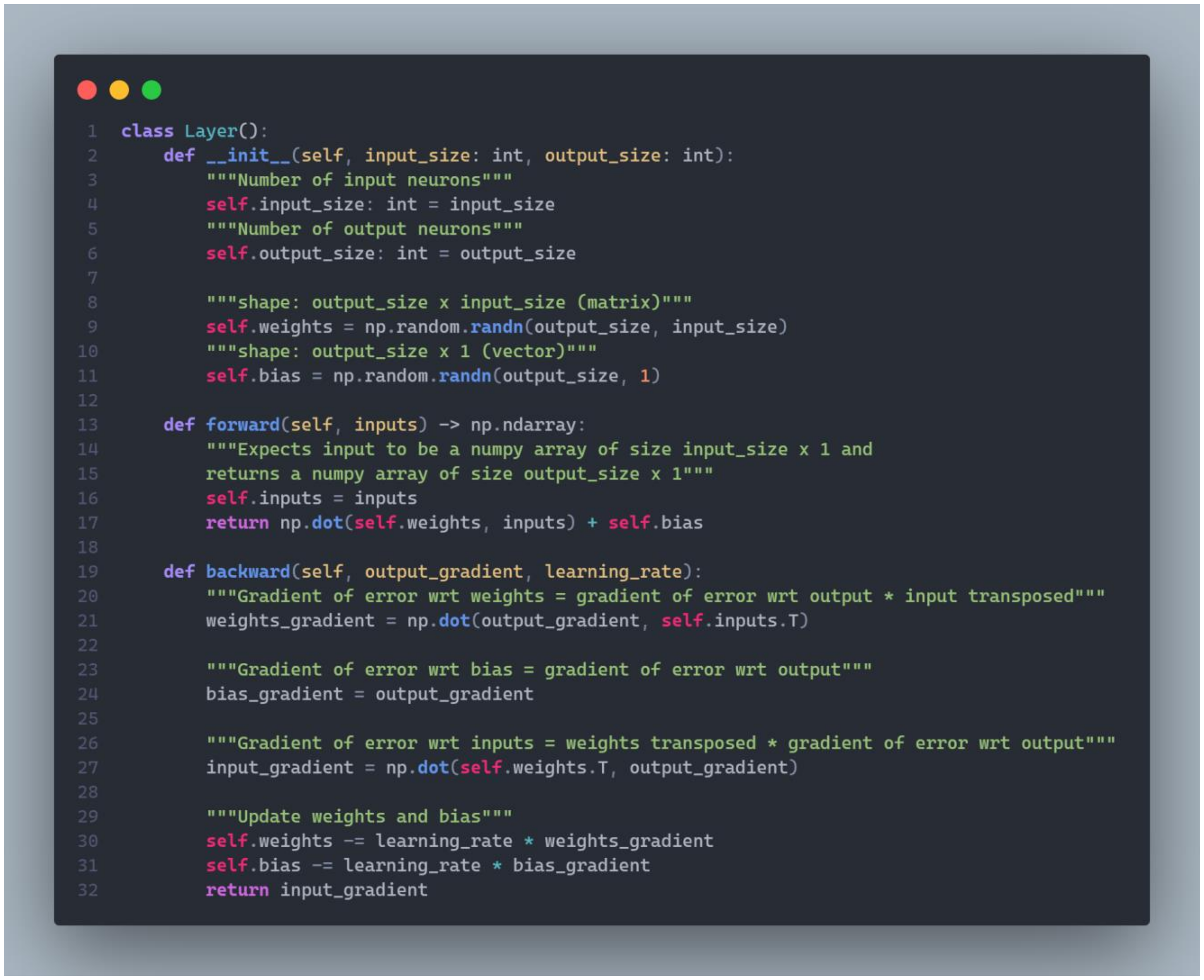
Python is used as the programming language for this assignment. In addition, Jupyter Notebook, which is widely used for machine learning related tasks, is also used as an editor.



```python
from typing import Callable

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

Figure [1]: Imported Libraries

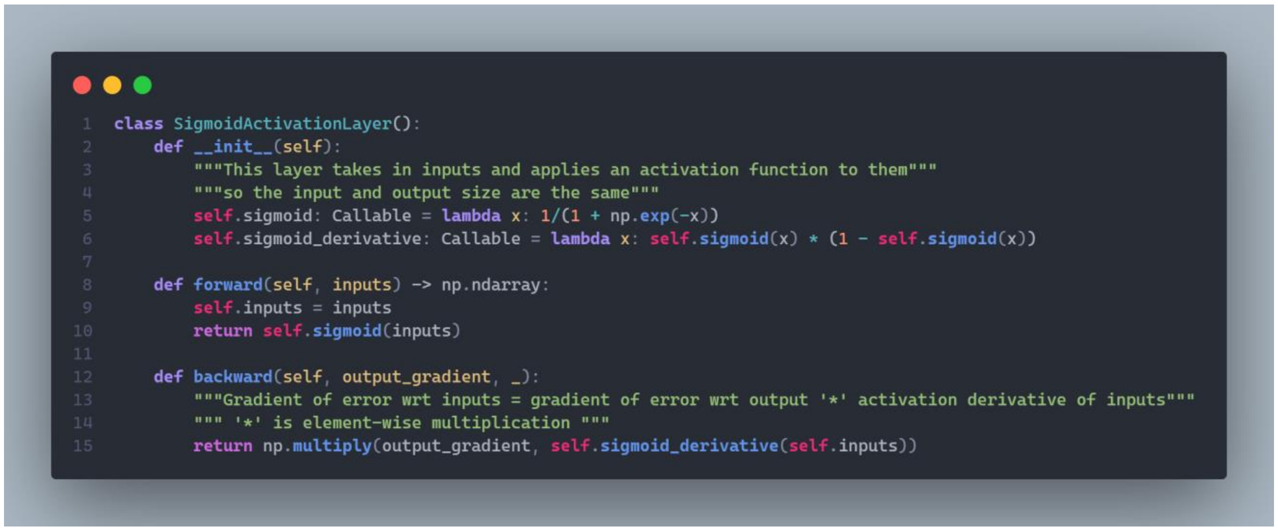
As seen in Figure 1, necessary libraries are imported. Pandas is used to read data from CSV files and save it in data frames. NumPy is used to store data in arrays, to take average, exponentiation, and power, to create random numbers, and for dot and element-wise multiplication purposes. Matplotlib is used to visualize the entire process. Typing is used to define types for variables throughout the code.



