CAP 6615 Neural Network

University of Florida

Cap 6615

**Programming Assignment 3 -- Recurrent Neural Network**

**Group Member:**

Fugang Deng

Wei He

Hang Ye

Zihang Huang

Jianan He

Weijia Sun

**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

Python code for RNN

2.1 S&P Dataset

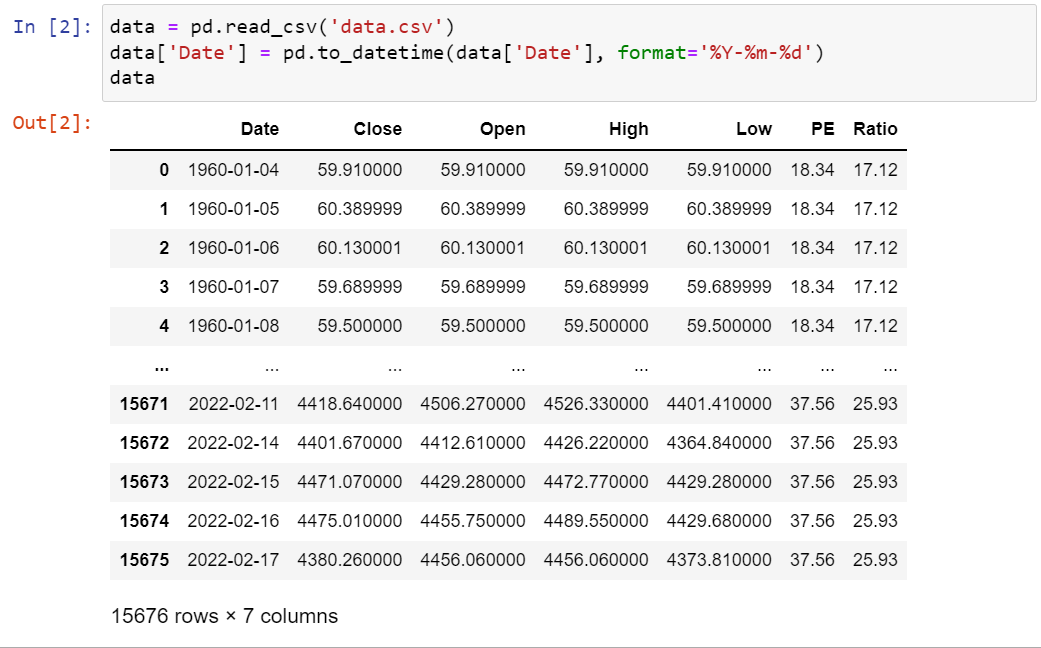


Figure 2.1: Stock Dataset: Python code

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 at time(year) t. The images of full data and test data are listed below in the left and right.

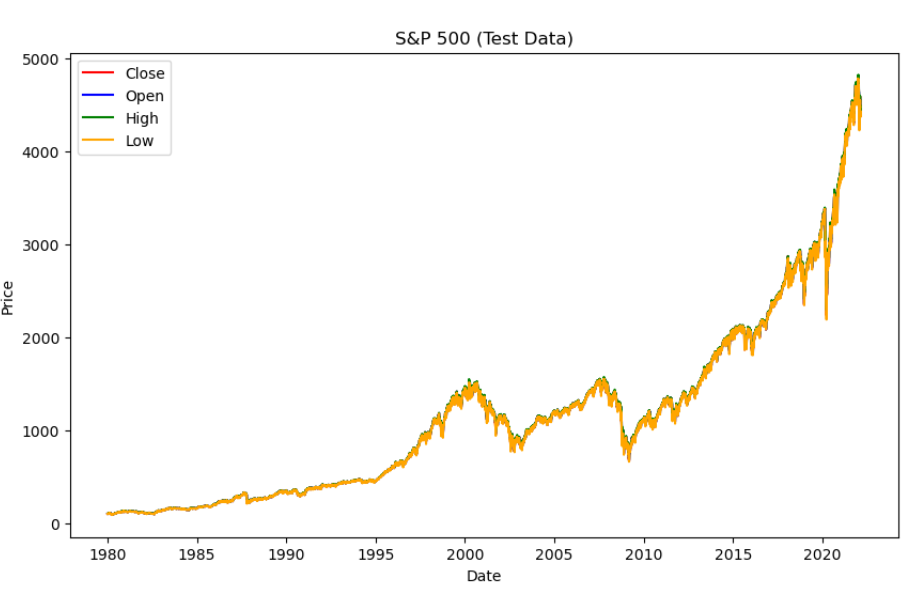
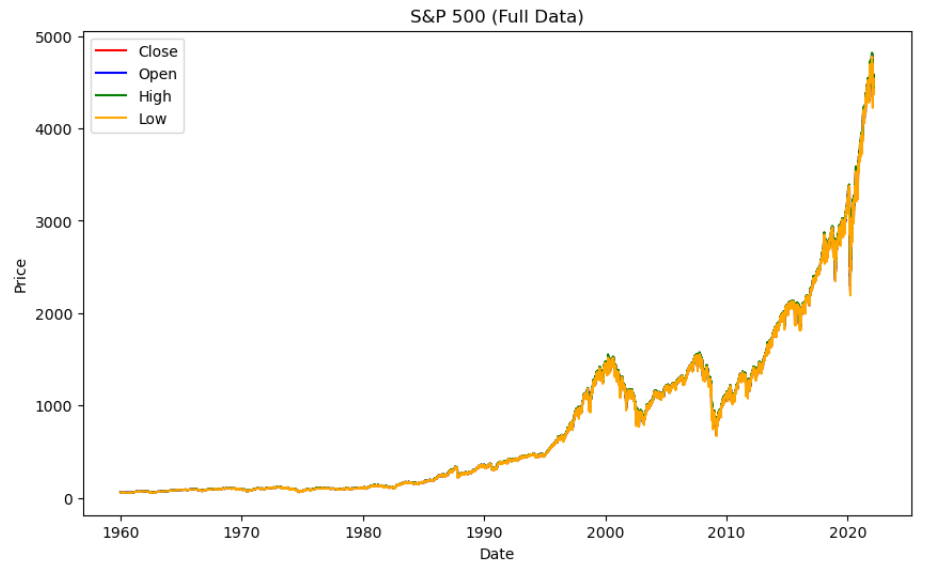


