CAP 6615
Neural Network

Programming Assignment 3 -- Recurrent Neural Network

Group Member:

Fugang Deng

Wei He

Hang Ye

Zihang Huang

Jianan He

Weijia Sun

University of Florida

CAP 6615

CAP 6615 - Neural Networks - Programming Assignment 3 – Recurrent NN

Group Member: Fugang Deng, Wei He, Hang Ye, Zihang Huang, Jianan He, Weijia Sun

Spring Semester 2022

18 Mar 2022

1 Network parameters

Parameter name	Value
epoch	100
Learning Rate	0.001
Batch Size	36
Input Size	4
Output Size	4
Sequence Length	180
Hidden Size	60
Num layer	1
Input dimension	4*180/time
Output dimension	4*1/time
Standard Deviation for Noise	0.001, 0.002, 0.003, 0.005, 0.01, 0.02, 0.03, 0.05 and 0.1

Table 1 Parameters

To run the code successfully, we would like to present the external library we used in python, and please make sure you install all the necessary libraries before running the code.

Library Name
Torch
sklearn
Numpy
Matplotlib

Table 2 Necessary Library

2 Python code for RNN

2.1 S&P Dataset

```
In [2]: data = pd.read_csv('data.csv')
        data['Date'] = pd.to_datetime(data['Date'], format='%Y-%m-%d')
        data
```

Out[2]:

	Date	Close	Open	High	Low	PE	Ratio
0	1960-01-04	59.910000	59.910000	59.910000	59.910000	18.34	17.12
1	1960-01-05	60.389999	60.389999	60.389999	60.389999	18.34	17.12
2	1960-01-06	60.130001	60.130001	60.130001	60.130001	18.34	17.12
3	1960-01-07	59.689999	59.689999	59.689999	59.689999	18.34	17.12
4	1960-01-08	59.500000	59.500000	59.500000	59.500000	18.34	17.12
...	...	...	...	...	...	...	...
15671	2022-02-11	4418.640000	4506.270000	4526.330000	4401.410000	37.56	25.93
15672	2022-02-14	4401.670000	4412.610000	4426.220000	4364.840000	37.56	25.93
15673	2022-02-15	4471.070000	4429.280000	4472.770000	4429.280000	37.56	25.93
15674	2022-02-16	4475.010000	4455.750000	4489.550000	4429.680000	37.56	25.93
15675	2022-02-17	4380.260000	4456.060000	4456.060000	4373.810000	37.56	25.93

15676 rows × 7 columns

Figure 2.1: Stock Dataset: Python code

In this assignment, we choose the stock data from 1960 to 2022, 15675 raw data records in total. Here we take the former 80% as training data which is total 12366 records, and the other 20% (= 3091 records) as the test part. In a later version, to prevent overfitting problem, another 20% data in training data set is separated out as the validation dataset in the training stage.

The column “Close”, “Open”, “High” and “Low” are taken in as a unit of input x_t at time(year) t. The images of full data and test data are listed below in the left and right.

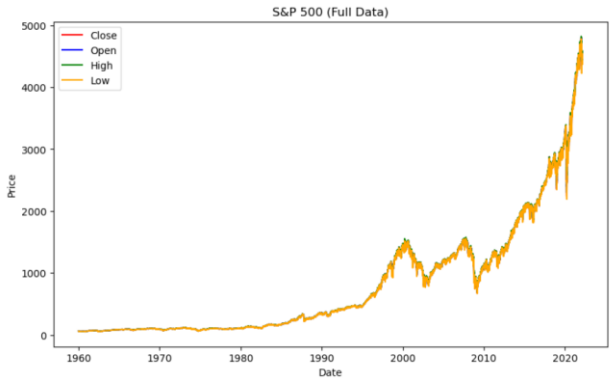


Figure 2.2: Full Data (1960-2020)

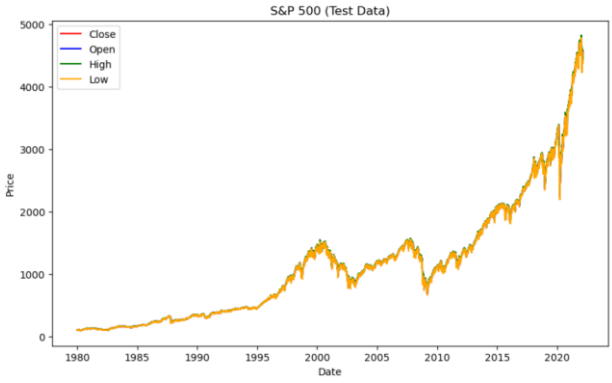


Figure 2.3: Test Data (1980-2020)

2.2 RNN

```
class RNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, num_layers):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size = input_size, hidden_size = hidden_size, num_layers = num_layers, batch_first = True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, x):
#         print(x.shape)
        batch_size = x.size(0)
        hidden = self.init_hidden(batch_size)
        res, hidden = self.rnn(x, hidden)
        res = self.linear(res)
        return res[:, -1, :], hidden

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll use in the forward pass
        # We'll send the tensor holding the hidden state to the device we specified earlier as well
        hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size)
        return hidden
```

Figure 2.4: RNN Python code

As what we have learned, RNN (recursive neural network) is fit for some video or voice sequence data related with time. It uses a consecutive series of data to predict the data of next unit. The prediction process is similar in this assignment. To implement the prediction, we take a series of first 180 days data as input set to predict the data of day 181. Then take the data from day2 to day181 to predict the data of day 182 until the end. Our sample window is half a year.

The backpropagation method is also used here to improve the performance of the neural network, though it is called BPFT and related with time. Same with the backpropagation, BPFT repeatedly use the chain rule, while the difference is BPFT has its loss function not only depending on the output layer in the current stage, but also the output in next stage. So, the weight update considers the gradient from now and later.

3 Training and Testing configuration

3.1 Training input and prediction: Noiseless

We get rid of records with 0 prices in the dataset basing on the consideration that 0 price in the dataset provide no calculating value to prediction, since there may be a holiday or similar stuff. The configurations of prediction and input [Close, Open, High, Low] are listed below.

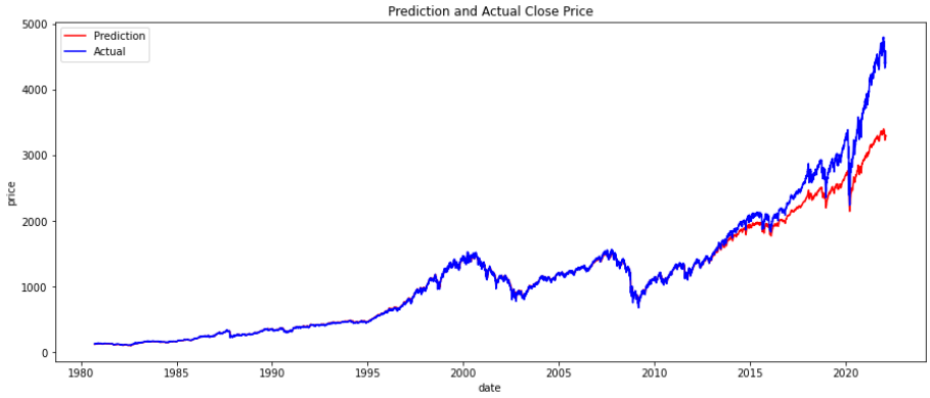


Figure 3.1: Noiseless Close Price input and prediction

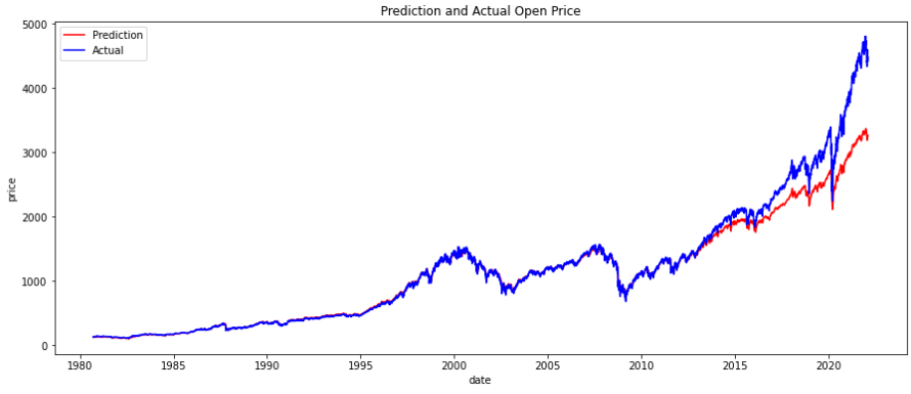


Figure 3.2: Noiseless Open Price input and prediction

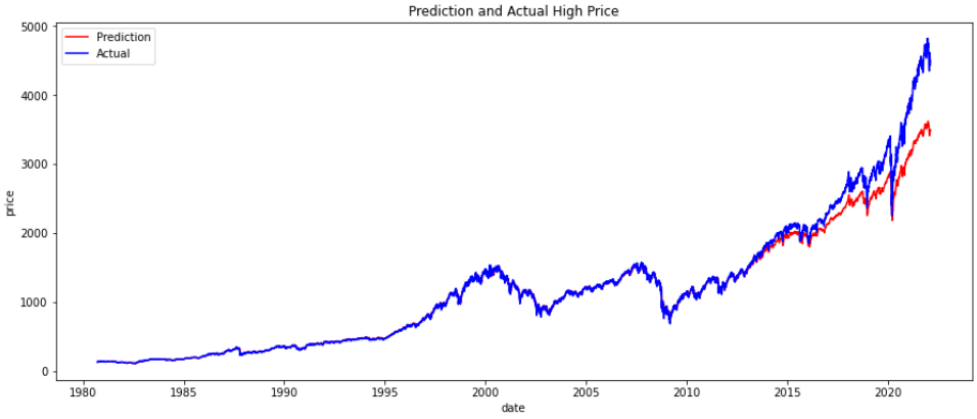


Figure 3.3: Noiseless High Price input and prediction

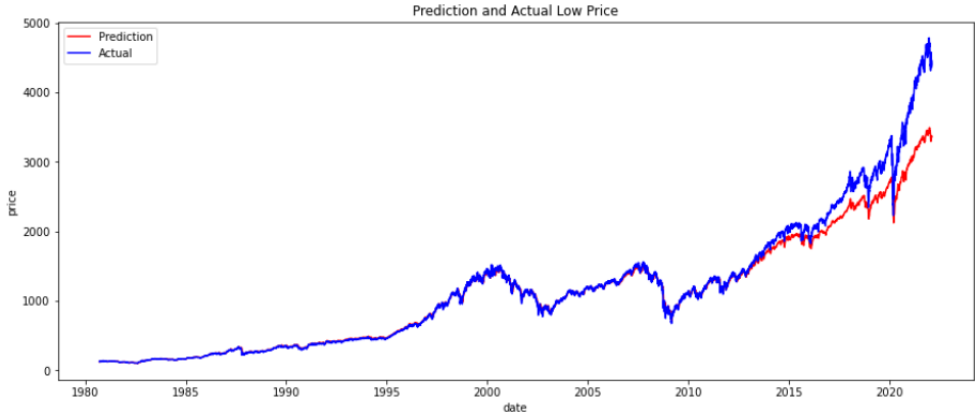


Figure 3.4: Noiseless Low Price input and prediction

Above is comparison between the real data and the predicted results from 1980 to 2020. There is a little vacuum part at the beginning as the first input serious without prediction. As the time line moves, there may be some un accuracy between the actual and prediction, which is normal. In such a training process, we use ‘real_train_list’ as the input label, and ‘label_train_list’ as the output result label.

To implement the training, a normalization step is used here. It is obvious that in a NN learning process, it is better to use normalization to make the prediction easier to converge when the gradient descent is a better choice. Also, normalization scales down the data into a smaller range to make character weights more than the distance of the data. A flow chat of training process is pasted below.