```python
class Layer():
    def __init__(self, input_size: int, output_size: int):
        """Number of input neurons"""
        self.input_size: int = input_size
        """Number of output neurons"""
        self.output_size: int = output_size

        """shape: output_size x input_size (matrix)"""
        self.weights = np.random.randn(output_size, input_size)
        """shape: output_size x 1 (vector)"""
        self.bias = np.random.randn(output_size, 1)

    def forward(self, inputs) -> np.ndarray:
        """Expects input to be a numpy array of size input_size x 1 and
        returns a numpy array of size output_size x 1"""
        self.inputs = inputs
        return np.dot(self.weights, inputs) + self.bias

    def backward(self, output_gradient, learning_rate):
        """Gradient of error wrt weights = gradient of error wrt output * input transposed"""
        weights_gradient = np.dot(output_gradient, self.inputs.T)

        """Gradient of error wrt bias = gradient of error wrt output"""
        bias_gradient = output_gradient

        """Gradient of error wrt inputs = weights transposed * gradient of error wrt output"""
        input_gradient = np.dot(self.weights.T, output_gradient)

        """Update weights and bias"""
        self.weights -= learning_rate * weights_gradient
        self.bias -= learning_rate * bias_gradient
        return input_gradient
```

Figure [2]: Dense Layer Class

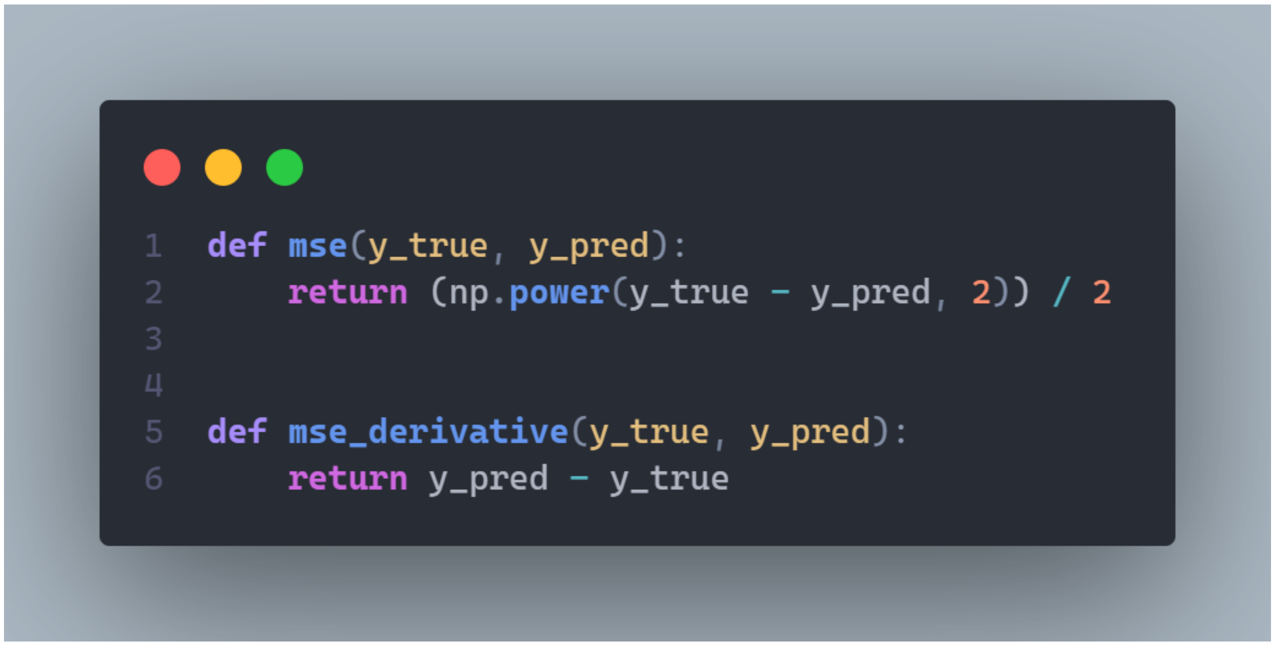
In order for all models to be created generically, I defined a class named ***Layer*** as seen in Figure 2. Weights and biases were kept separately in the layer class. In addition, forward and backward methods are defined in the class, and these are used while training the model.



```python
class SigmoidActivationLayer():
    def __init__(self):
        """This layer takes in inputs and applies an activation function to them"""
        """so the input and output size are the same"""
        self.sigmoid: Callable = lambda x: 1/(1 + np.exp(-x))
        self.sigmoid_derivative: Callable = lambda x: self.sigmoid(x) * (1 - self.sigmoid(x))

    def forward(self, inputs) -> np.ndarray:
        self.inputs = inputs
        return self.sigmoid(inputs)