Figure 2.2: Full Data (1960-2020) Figure 2.3: Test Data (1980-2020)

2.2 RNN

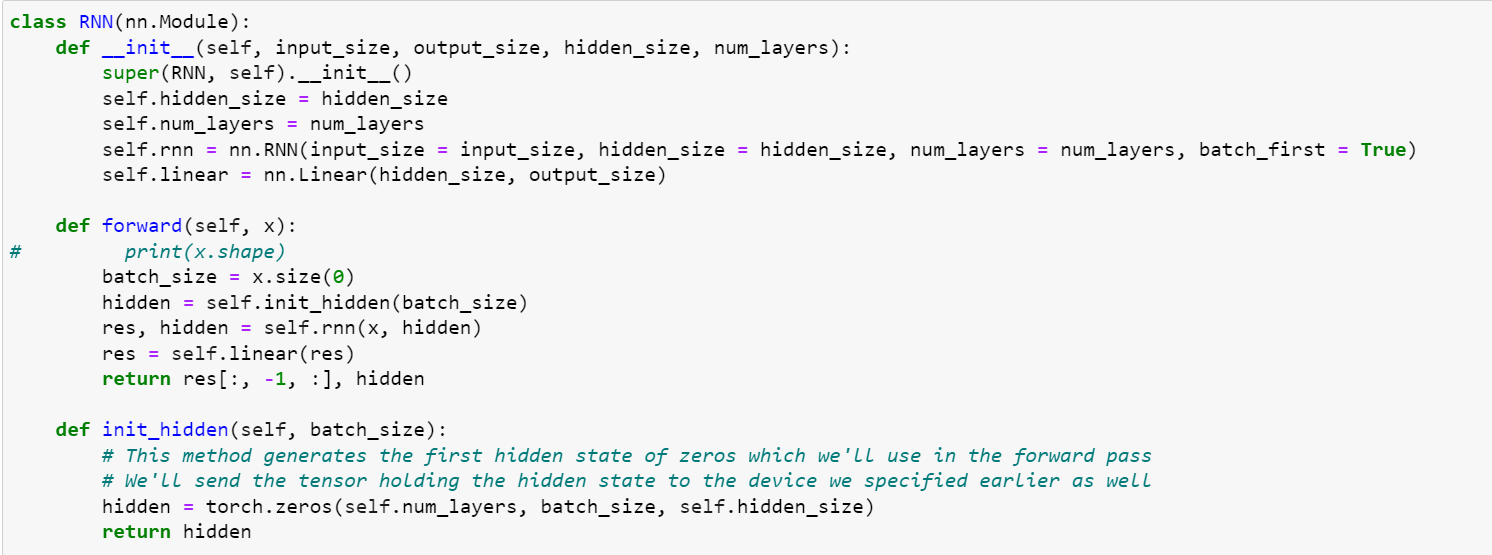


Figure 2.4: RNN Python code

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 length 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.

Training and Testing configuration

* 1. Training input and prediction: Noiseless

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

The prediction errors of inputs [Close, Open, High, Low] are listed one-by-one below.