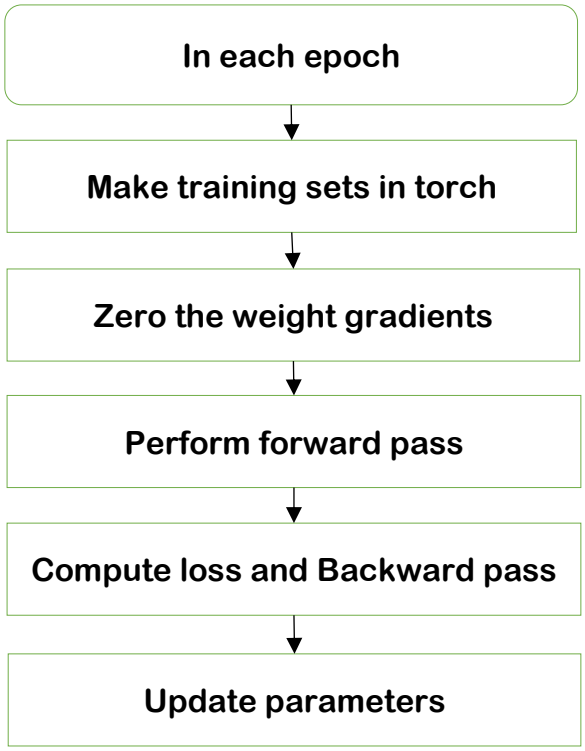


Figure 3.5: Training process

The MSE loss in the training process shows as the picture below. It can be seen that the MSE loss goes down and down rapidly at first and even near 0 when the training epoch goes larger. In our experiments, the loss reaches 0 at the end.

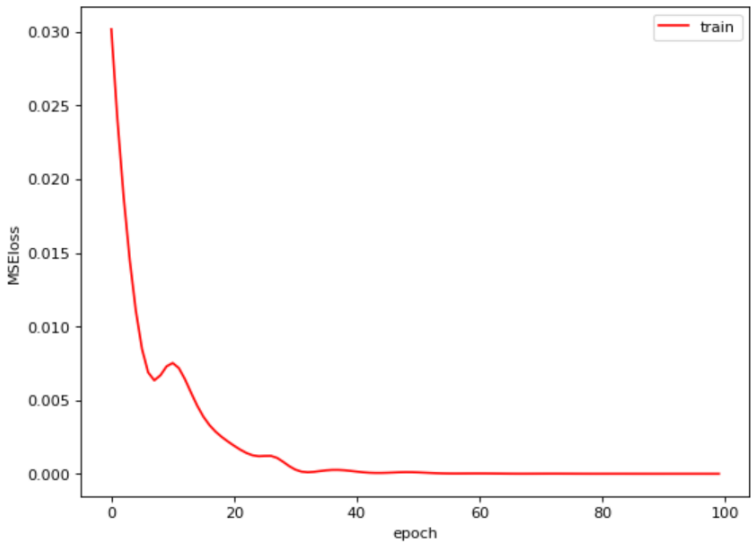


Figure 3.6: MSE Loss

Here as we test the performance of our RNN network accuracy by prediction error, which calculation is given in the assignment description:

$$predictio\ error = \frac{predicted\ price - actual\ price}{actual\_price}$$

A statistic image contains errors of all four columns of [Close, Open, High, Low] is generated by us to give a direct error percentage picture. It can be seen that as the year grows, the error percentage is higher and higher after 2010.

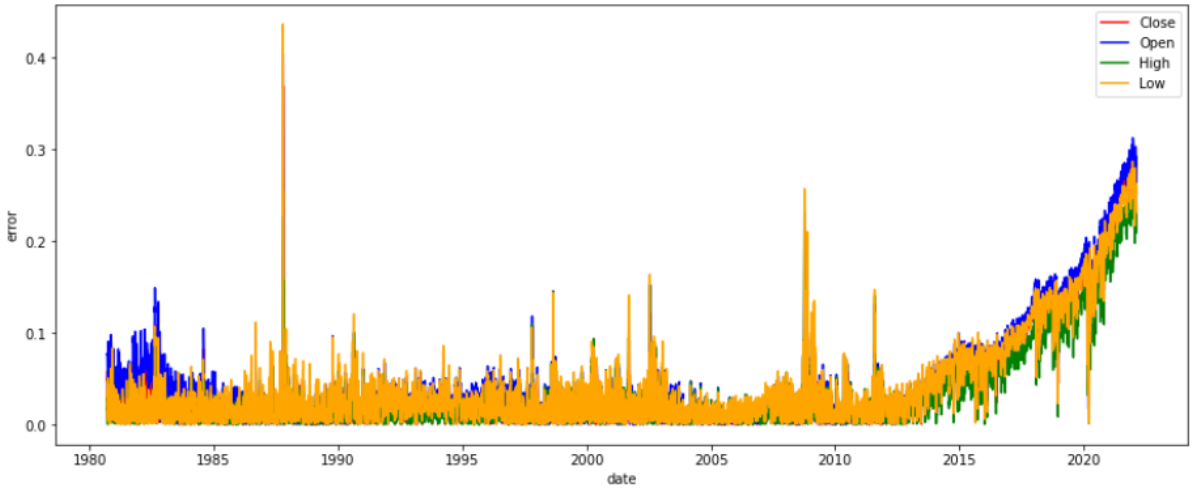


Figure 3.7: Error Percentage

3.2 With Validation set: Noiseless

As the overfitting problem is mentioned in 3.1, we separate 20% data from the training dataset as the validation dataset. A validation set is used to give an unbiased estimate of model skill when tuning model's hyperparameters. The validation set is stored in #real_val_list, #label_val_list and #date_val_list. After the separation, training set has 9893 records left and validation set has 2473 records.

The estimate process with validation set is added into the training process before updating parameters. It can be seen that as the training epoch grows, the loss of training set and validation set are both going down. At the end, they all reach 0. But it is also obvious that the validation loss goes down more slowly, which is reasonable and understandable for such a model.

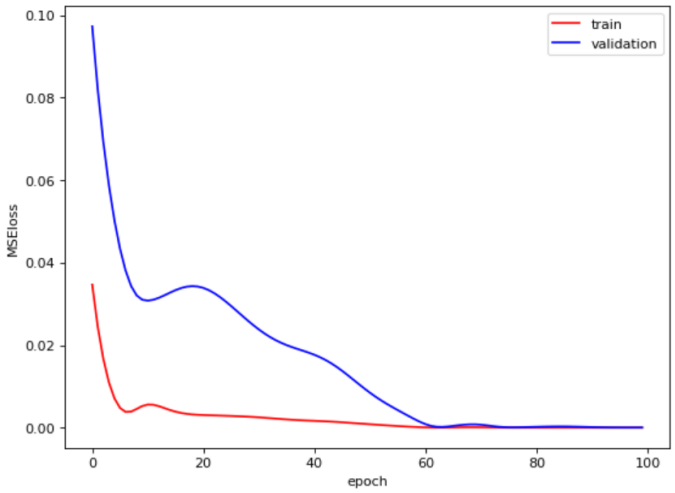


Figure 3.8: Loss of Train and Validation Set

And with the validation dataset added in, the results of our experiment with the total error percentage showing are shown below. The errors are lower than those without validation set a little.