    def backward(self, output_gradient, _):
        """Gradient of error wrt inputs = gradient of error wrt output '*' activation derivative of inputs"""
        """ '*' is element-wise multiplication """
        return np.multiply(output_gradient, self.sigmoid_derivative(self.inputs))
```

Figure [3]: Sigmoid Activation Layer Class

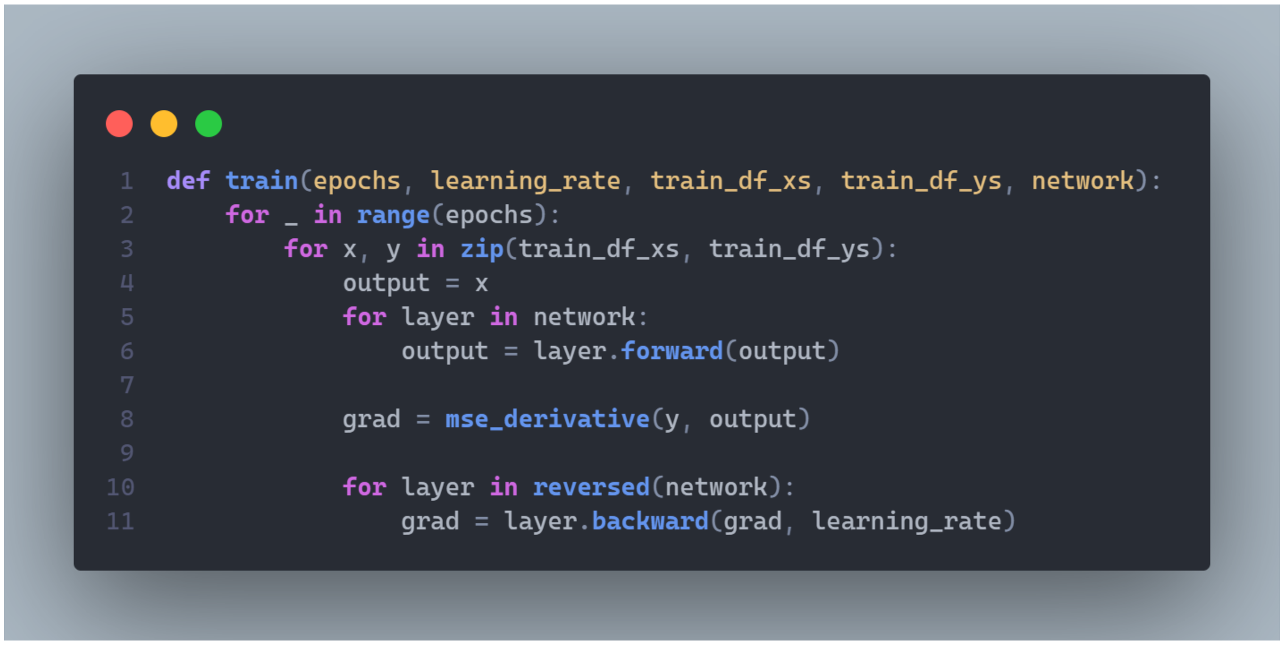
Next, I defined the ***SigmoidActivationLayer*** class to use sigmoid as a non-linearity function, as seen in Figure 3.



```python
def mse(y_true, y_pred):
    return (np.power(y_true - y_pred, 2)) / 2


def mse_derivative(y_true, y_pred):
    return y_pred - y_true
```

Figure [4]: Functions of Mean Squared Error and Derivative of Mean Squared Error

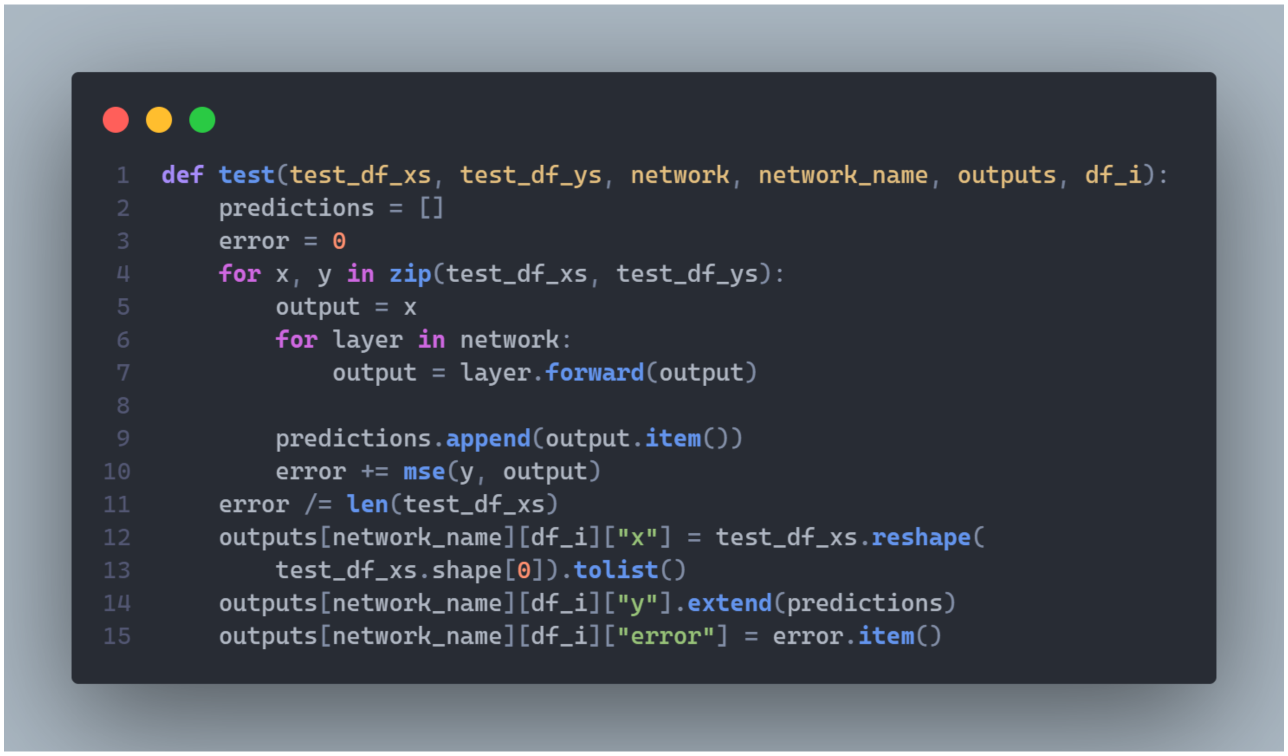
Then, in figure 4, I created the *mse* function to calculate the model's error and the *mse_derivative* function to use it in the backpropagation part.