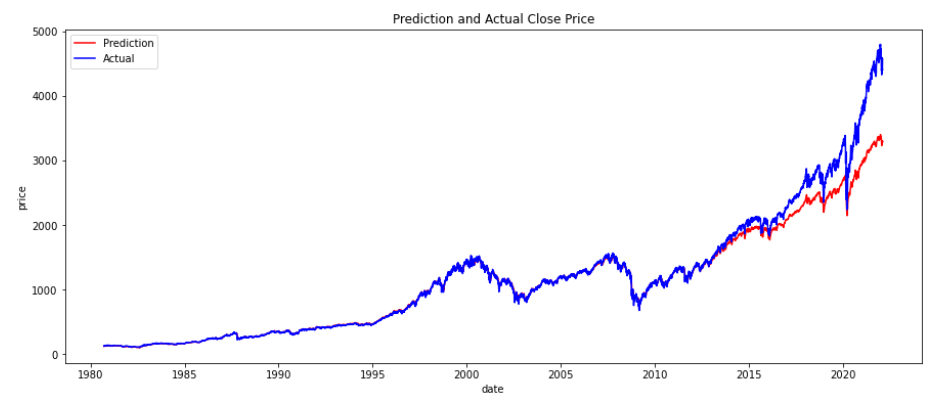


Figure 3.1: Noiseless Close Price input and prediction

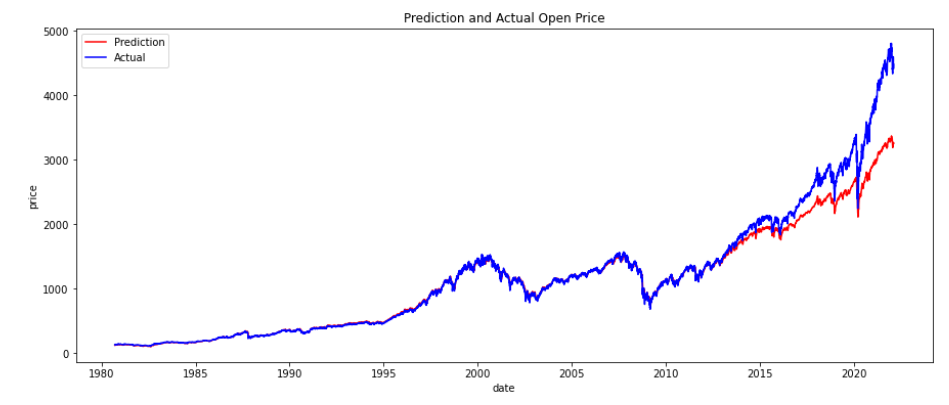


Figure 3.2: Noiseless Open Price input and prediction

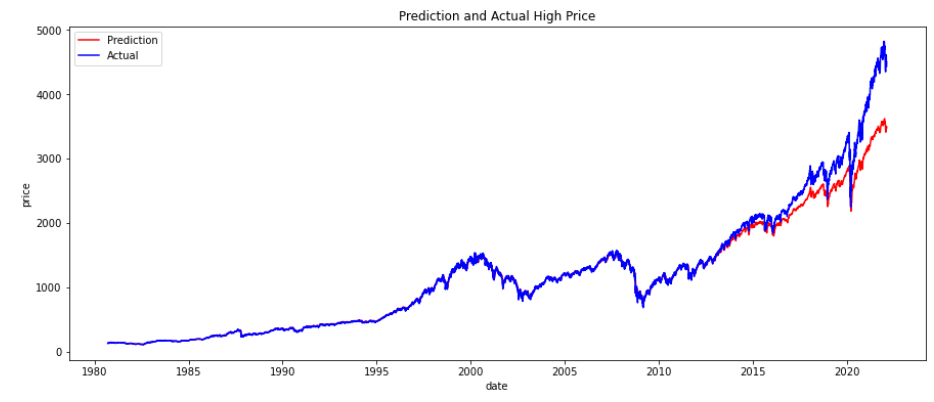


Figure 3.3: Noiseless High Price input and prediction

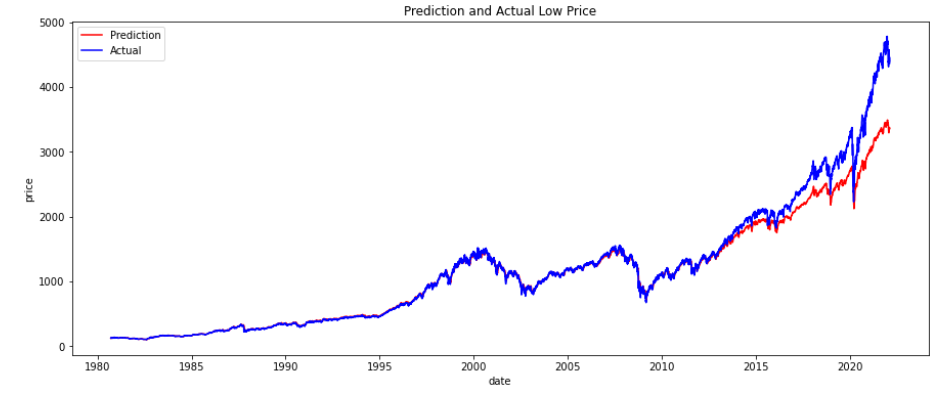


Figure 3.4: Noiseless Low Price input and prediction

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.

In an epoch

Make training sets in torch

Zero the weight gradients

Perform forward pass

Compute loss and Backward pass

Update parameters

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.

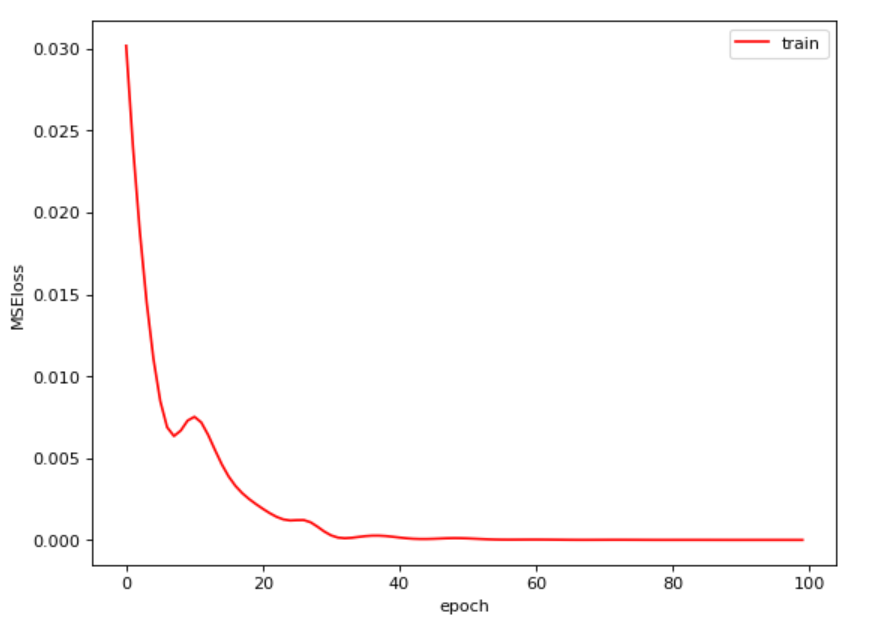


Figure 3.6: MSE Loss

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.