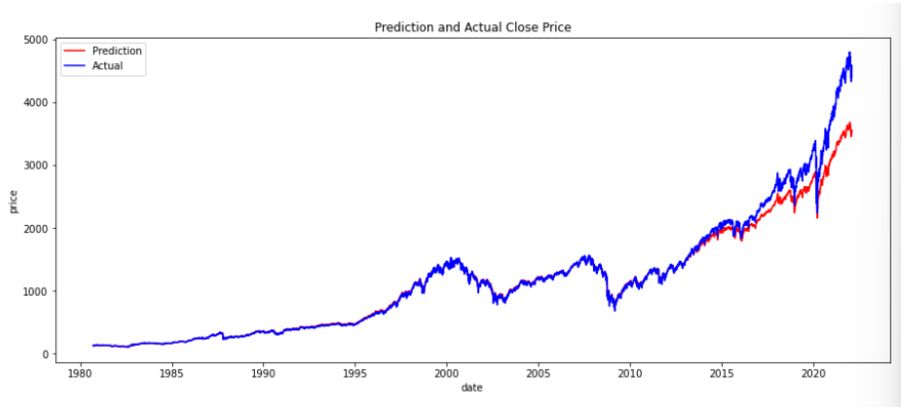


Figure 3.9: Noiseless Close Price with Validation Set

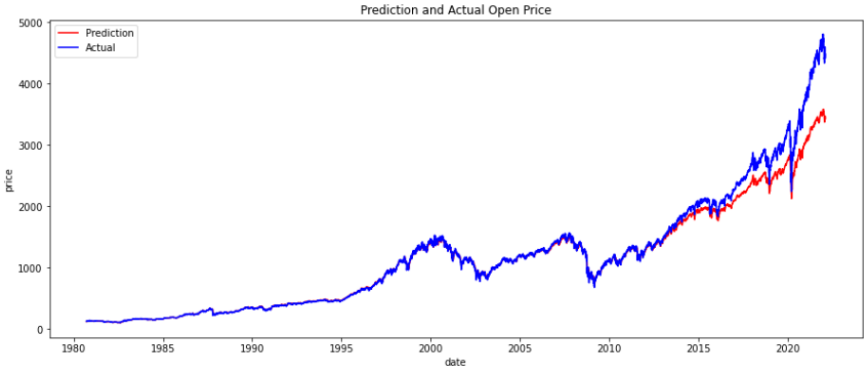


Figure 3.10: Noiseless Open Price with Validation Set

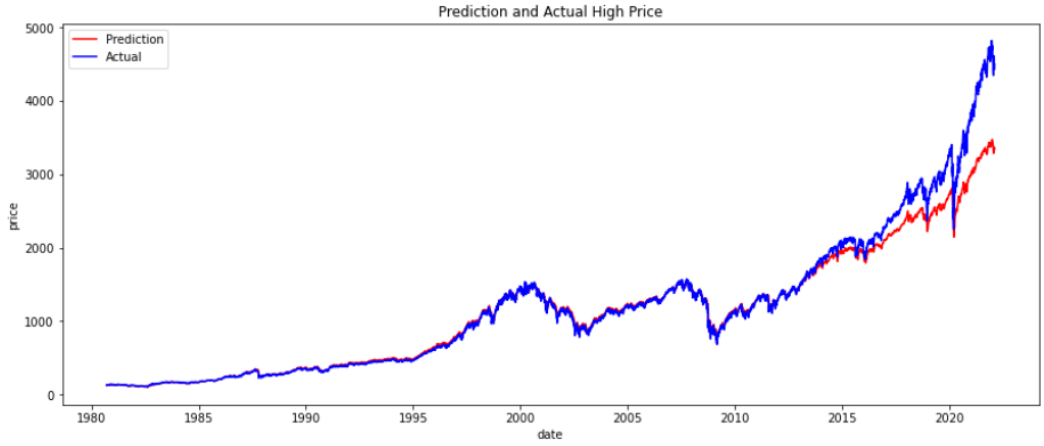


Figure 3.11: Noiseless High Price with Validation Set

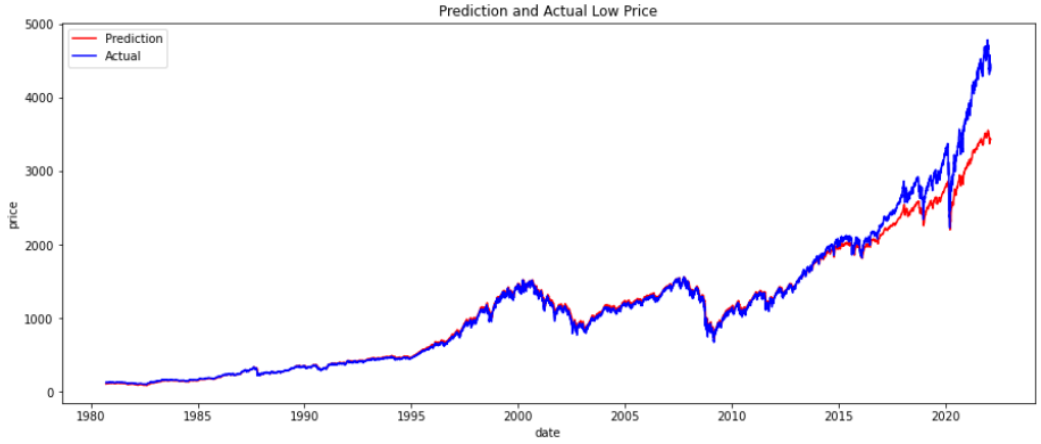


Figure 3.12: Noiseless Low Price with Validation Set

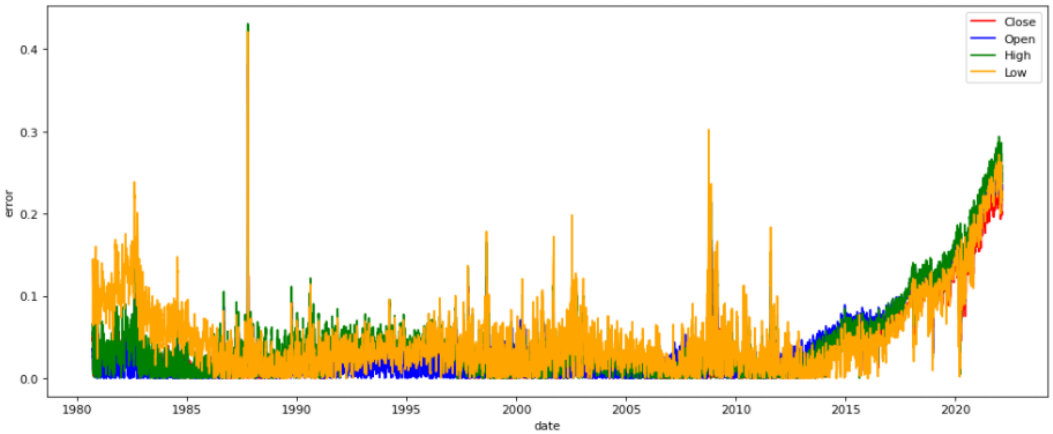


Figure 3.13: Noiseless Prediction Error

3.3 Training data: Noise-corrupted

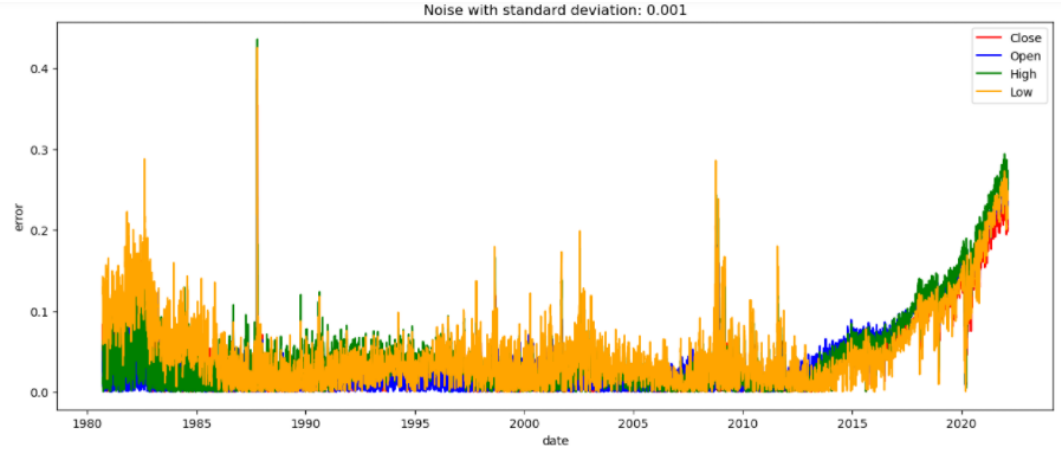


Figure 3.14: Noise-corrupted, SD 0.001

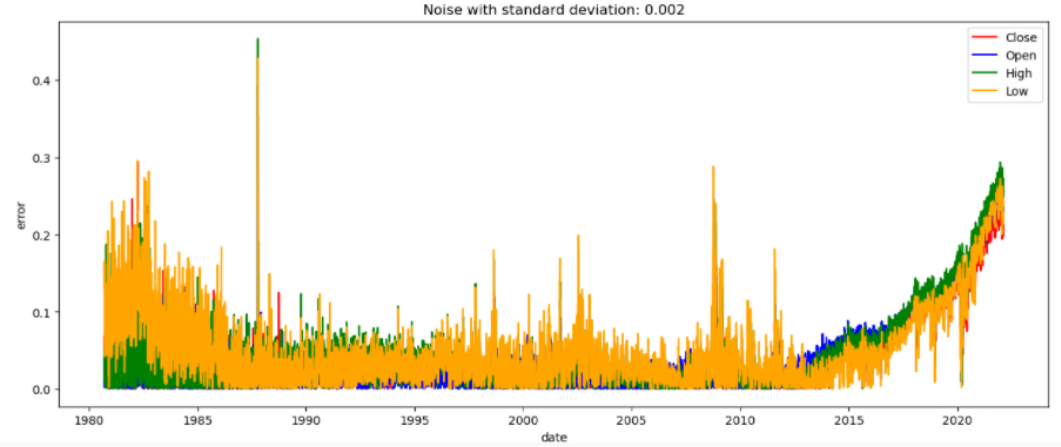


Figure 3.15: Noise-corrupted, SD 0.002

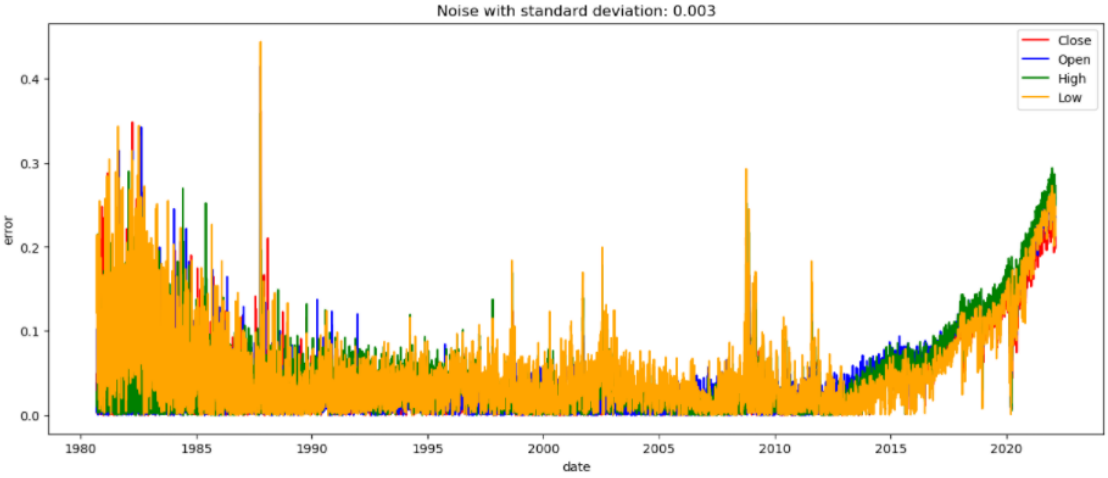


Figure 3.16: Noise-corrupted, SD 0.003

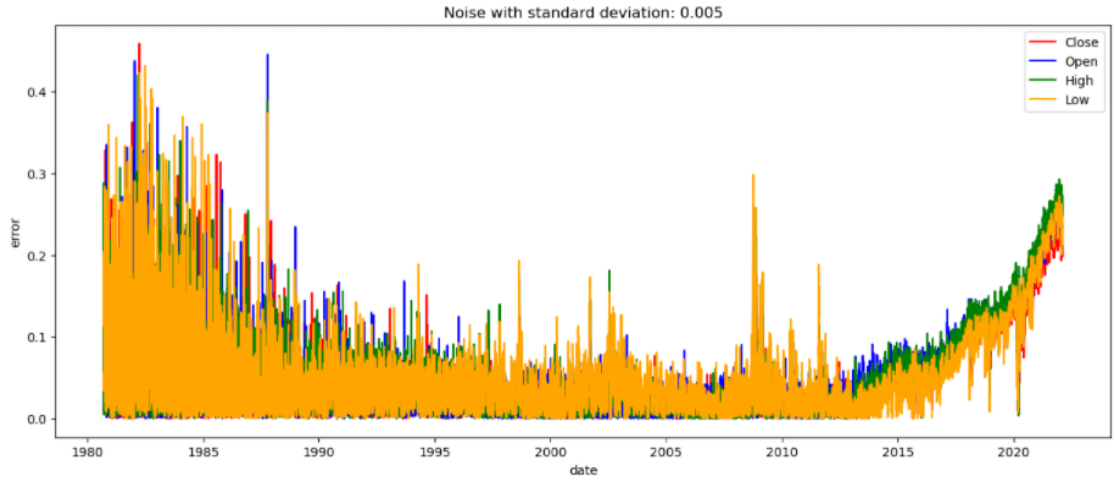


Figure 3.17: Noise-corrupted, SD 0.005

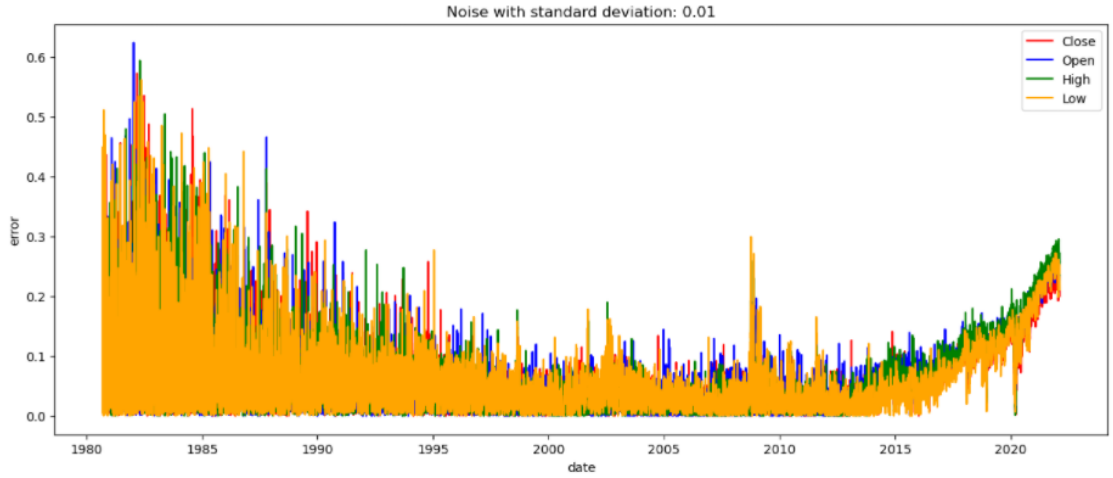


Figure 3.18: Noise-corrupted, SD 0.01

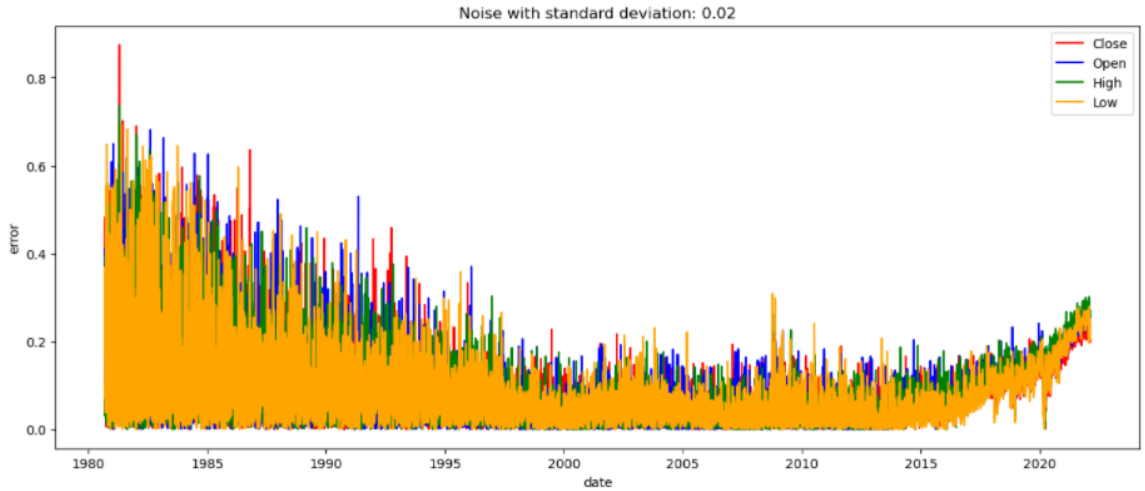


Figure 3.19: Noise-corrupted, SD 0.02

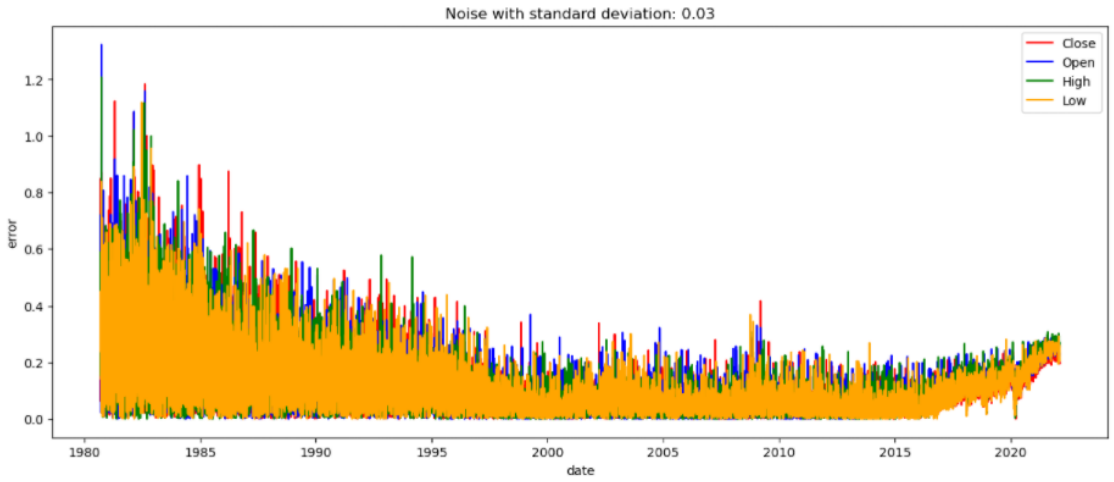


Figure 3.20: Noise-corrupted, SD 0.03

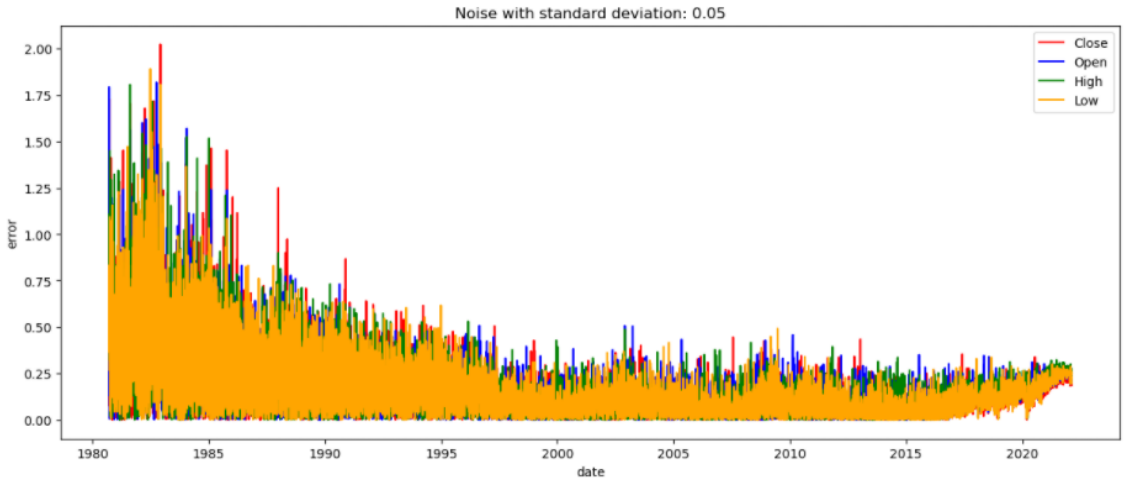


Figure 3.21: Noise-corrupted, SD 0.05