```python
def train(epochs, learning_rate, train_df_xs, train_df_ys, network):
    for _ in range(epochs):
        for x, y in zip(train_df_xs, train_df_ys):
            output = x
            for layer in network:
                output = layer.forward(output)

            grad = mse_derivative(y, output)

            for layer in reversed(network):
                grad = layer.backward(grad, learning_rate)
```

Figure [5]: Train Function

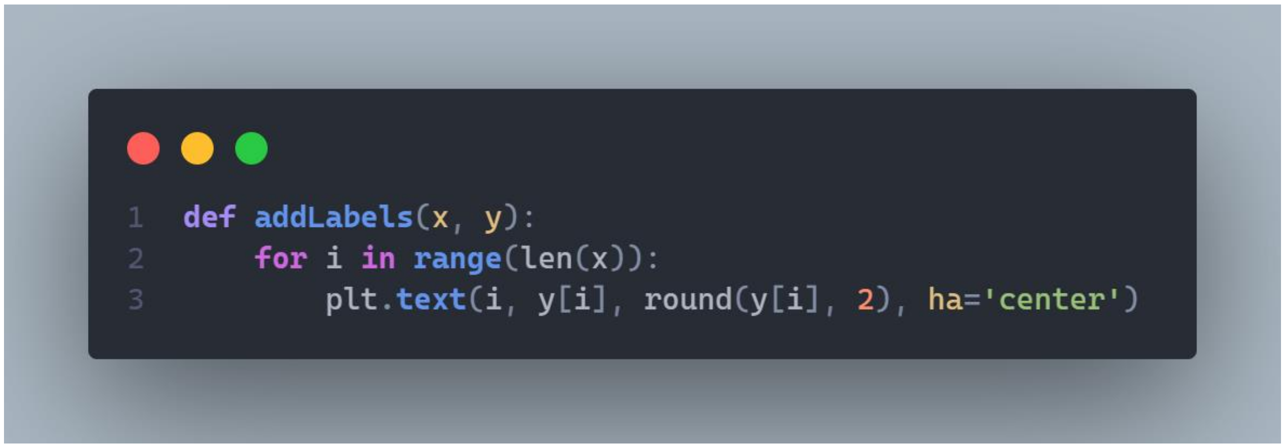
In Figure 5, I created the train function that I will use while training the model, you can see that I use the forward and backward functions defined above in this function.



```python
def test(test_df_xs, test_df_ys, network, network_name, outputs, df_i):
    predictions = []
    error = 0
    for x, y in zip(test_df_xs, test_df_ys):
        output = x
        for layer in network:
            output = layer.forward(output)

        predictions.append(output.item())
        error += mse(y, output)
    error /= len(test_df_xs)
    outputs[network_name][df_i]["x"] = test_df_xs.reshape(
        test_df_xs.shape[0]).tolist()
    outputs[network_name][df_i]["y"].extend(predictions)
    outputs[network_name][df_i]["error"] = error.item()
```

Figure [6]: Test Function

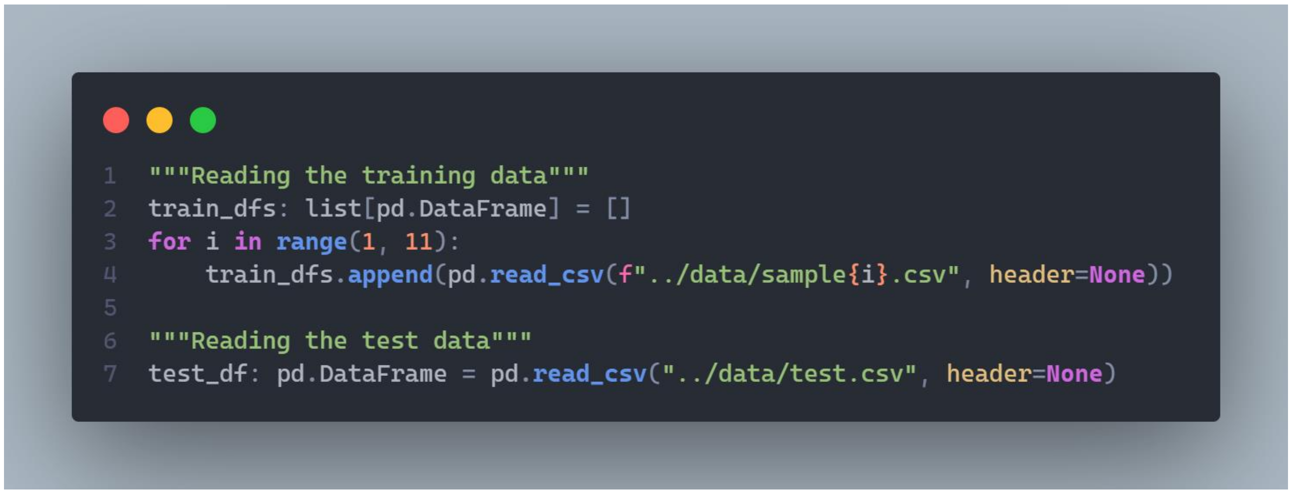
In Figure 6, you can see the test function I will use while testing the model. Also, in this method, I have saved the predictions of each model in a dictionary named outputs.



```python
def addLabels(x, y):
    for i in range(len(x)):
        plt.text(i, y[i], round(y[i], 2), ha='center')
```

Figure [7]: Add Labels Function

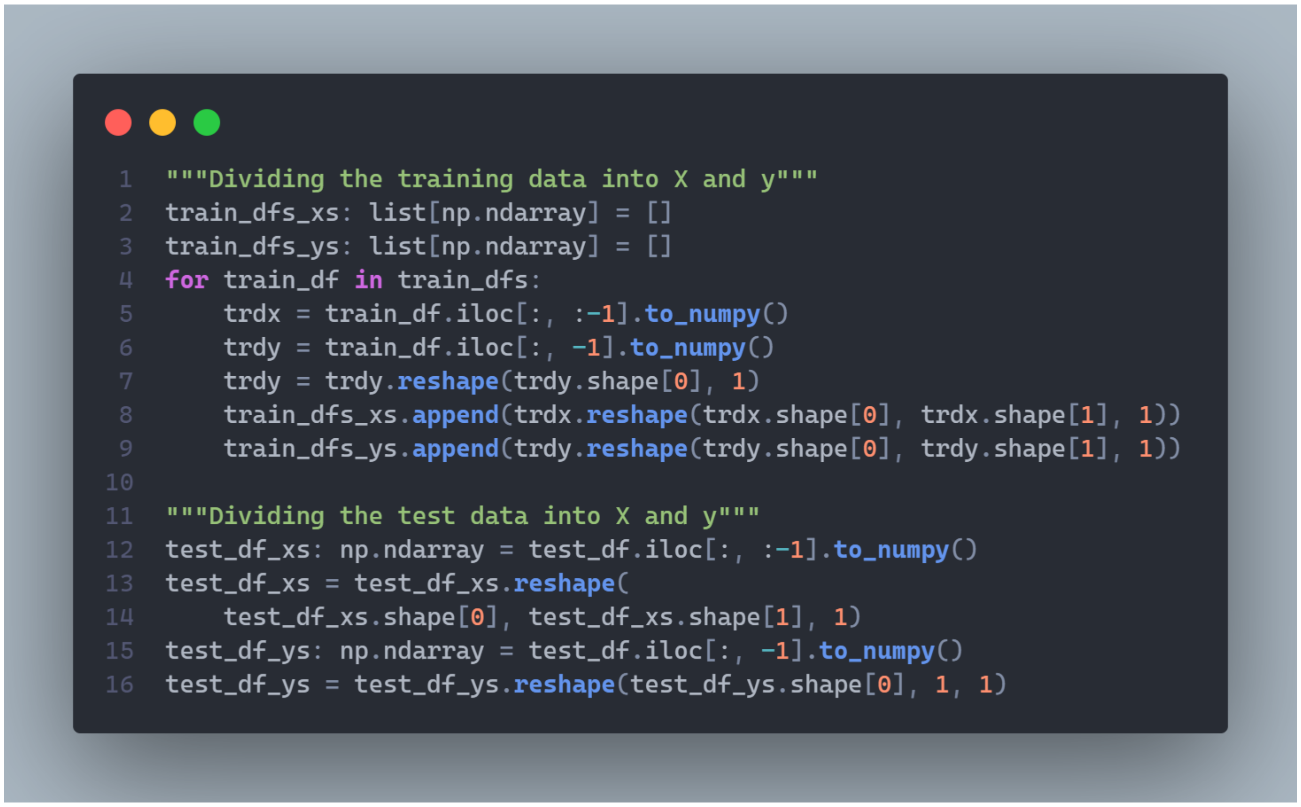
The *addLabels* function in Figure 7 is used to make the MSE values appear above the bars in the MSE vs h graph that you will see in the results section.