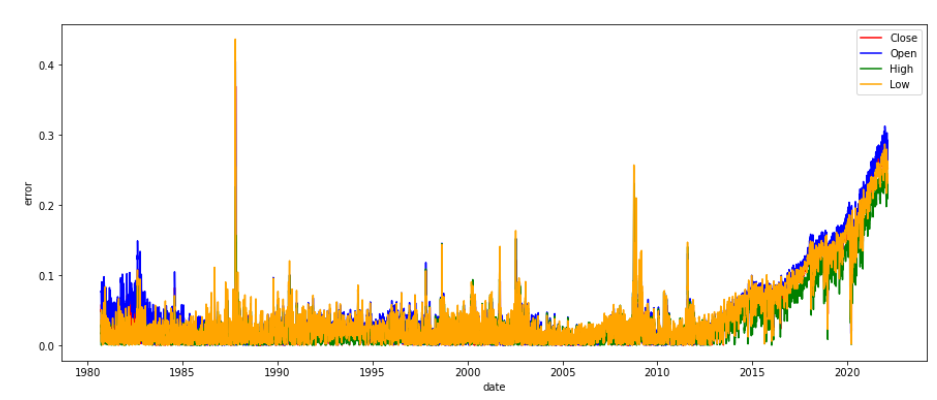


Figure 3.7: Error Percentage

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.

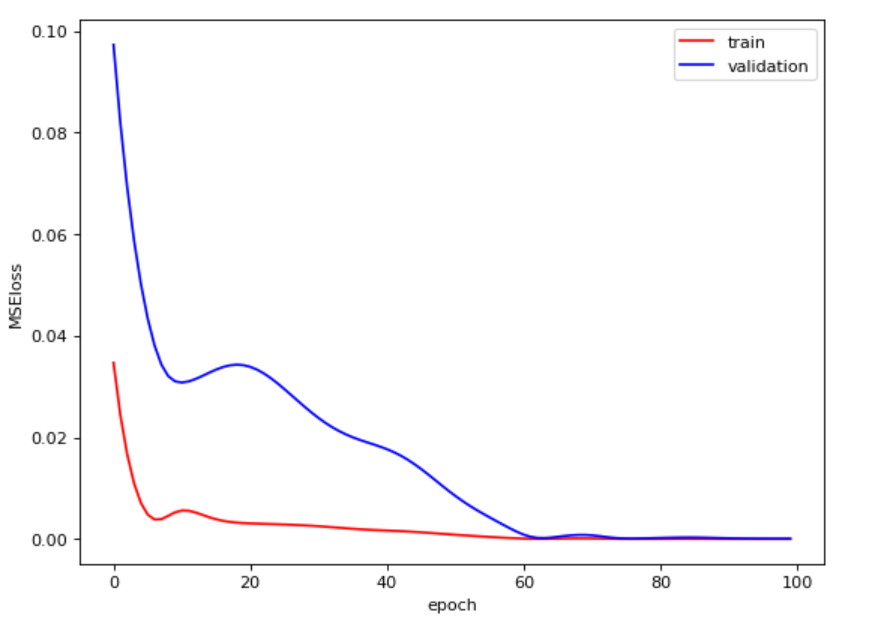


Figure 3.8: Loss of Train and Validation Set

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.