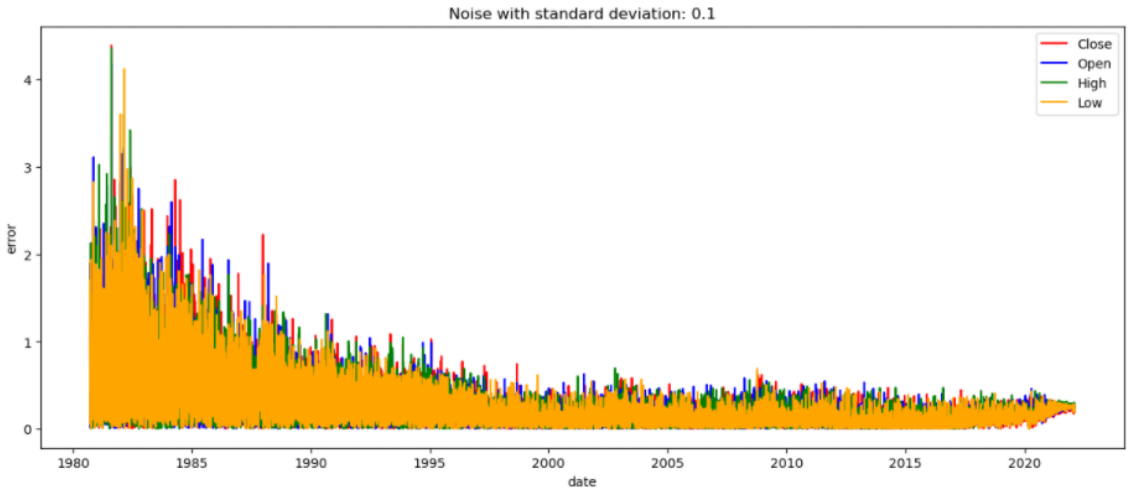


Figure 3.22: Noise-corrupted, SD 0.1

In the noise-corrupted process, we add the Gaussian Noise with the Standard deviation of 0.001, 0.002, 0.003, 0.005, 0.01, 0.02, 0.03, 0.05, and 0.1. As a result, you would see the error performance go higher as the standard deviation going up. When the standard deviation reaches 0.1, the error is obvious and large.

4 Unoptimized RNN Output Results VS. Optimized RNN Results

Previously, we set the hyperparameter hidden_size=16, we get the following result.

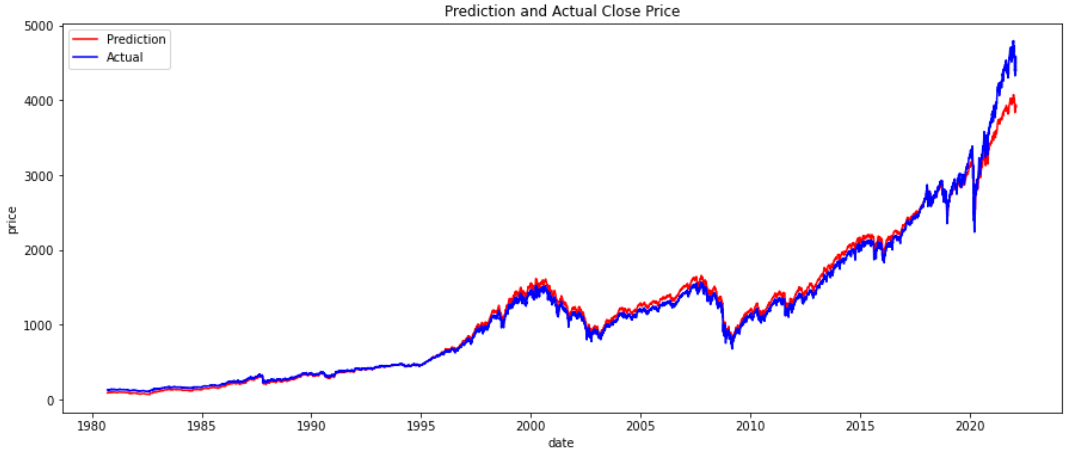


Figure 4.1: hidden_size = 16, Close Price

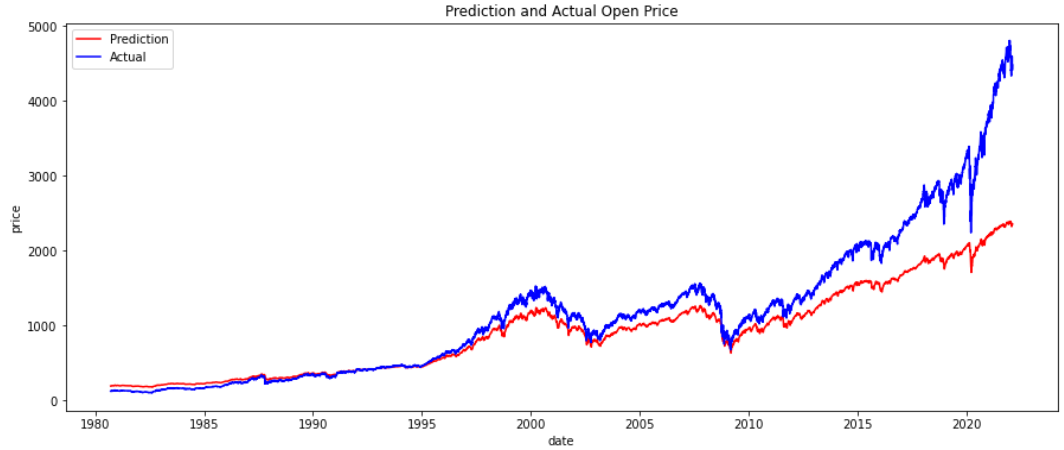


Figure 4.2: hidden_size = 16, Open Price

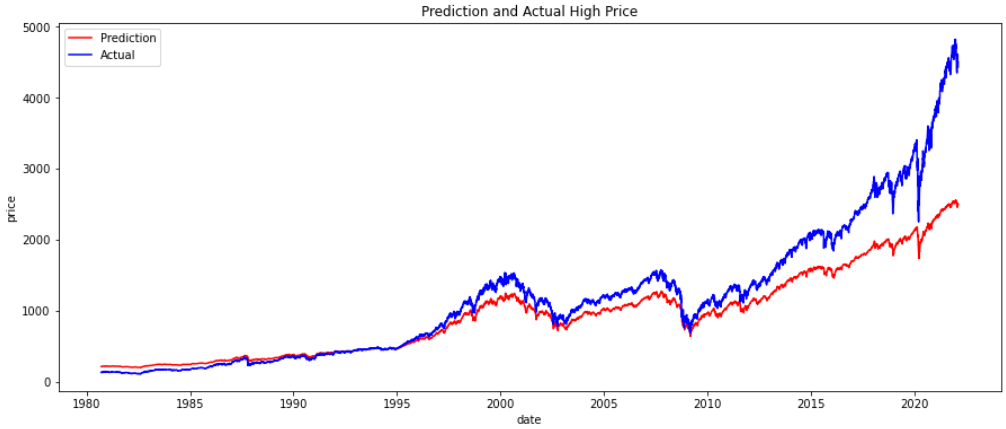


Figure 4.3: hidden_size = 16, High Price

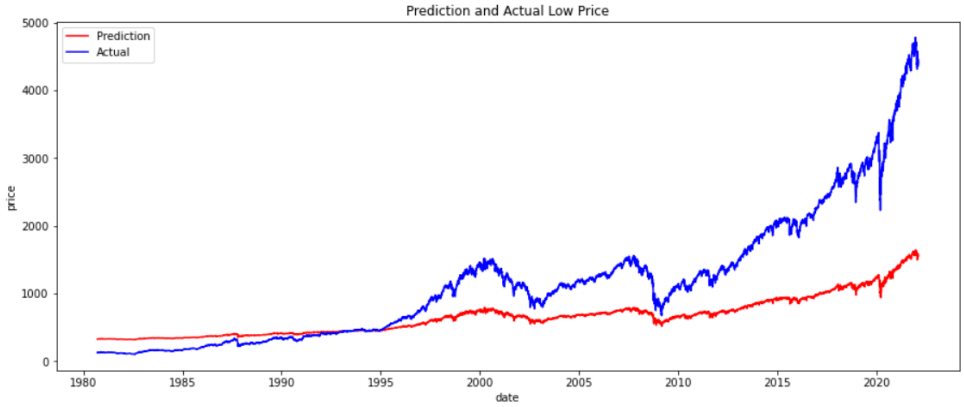


Figure 4.4: hidden_size = 16, Low Price

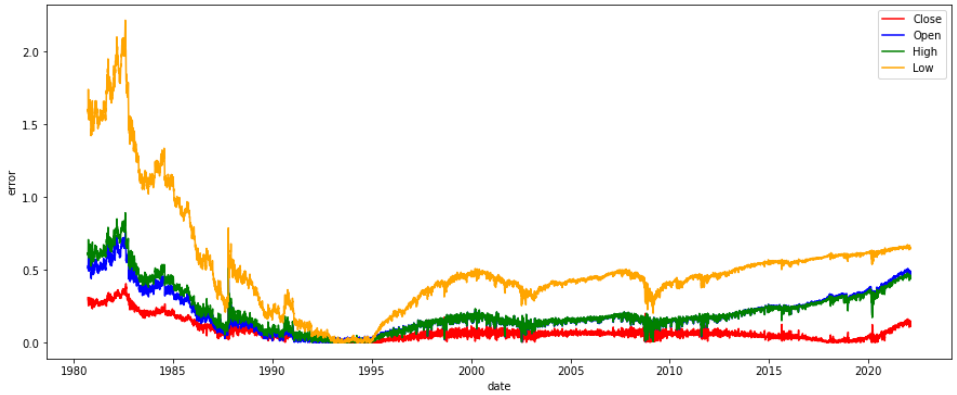


Figure 4.5: hidden_size = 16, Prediction Error

According to the result our model's prediction goes up when the real data goes up, but it doesn't catch up the increment of the real data (except close dataset). So, we change the hidden_size to 60. And the results are shown as follow:

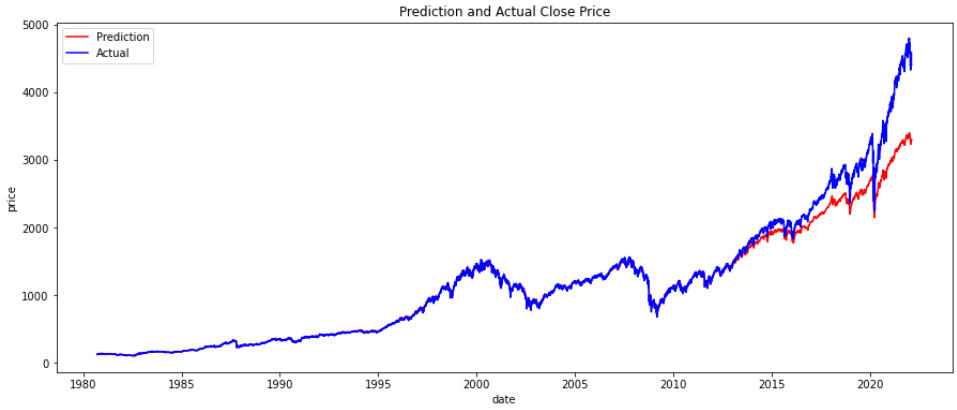


Figure 4.6: hidden_size = 60, Close Price

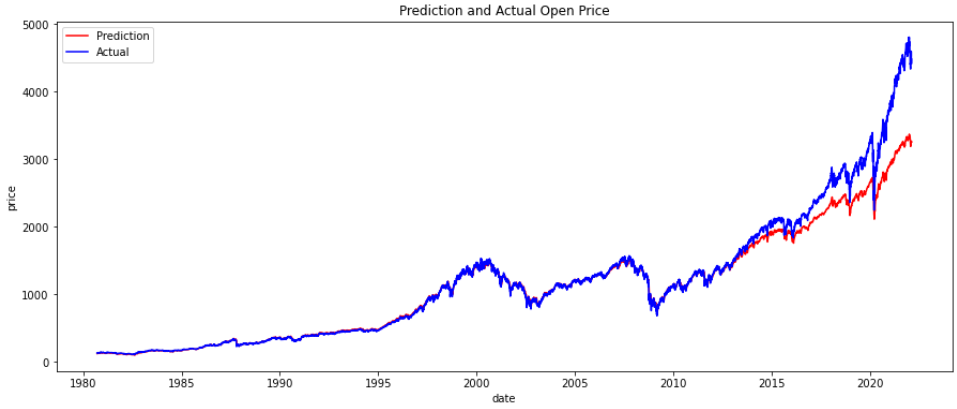


Figure 4.7: hidden_size = 60, Open Price

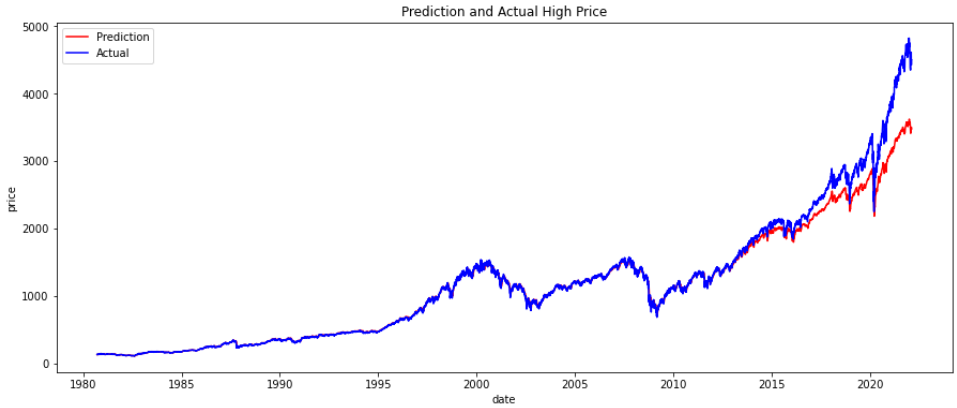


Figure 4.8: hidden_size = 60, High Price

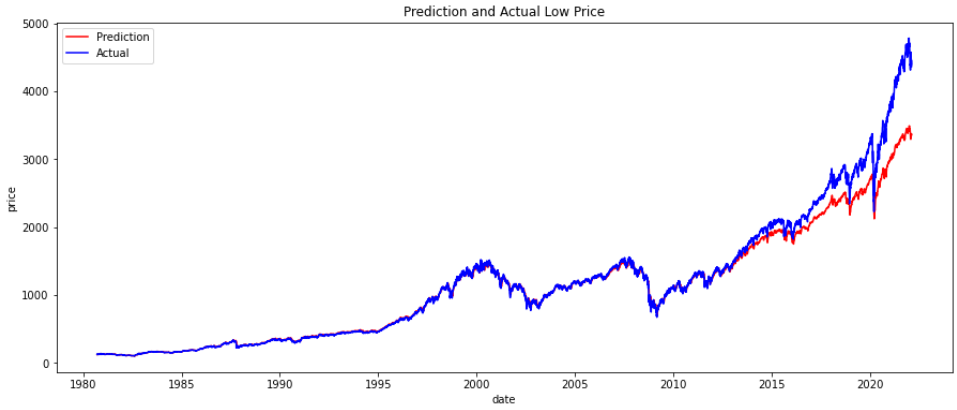


Figure 4.9: hidden_size = 60, Low Price

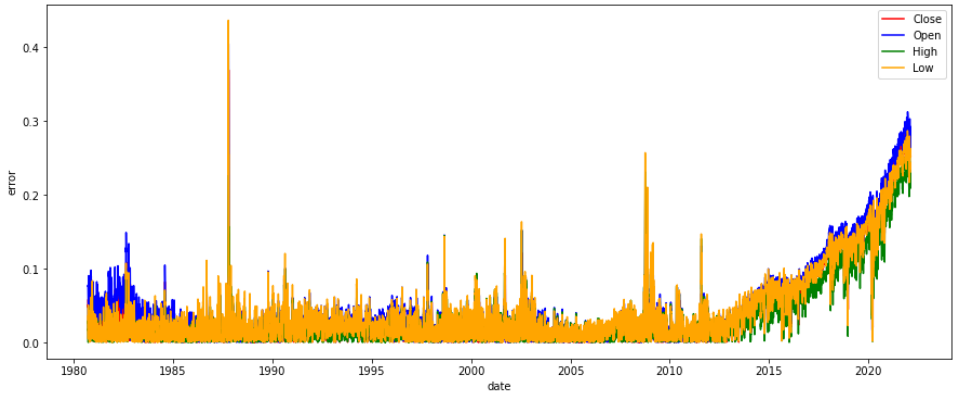


Figure 4.10: hidden_size = 60, Prediction Error

When hidden_size = 60, the prediction is way better than the previous result.

5 Discussion

Q: Discussion (in detail) of why your RNNs performed the way they did, and how you could improve their performance in future.

A: Previously, we set the hyperparameter hidden_size=16, we get the following result.

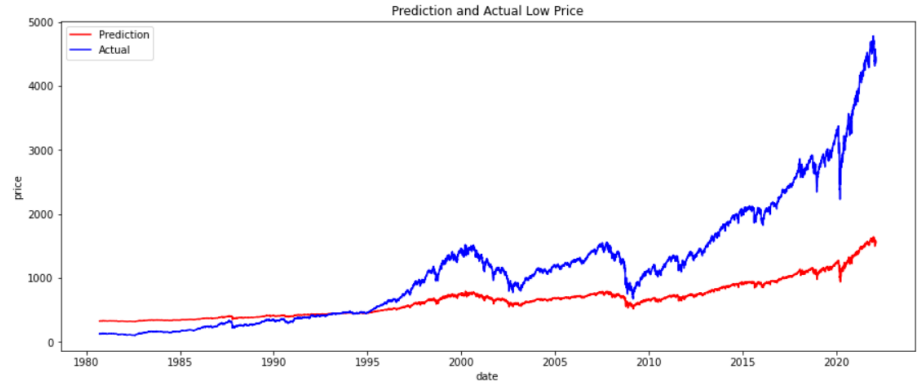


Figure 5.1: hidden_size = 16

According to the result, our model's prediction goes up when the real data goes up, but it doesn't catch up the increment of the real data. So, we change the hidden_size to 60. And the result is shown as follow

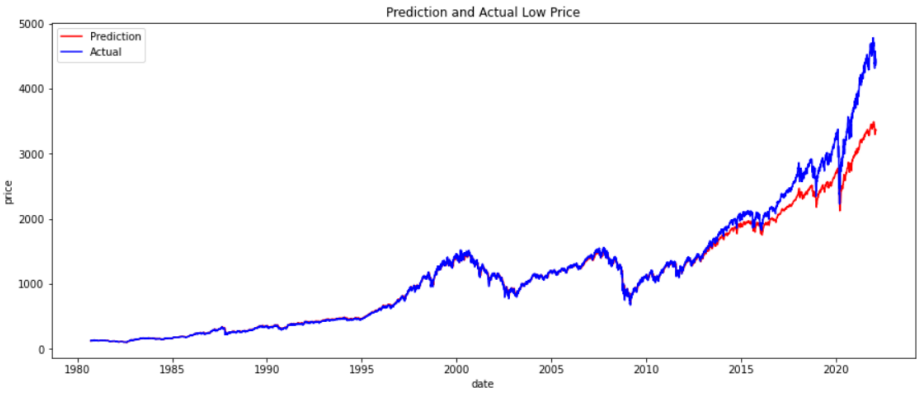


Figure 5.2: hidden_size = 60

The prediction is way better than the previous result.

The reason our RNNs performed this way is because hidden_size Defines the size of the hidden state. Therefore, if hidden_size is set as 4, then the hidden state at each time step is a vector of length 4. Base on the data we have, the more the hidden_size is, the more complicated patterns our RNN can recognize (it ended up with lower bias).

In the future, we will tune other hyperparameters (num_layers, learning_rate, num_epochs) to improve the performance. Also, if the hidden_size amount is to large it could cause overfitting issue, so we will also take that into account.

6 Extra Credit

6.1 Add in PE and Ratio

```
In [3]: data = pd.read_csv('data.csv')
        data['Date'] = pd.to_datetime(data['Date'], format='%Y-%m-%d')
        data
```

Out[3]:

	Date	Close	Open	High	Low	PE	Ratio
0	1960-01-04	59.910000	59.910000	59.910000	59.910000	18.34	17.12
1	1960-01-05	60.389999	60.389999	60.389999	60.389999	18.34	17.12
2	1960-01-06	60.130001	60.130001	60.130001	60.130001	18.34	17.12
3	1960-01-07	59.689999	59.689999	59.689999	59.689999	18.34	17.12
4	1960-01-08	59.500000	59.500000	59.500000	59.500000	18.34	17.12
...	...	...	...	...	...	...	...
15671	2022-02-11	4418.640000	4506.270000	4526.330000	4401.410000	37.56	25.93
15672	2022-02-14	4401.670000	4412.610000	4426.220000	4364.840000	37.56	25.93
15673	2022-02-15	4471.070000	4429.280000	4472.770000	4429.280000	37.56	25.93
15674	2022-02-16	4475.010000	4455.750000	4489.550000	4429.680000	37.56	25.93
15675	2022-02-17	4380.260000	4456.060000	4456.060000	4373.810000	37.56	25.93

15676 rows × 7 columns

*Figure 6.1: Data with 0 (10576 * 7)*

```
In [30]: rows = [x for x in data.index if data.loc[x]['Open'] == 0]
         data = data.drop(rows, axis=0)
         data
```

Out[30]:

	Date	Close	Open	High	Low	PE	Ratio
0	1960-01-04	59.910000	59.910000	59.910000	59.910000	18.34	17.12
1	1960-01-05	60.389999	60.389999	60.389999	60.389999	18.34	17.12
2	1960-01-06	60.130001	60.130001	60.130001	60.130001	18.34	17.12
3	1960-01-07	59.689999	59.689999	59.689999	59.689999	18.34	17.12
4	1960-01-08	59.500000	59.500000	59.500000	59.500000	18.34	17.12
...	...	...	...	...	...	...	...
15671	2022-02-11	4418.640000	4506.270000	4526.330000	4401.410000	37.56	25.93
15672	2022-02-14	4401.670000	4412.610000	4426.220000	4364.840000	37.56	25.93
15673	2022-02-15	4471.070000	4429.280000	4472.770000	4429.280000	37.56	25.93
15674	2022-02-16	4475.010000	4455.750000	4489.550000	4429.680000	37.56	25.93
15675	2022-02-17	4380.260000	4456.060000	4456.060000	4373.810000	37.56	25.93

15640 rows × 7 columns

*Figure 6.2: Data without 0 (15640 * 7)*

Here to join two datasets, we add in P/E and Ratio column to [Close, Open, High, Low] set as a joined input. The prediction results perform better than before.