```python
"""Reading the training data"""
train_dfs: list[pd.DataFrame] = []
for i in range(1, 11):
    train_dfs.append(pd.read_csv(f"../data/sample{i}.csv", header=None))

"""Reading the test data"""
test_df: pd.DataFrame = pd.read_csv("../data/test.csv", header=None)
```

Figure [8]: Reading the Training and Test Data

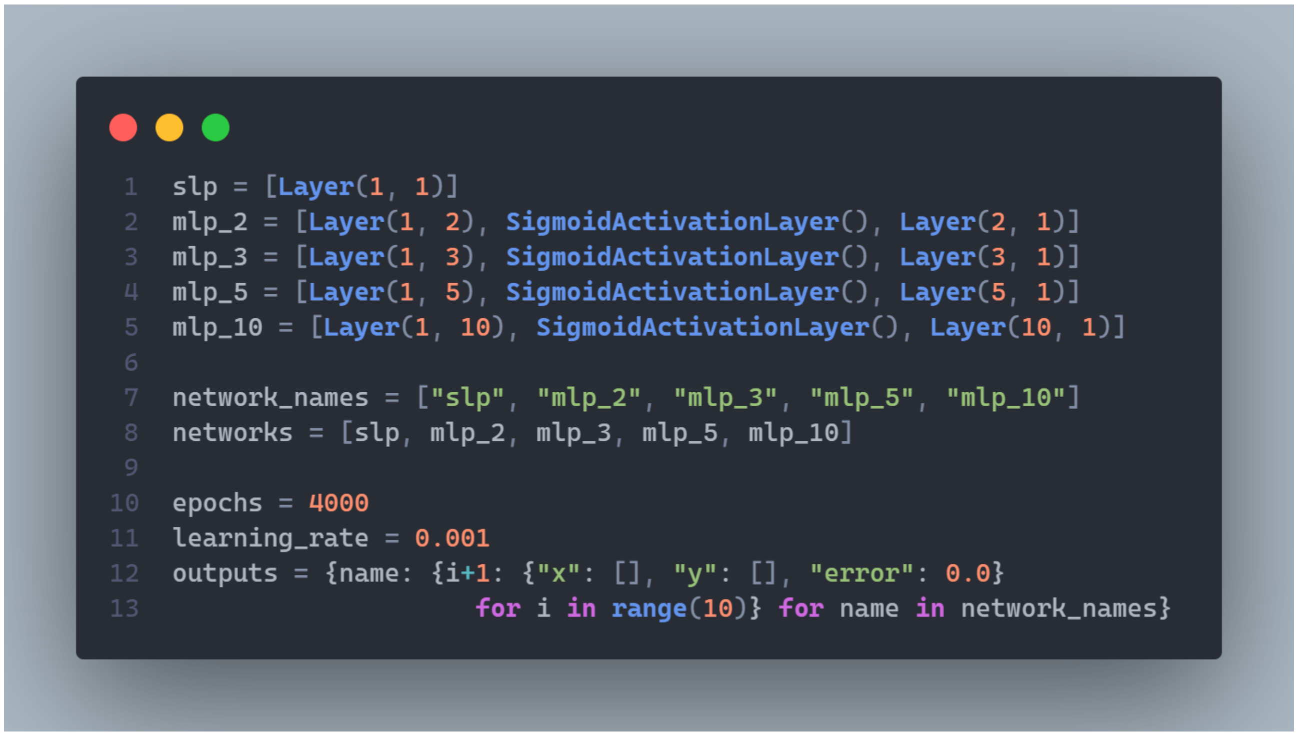
Finally, in figure 8, we read our train and test data from different csv files and save them in different NumPy arrays.



```python
"""Dividing the training data into X and y"""
train_dfs_xs: list[np.ndarray] = []
train_dfs_ys: list[np.ndarray] = []
for train_df in train_dfs:
    trdx = train_df.iloc[:, :-1].to_numpy()
    trdy = train_df.iloc[:, -1].to_numpy()
    trdy = trdy.reshape(trdy.shape[0], 1)
    train_dfs_xs.append(trdx.reshape(trdx.shape[0], trdx.shape[1], 1))
    train_dfs_ys.append(trdy.reshape(trdy.shape[0], trdy.shape[1], 1))