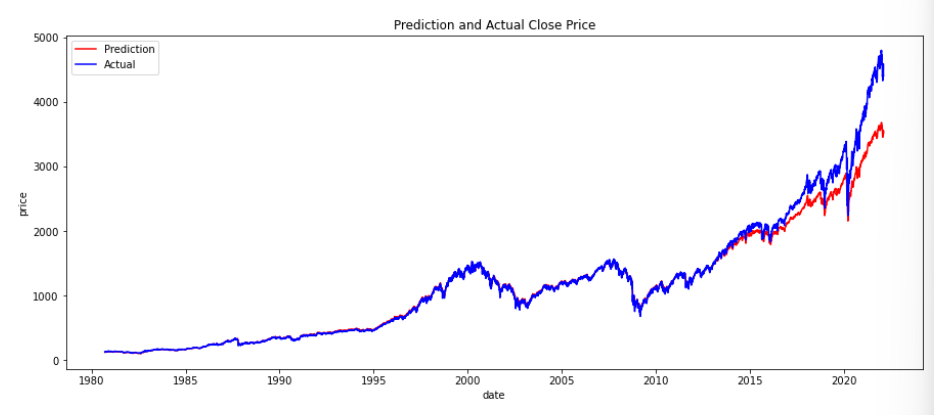


Figure 3.9: Noiseless Close Price with Validation Set

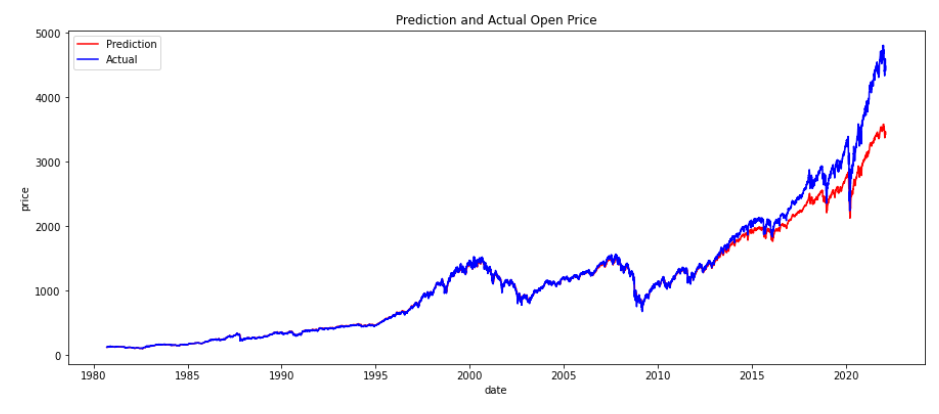


Figure 3.10: Noiseless Open Price with Validation Set

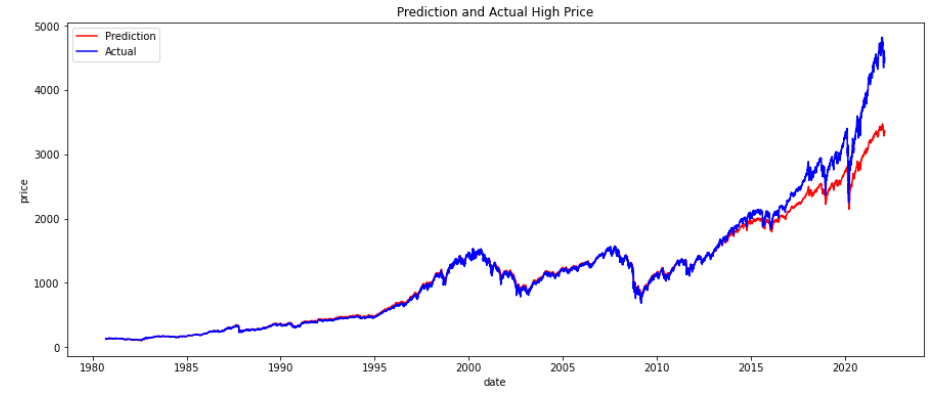


Figure 3.11: Noiseless High Price with Validation Set

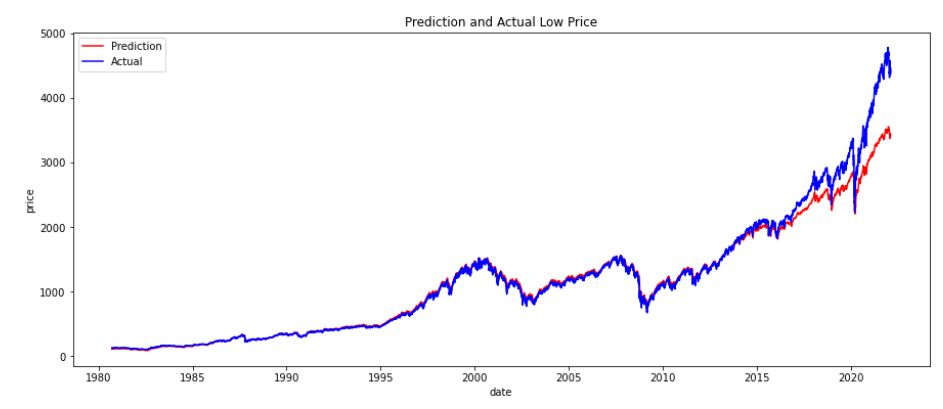


Figure 3.12: Noiseless Low Price with Validation Set

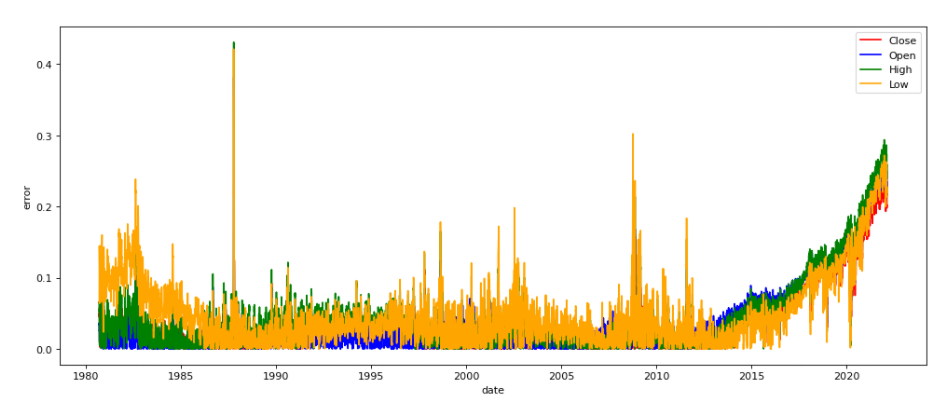


Figure 3.13: Noiseless Prediction Error

3.3 Training data: Noise-corrupted