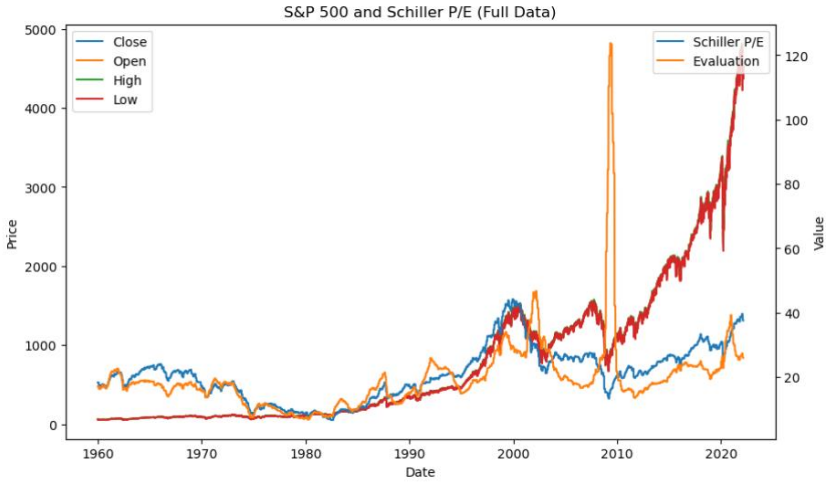


Figure 6.3: Full Data

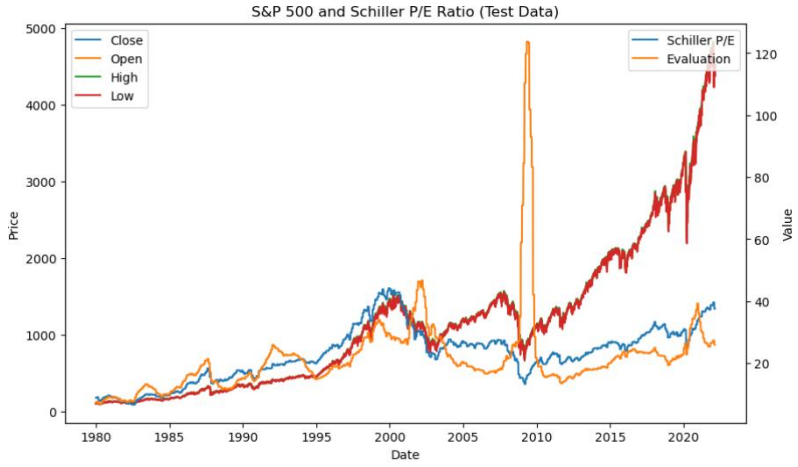


Figure 6.4: Test Data

6.2 Dataset Configuration with Prediction Comparison

Without PE and Ratio

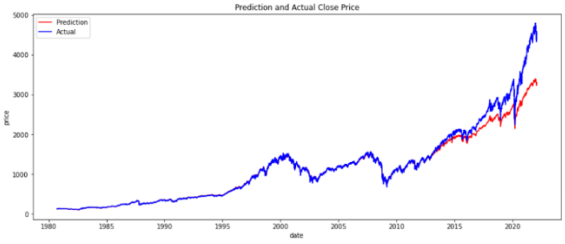


Figure 3.1: Noiseless Close Price input and prediction

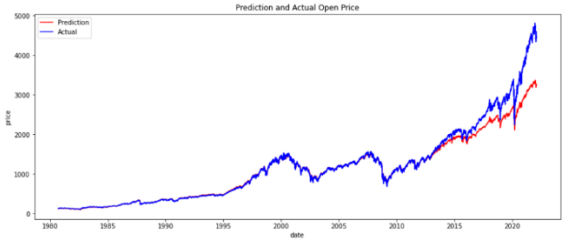


Figure 3.2: Noiseless Open Price input and prediction

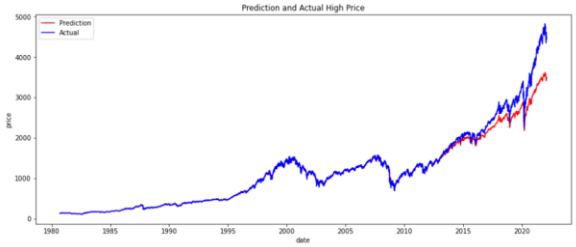


Figure 3.3: Noiseless High Price input and prediction

With PE and Ratio

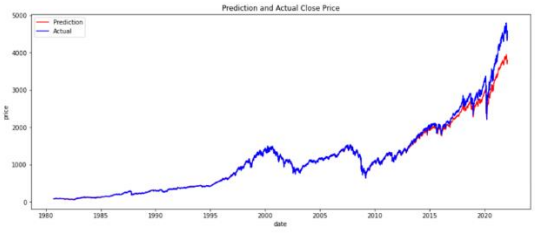


Figure 4.4: Noiseless Close Price input and prediction

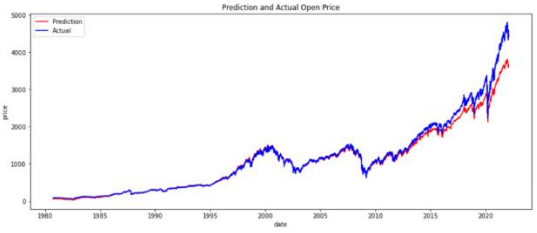


Figure 4.5: Noiseless Open Price input and prediction

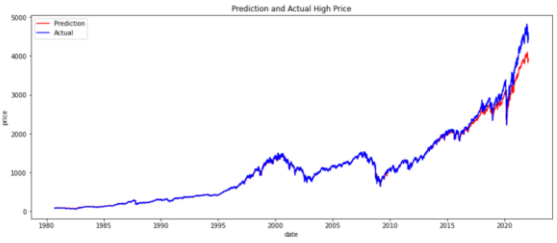


Figure 4.6: Noiseless High Price input and prediction

Without PE and Ratio **With PE and Ratio**

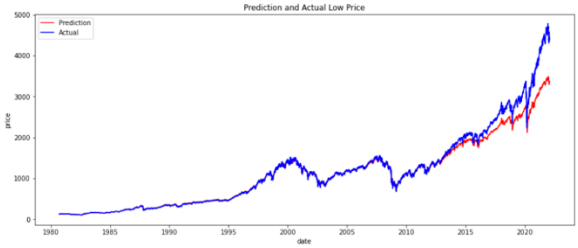


Figure 3.4: Noiseless Low Price input and prediction

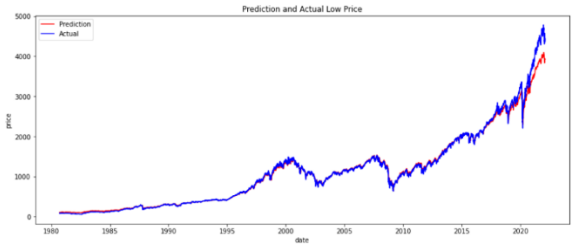
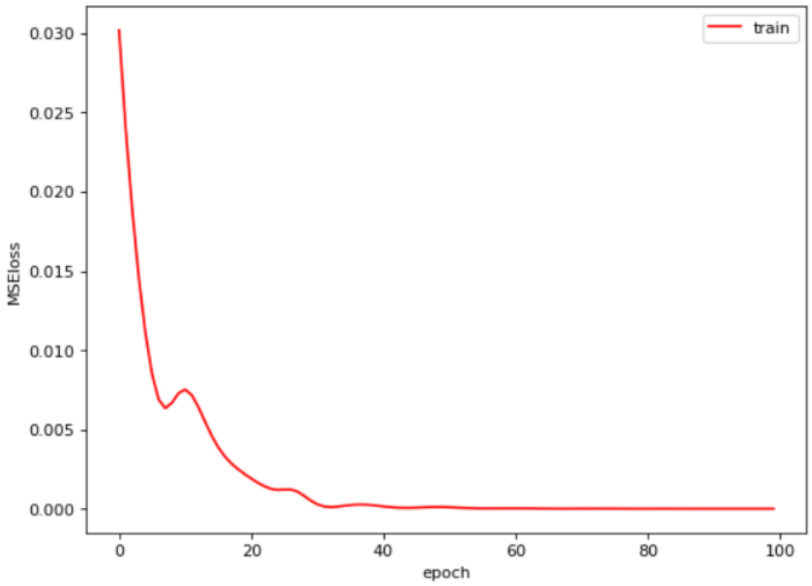


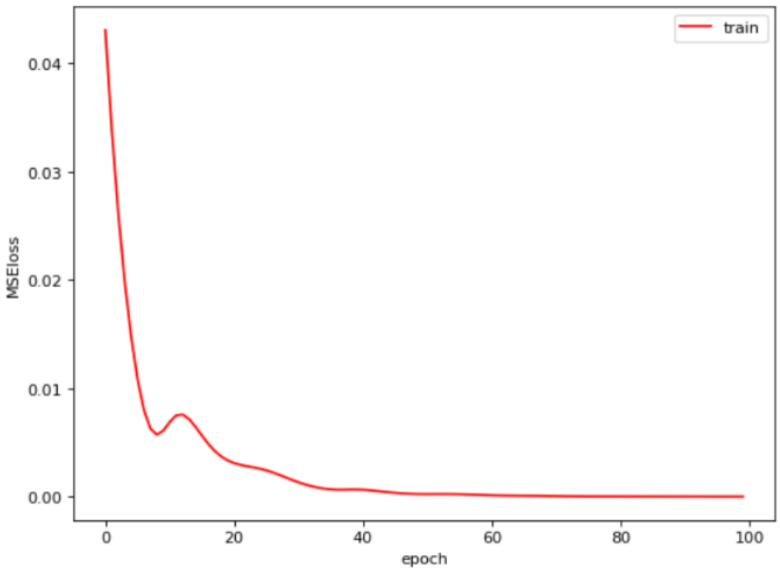
Figure 4.7: Noiseless Low Price input and prediction

After we add in the PE and Ratio, the accuracy of the prediction model improves really a lot. Comparing to four columns before, the predicting line becomes fit, and be nearer to the original price line.

Besides, The MSE loss in the training process shows that after data joining (added PE and ratio), the MSE loss gets a little higher than before at the beginning, whereas it still reaches 0 in the end.



Without PE and Ratio



With PE and Ratio

Figure 6.5: MSE Loss before and after Data optimizing

The prediction error here shows a much better performance as what the training and prediction data configurations did. Though the errors are higher than before in 1980-1987, they go down later and primarily always lower than before.

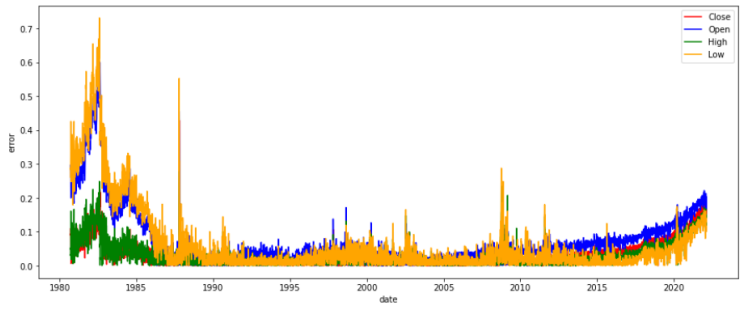


Figure 6.6: Error Percentage without adding validation dataset (with PE and Ratio)

6.3 With Validation Dataset: Noiseless

MSE loss of validation dataset gives a more rapid convergence when descending between epoch 0 to epoch 60. Training dataset has no obvious difference in descending performance.