"""Dividing the test data into X and y"""
test_df_xs: np.ndarray = test_df.iloc[:, :-1].to_numpy()
test_df_xs = test_df_xs.reshape(
    test_df_xs.shape[0], test_df_xs.shape[1], 1)
test_df_ys: np.ndarray = test_df.iloc[:, -1].to_numpy()
test_df_ys = test_df_ys.reshape(test_df_ys.shape[0], 1, 1)
```

Figure [9]: Dividing and Reshaping Training and Test Data

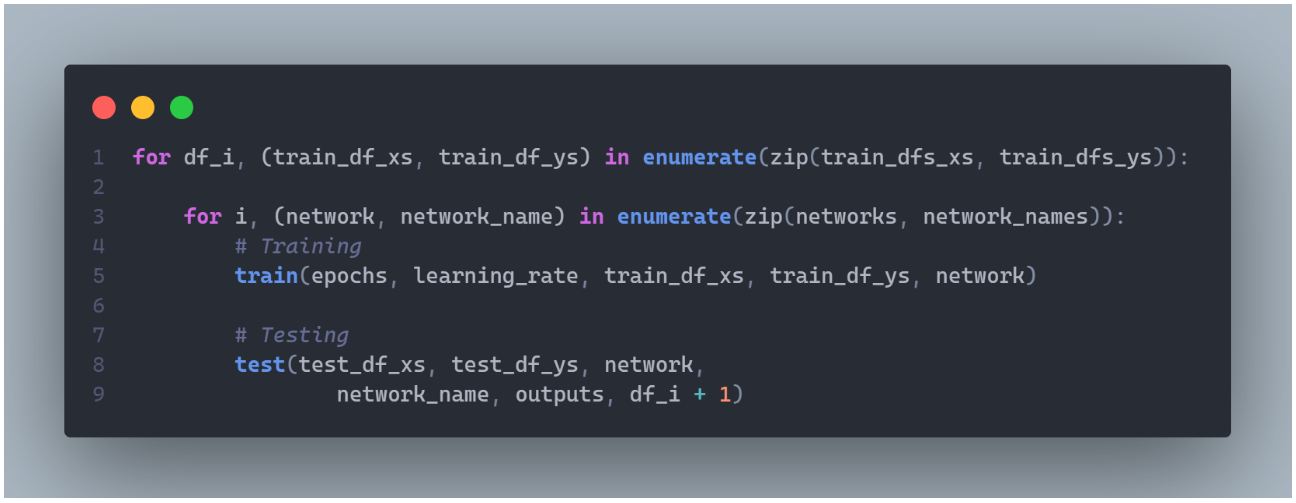
In Figure 9, before training the model, I ensure that our data is separated into x, y columns and reshaped so that I can provide this data as input to our model.



```python
slp = [Layer(1, 1)]
mlp_2 = [Layer(1, 2), SigmoidActivationLayer(), Layer(2, 1)]
mlp_3 = [Layer(1, 3), SigmoidActivationLayer(), Layer(3, 1)]
mlp_5 = [Layer(1, 5), SigmoidActivationLayer(), Layer(5, 1)]
mlp_10 = [Layer(1, 10), SigmoidActivationLayer(), Layer(10, 1)]

network_names = ["slp", "mlp_2", "mlp_3", "mlp_5", "mlp_10"]
networks = [slp, mlp_2, mlp_3, mlp_5, mlp_10]

epochs = 4000
learning_rate = 0.001
outputs = {name: {i+1: {"x": [], "y": [], "error": 0.0}
                    for i in range(10)} for name in network_names}
```

Figure [10]: SLP & MLP Models, Hyperparameters, and Outputs List

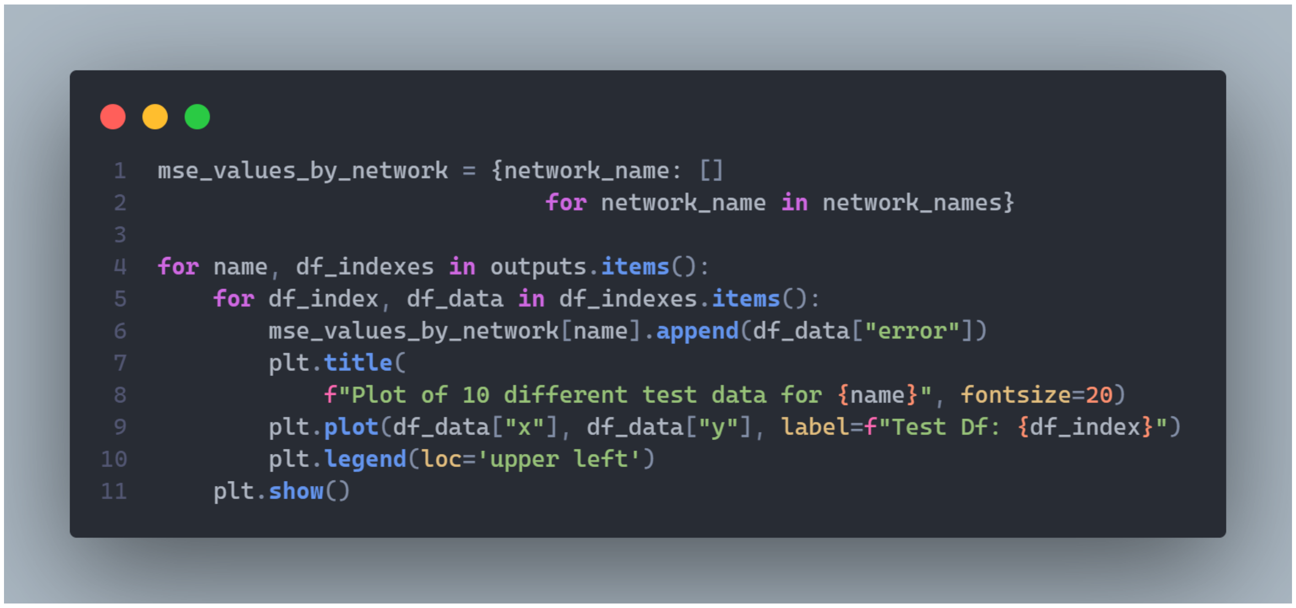
In Figure 10, you can see how I created 5 different models and assigned parameters such as epoch and learning rate to variables before training these models.