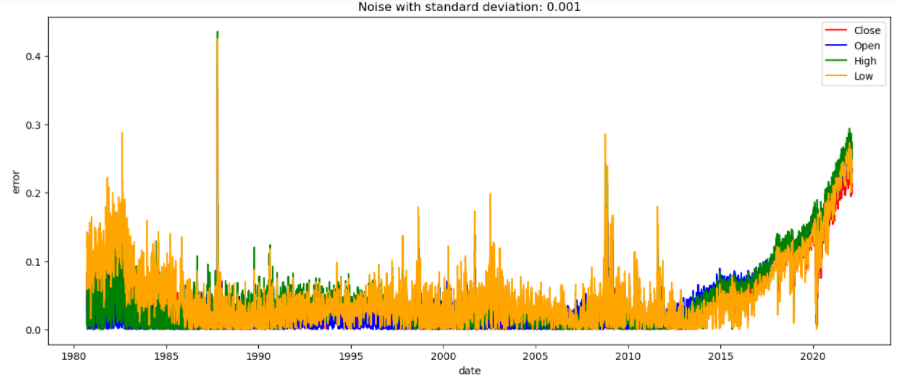


Figure 3.14: Noise-corrupted, SD 0.001

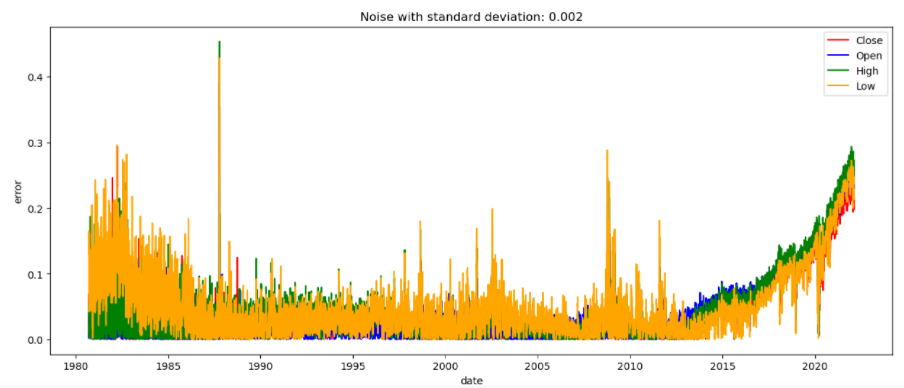


Figure 3.15: Noise-corrupted, SD 0.002

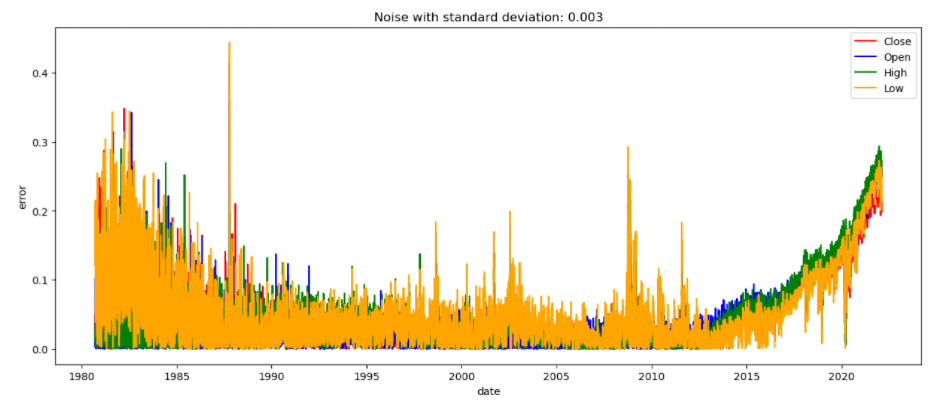


Figure 3.16: Noise-corrupted, SD 0.003

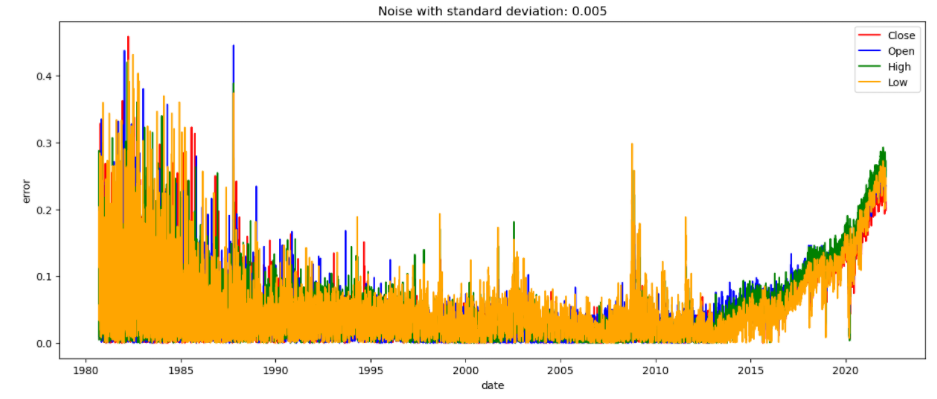


Figure 3.17: Noise-corrupted, SD 0.005

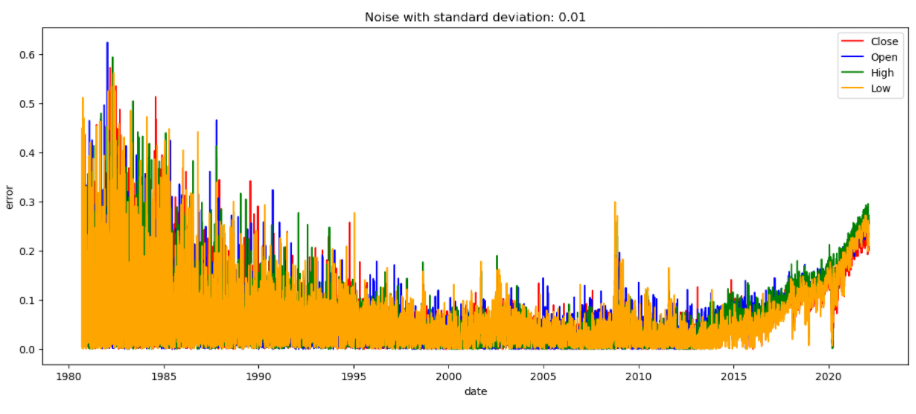


Figure 3.18: Noise-corrupted, SD 0.01

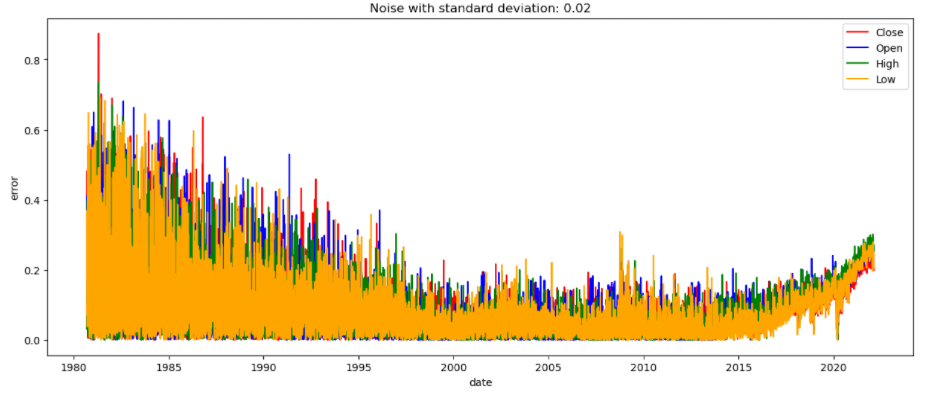


Figure 3.19: Noise-corrupted, SD 0.02

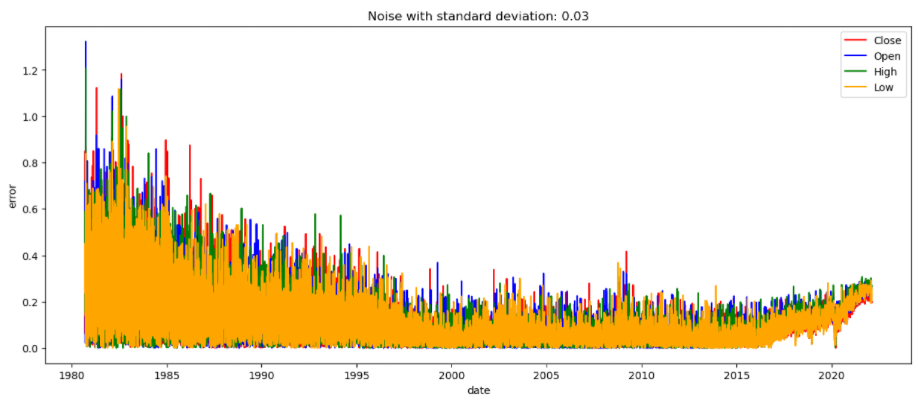