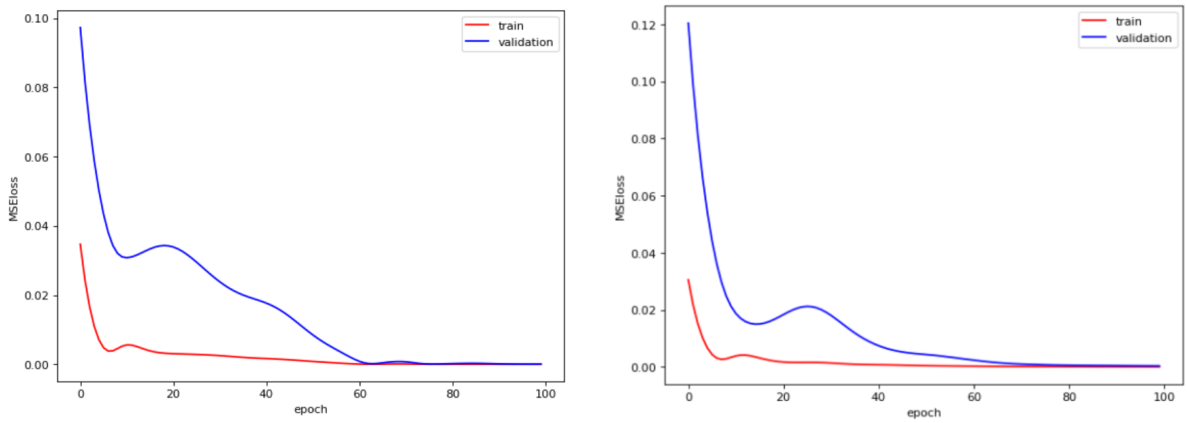


Figure 6.7: MSE Loss before and after

The configuration shows that the Close price gets a more accurate prediction, while the other three gets a little worse than before.

Without PE and Ratio

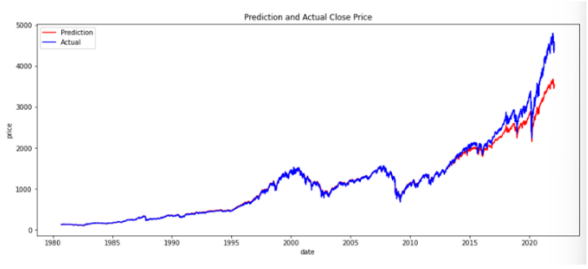


Figure 3.9: Noiseless Close Price with Validation Set

Figure 3.10: Noiseless Open Price with Validation Set

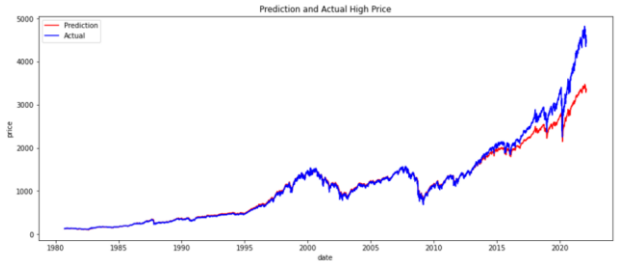


Figure 3.11: Noiseless High Price with Validation Set

With PE and Ratio

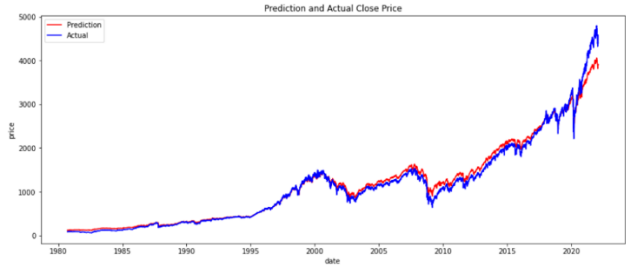


Figure 4.11: Noiseless Close Price with Validation Set

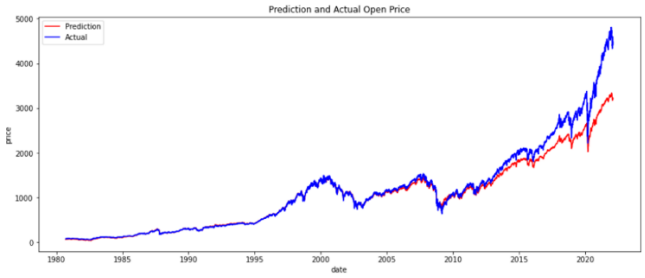


Figure 4.12: Noiseless Open Price with Validation Set

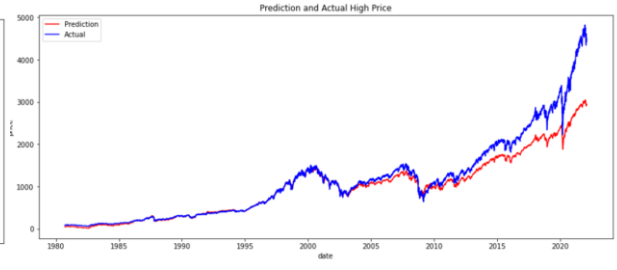


Figure 4.13: Noiseless High Price with Validation Set

Without PE and Ratio **With PE and Ratio**

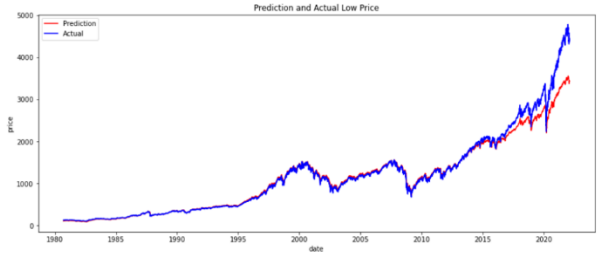


Figure 3.12: Noiseless Low Price with Validation Set

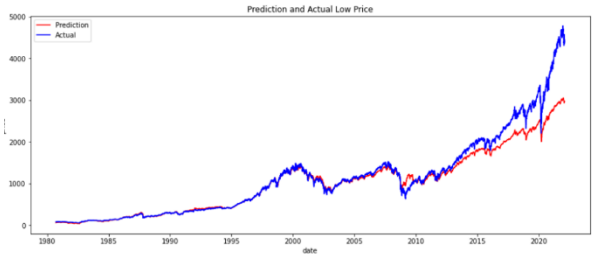


Figure 4.14: Noiseless Low Price with Validation Set

The prediction (with PE and Ratio) errors vary apparently in [Close, Open, High, Low] prices, and even be much higher than before in most time slots. Especially at the beginning of time series like 1980s, the Close price even gets the error near 1.0. The possibly reason of such a reversely phenomenon may be first limited dataset samples. Second the Model might be overfitted.