```python
for df_i, (train_df_xs, train_df_ys) in enumerate(zip(train_dfs_xs, train_dfs_ys)):

    for i, (network, network_name) in enumerate(zip(networks, network_names)):
        # Training
        train(epochs, learning_rate, train_df_xs, train_df_ys, network)

        # Testing
        test(test_df_xs, test_df_ys, network,
                network_name, outputs, df_i + 1)
```

Figure [11]: Main Algorithm for Training and Testing

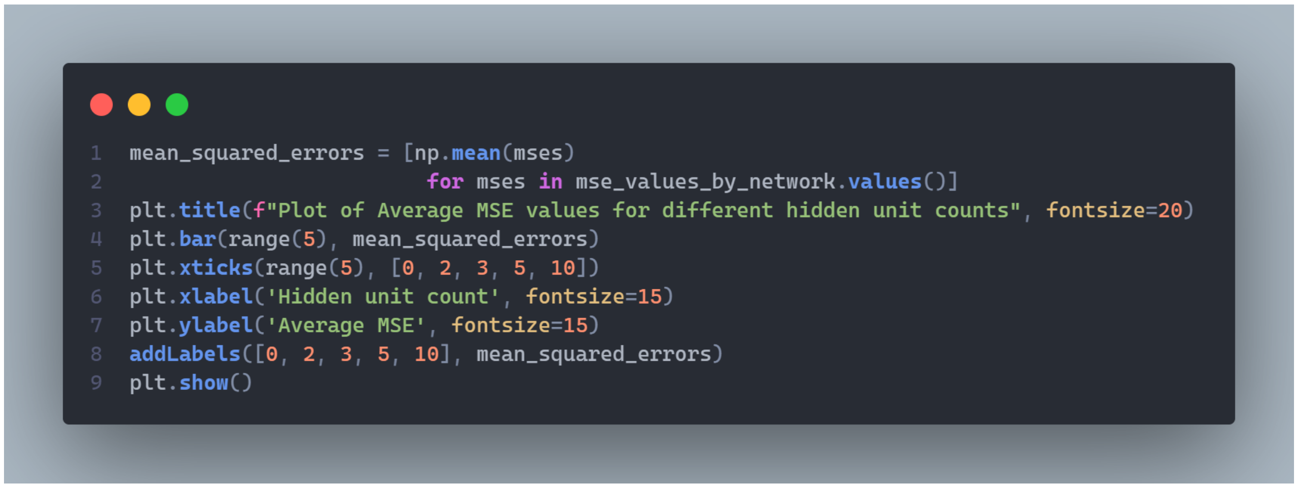
In Figure 11 you can see how I added the train and test functions we created to the algorithm to train and test the models.



```python
mse_values_by_network = {network_name: []
                         for network_name in network_names}