Figure 3.20: Noise-corrupted, SD 0.03

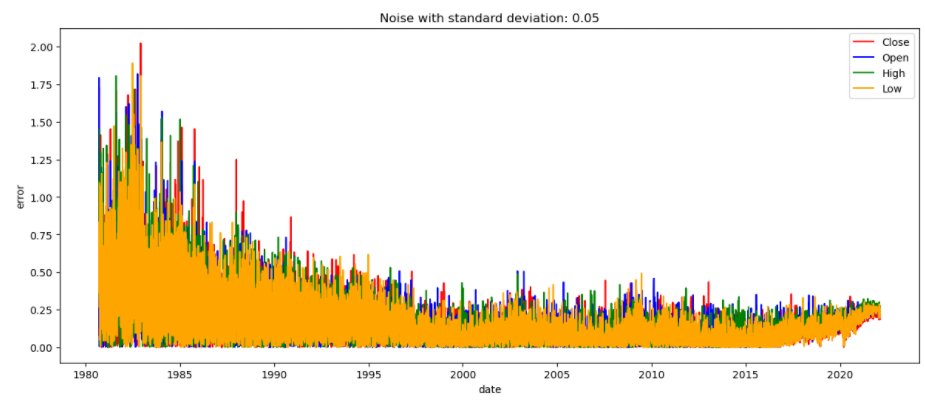


Figure 3.21: Noise-corrupted, SD 0.05

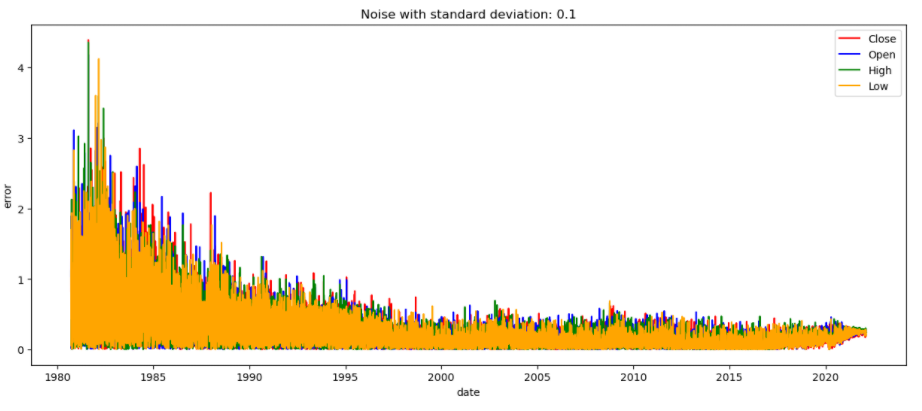


Figure 3.22: Noise-corrupted, SD 0.1

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.

1. Optimization
   1. Dataset Exclude 0 records

The improvement we did here is 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. Therefore, we removed records with 0 prices. And the performance of RNN gets better as what are shown below.