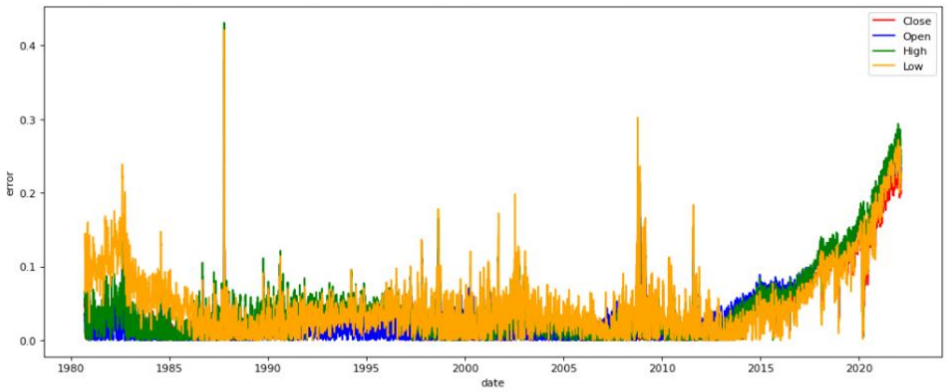


Figure 3.13: Noiseless Prediction Error

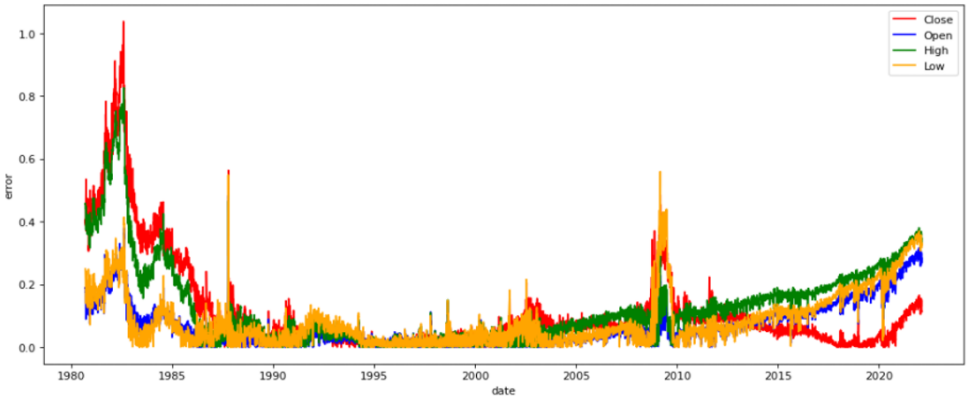


Figure 4.15: Noiseless Prediction Error

6.4 Noise-corrupted

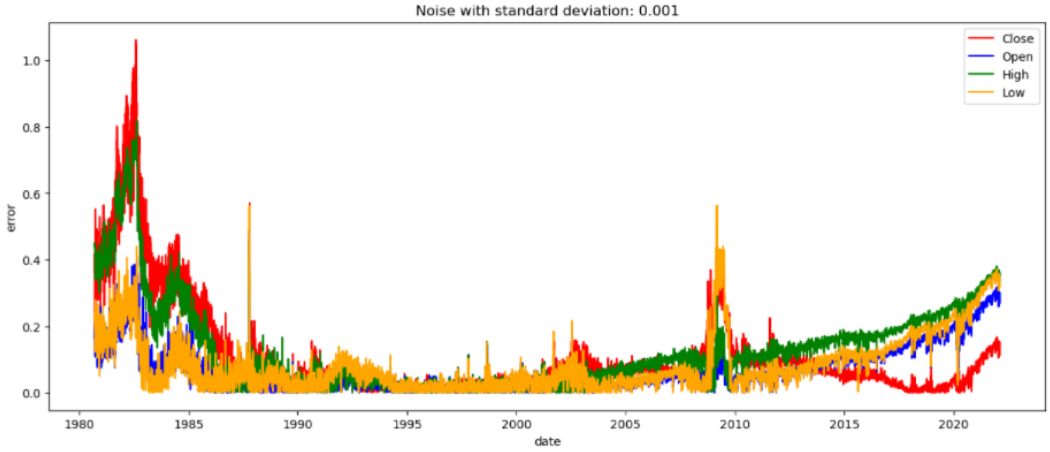


Figure 6.8: Noise-corrupted, SD 0.001

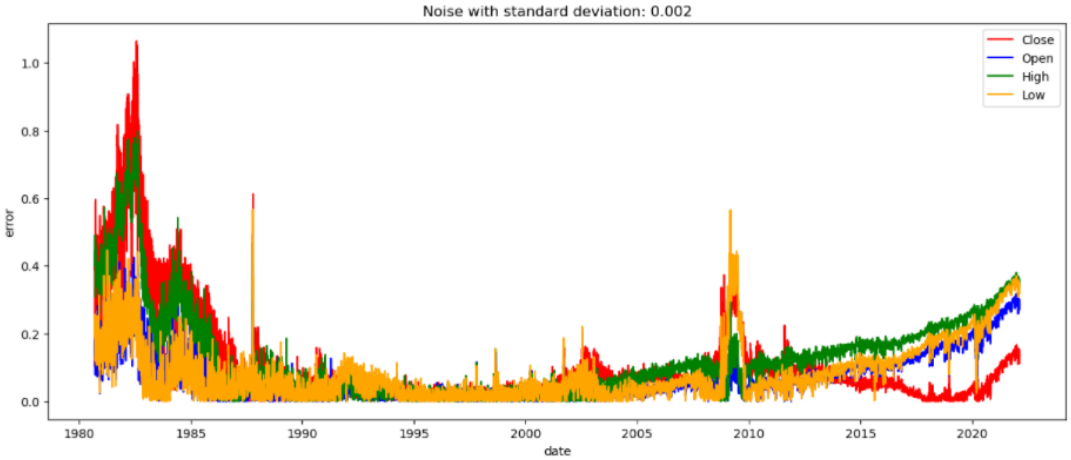


Figure 6.9: Noise-corrupted, SD 0.002

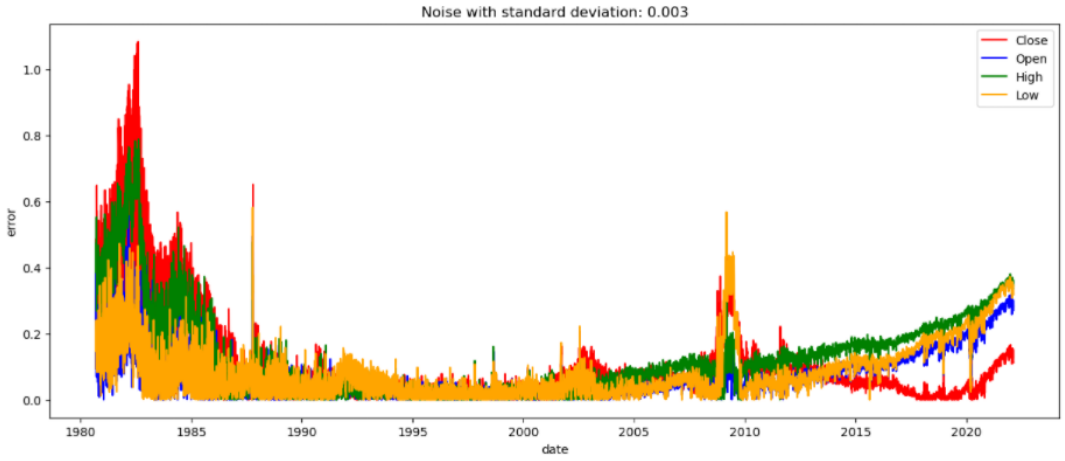


Figure 6.10: Noise-corrupted, SD 0.003

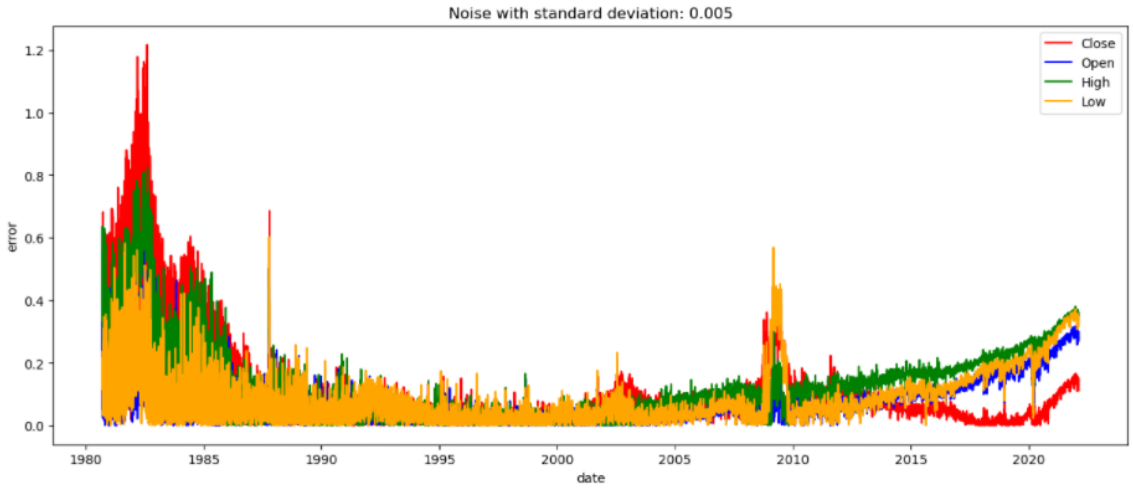


Figure 6.11: Noise-corrupted, SD 0.005

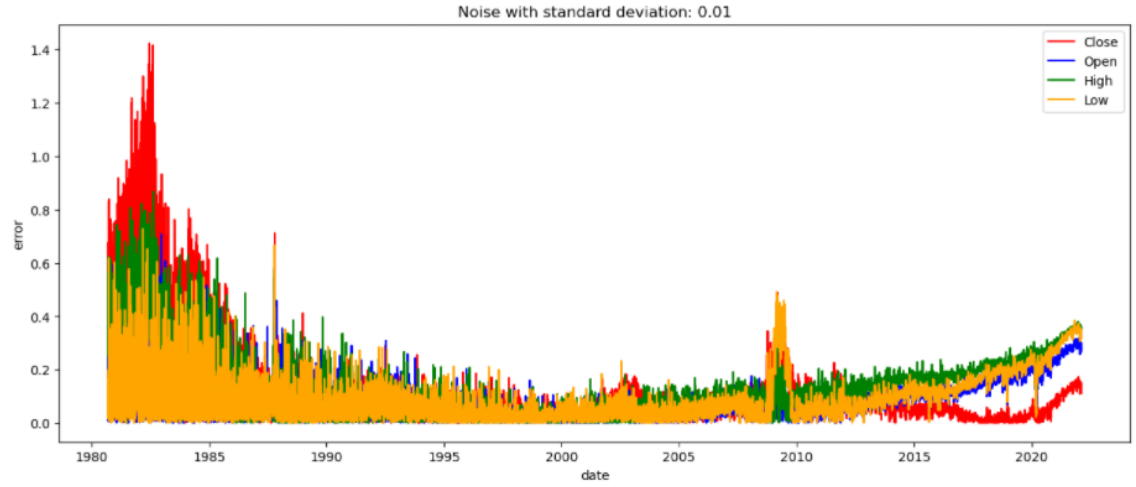


Figure 6.12: Noise-corrupted, SD 0.01

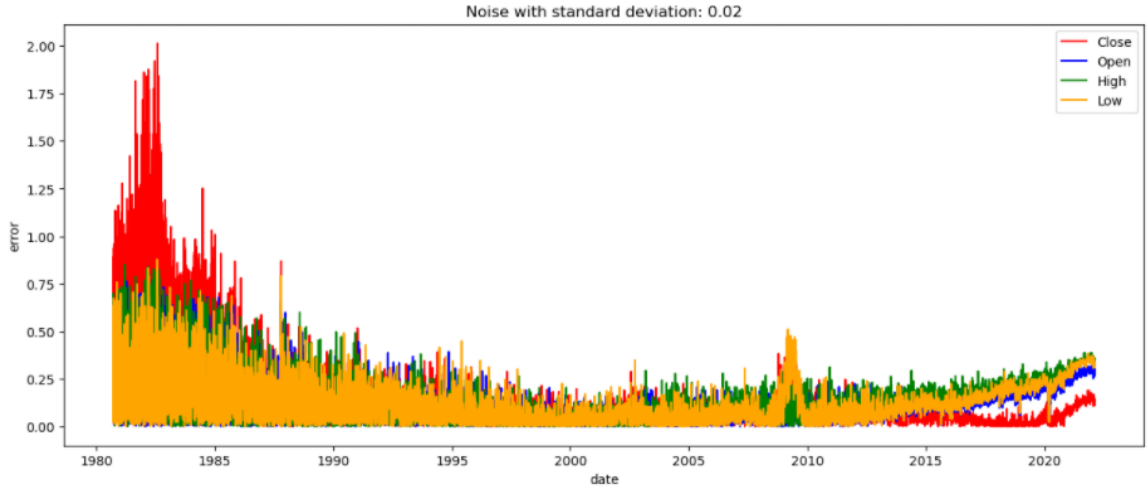


Figure 6.13: Noise-corrupted, SD 0.02

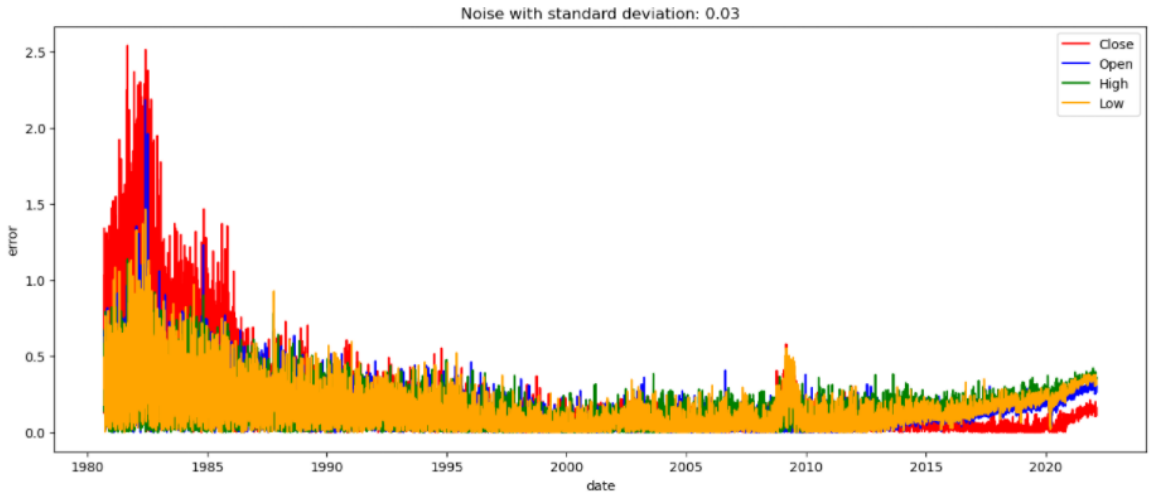


Figure 6.14: Noise-corrupted, SD 0.03

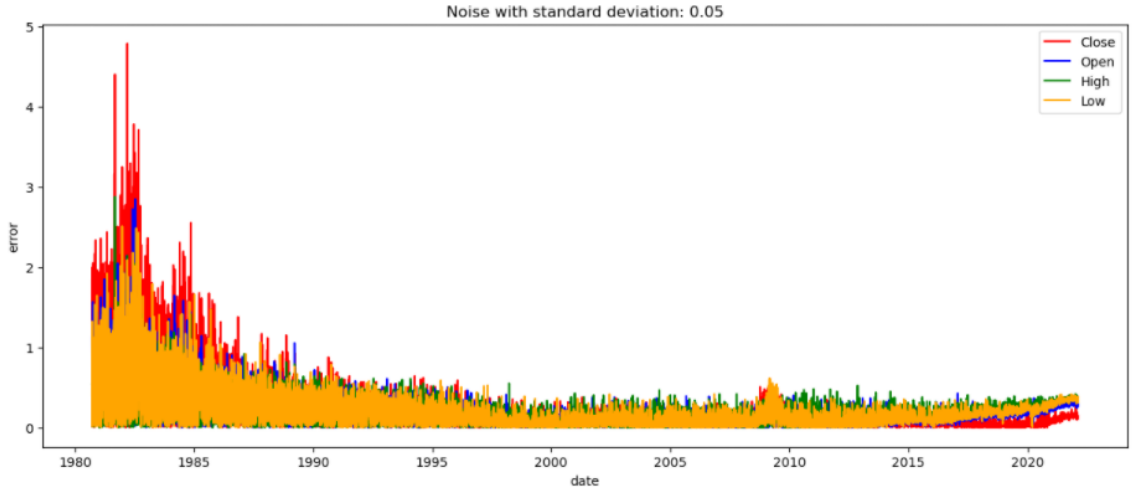


Figure 6.15: Noise-corrupted, SD 0.05

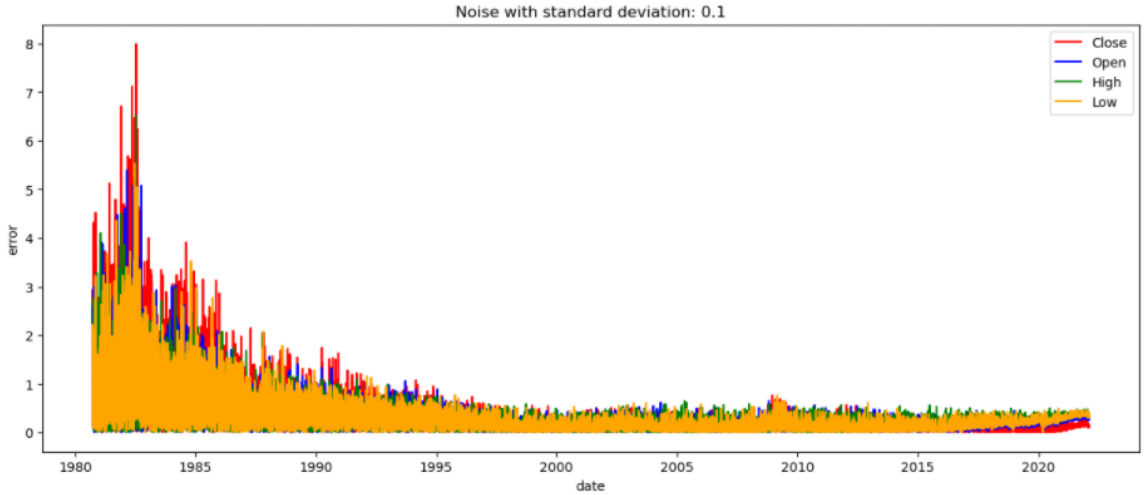


Figure 6.16: Noise-corrupted, SD 0.1

So, the solution for limited dataset samples is to collect for data for training. And the solution for overfitted model is to regularize the model (decrease model complexity or decrease the validation dataset size); or to improve MSE we may use less inputs (only those which are more relevant or statistically significant).