for name, df_indexes in outputs.items():
    for df_index, df_data in df_indexes.items():
        mse_values_by_network[name].append(df_data["error"])
        plt.title(
            f"Plot of 10 different test data for {name}", fontsize=20)
        plt.plot(df_data["x"], df_data["y"], label=f"Test Df: {df_index}")
        plt.legend(loc='upper left')
    plt.show()
```

Figure [12]: Plotting of Predicted Test Functions

In Figure 12, 50 different graphs were drawn in 5 different plots using test set predictions for each type of model trained with H value.



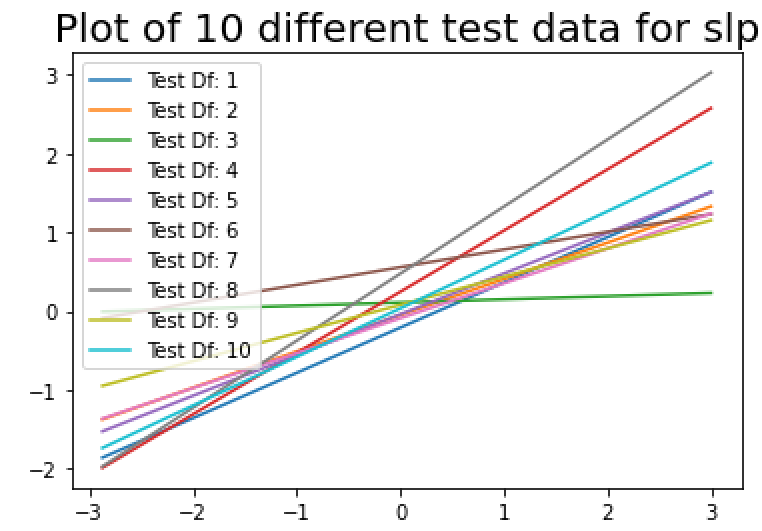
```python
mean_squared_errors = [np.mean(mses)
                       for mses in mse_values_by_network.values()]
plt.title(f"Plot of Average MSE values for different hidden unit counts", fontsize=20)
plt.bar(range(5), mean_squared_errors)
plt.xticks(range(5), [0, 2, 3, 5, 10])
plt.xlabel('Hidden unit count', fontsize=15)
plt.ylabel('Average MSE', fontsize=15)
addLabels([0, 2, 3, 5, 10], mean_squared_errors)
plt.show()
```

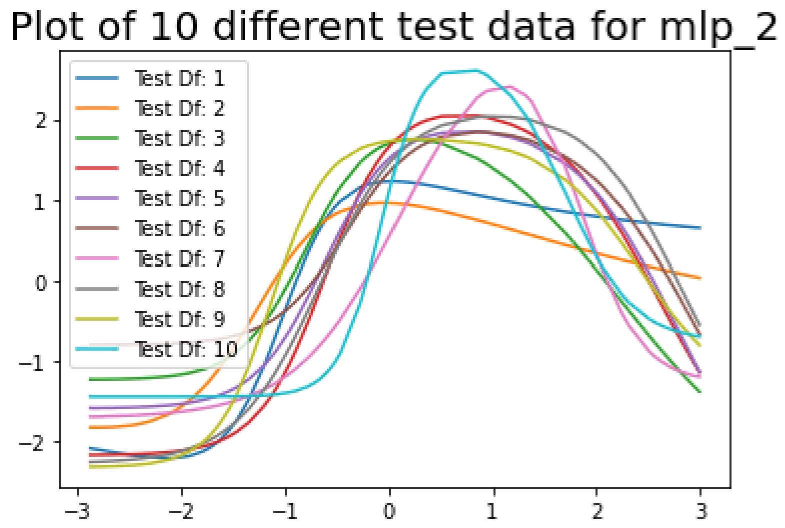
Figure [13]: Plotting of Average MSE

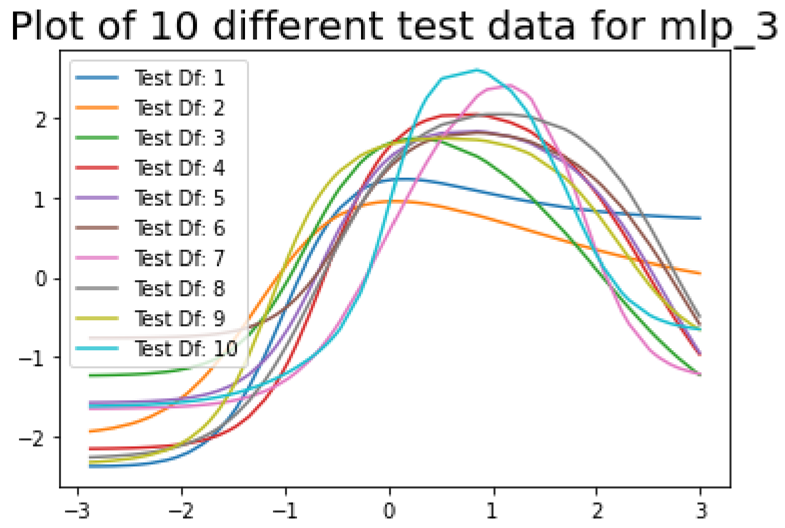
Finally, in figure 13, the MSE value of each network was calculated and plotted by taking the average of the 10 error values calculated for each network.

4. Results

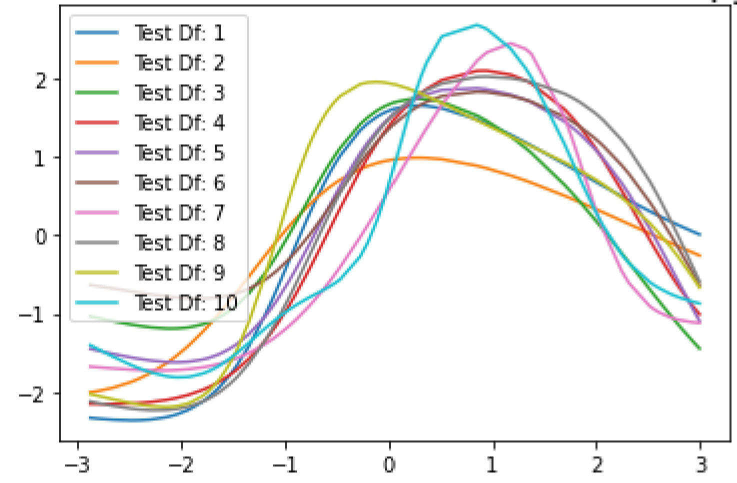
Below you can see the plots of 10 different predictions and average mean squared errors on every H value.


Plot of 10 different test data for slp
Test Df: 1
Test Df: 2
Test Df: 3
Test Df: 4
Test Df: 5
Test Df: 6
Test Df: 7
Test Df: 8
Test Df: 9
Test Df: 10
−3
−2
−1
0
1
2
3

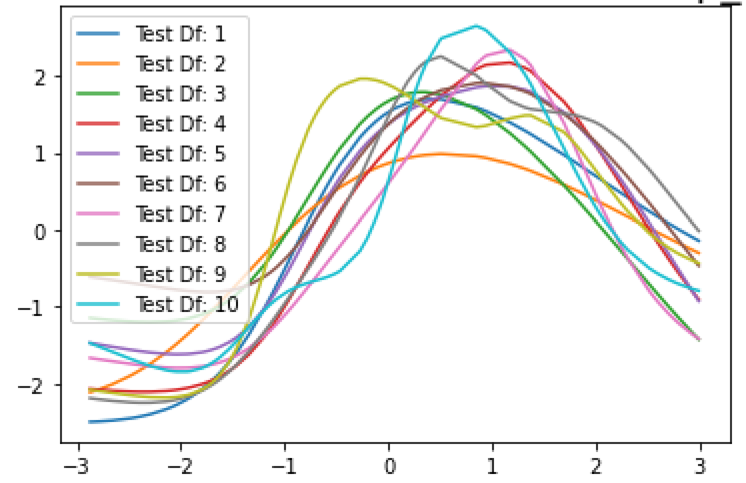

Plot of 10 different test data for mlp_2
Test Df: 1
Test Df: 2
Test Df: 3
Test Df: 4
Test Df: 5
Test Df: 6
Test Df: 7
Test Df: 8
Test Df: 9
Test Df: 10
−3
−2
−1
0
1
2
3


Plot of 10 different test data for mlp_3
Test Df: 1
Test Df: 2
Test Df: 3
Test Df: 4
Test Df: 5
Test Df: 6
Test Df: 7
Test Df: 8
Test Df: 9
Test Df: 10
−3
−2
−1
0
1
2
3

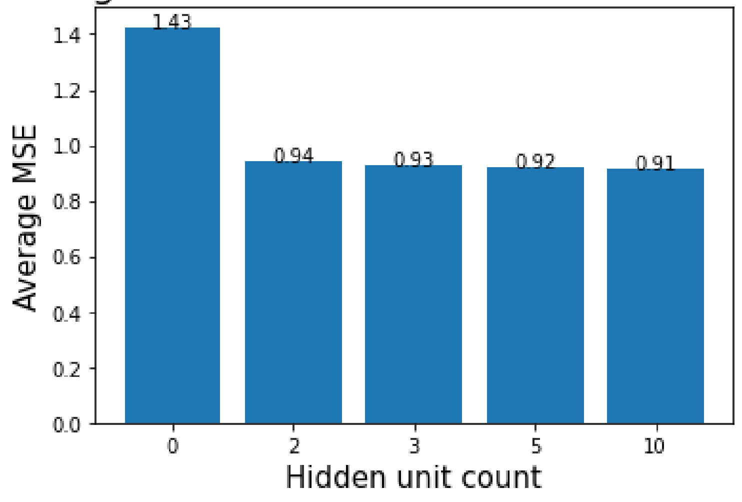
Plot of 10 different test data for mlp_5



Plot of 10 different test data for mlp_10



Plot of Average MSE values for different hidden unit counts



As a result, we see that the MSE value decreases as H increases, but as the model becomes more complex, the chance of overfitting also increases.