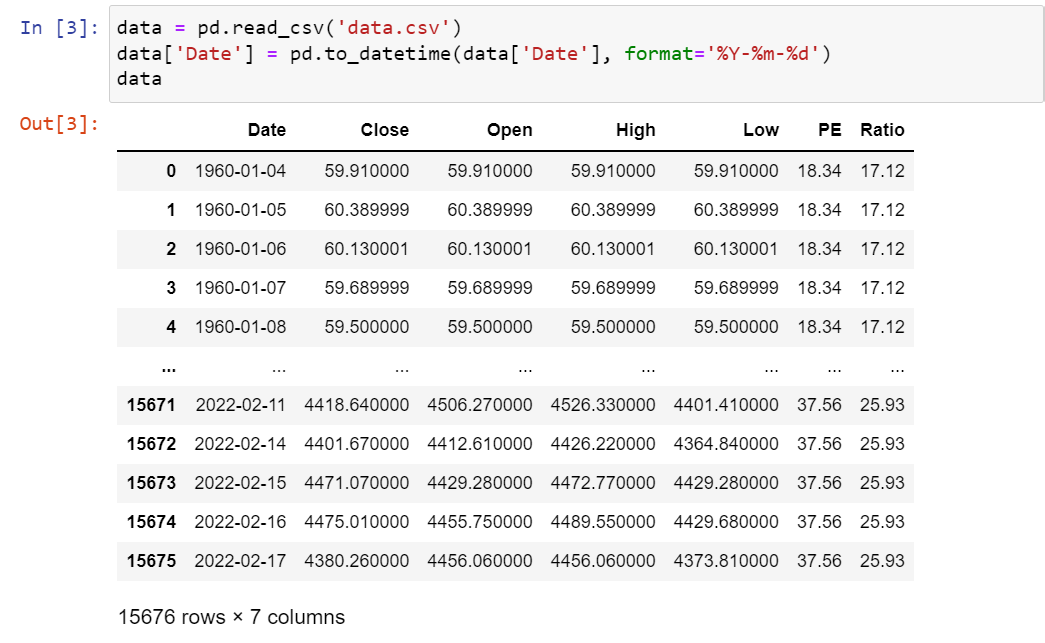
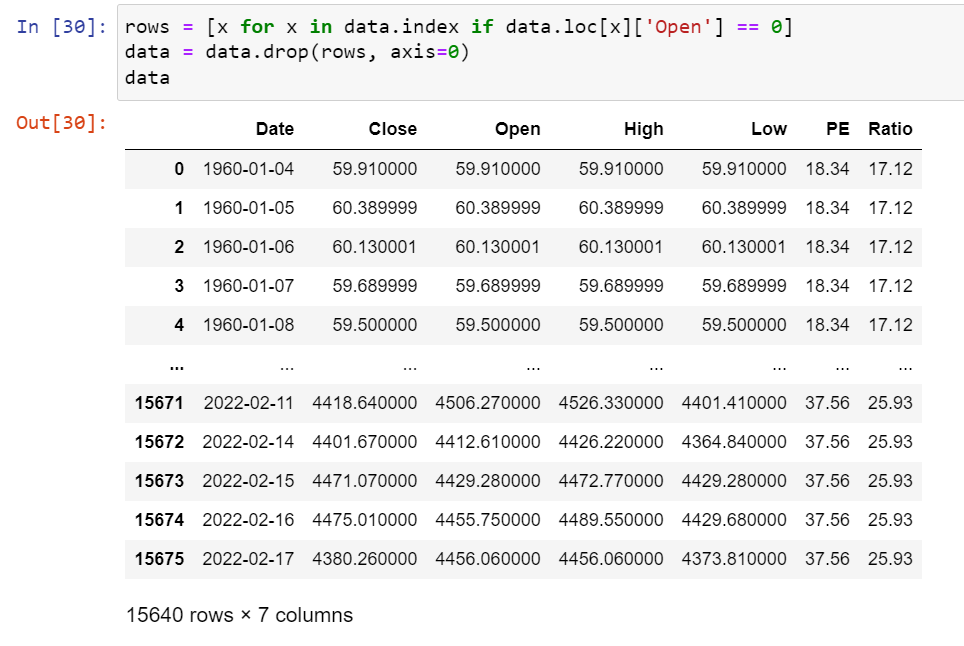
 

Figure 4.1: Data with 0 (10576 \* 7) Figure 4.1: Data without 0 (15640 \* 7)

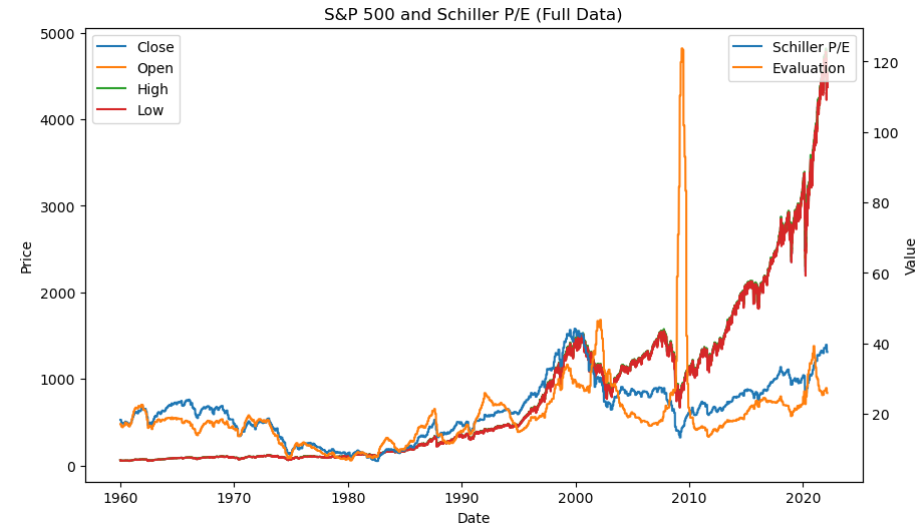


Figure 4.2: Full Data

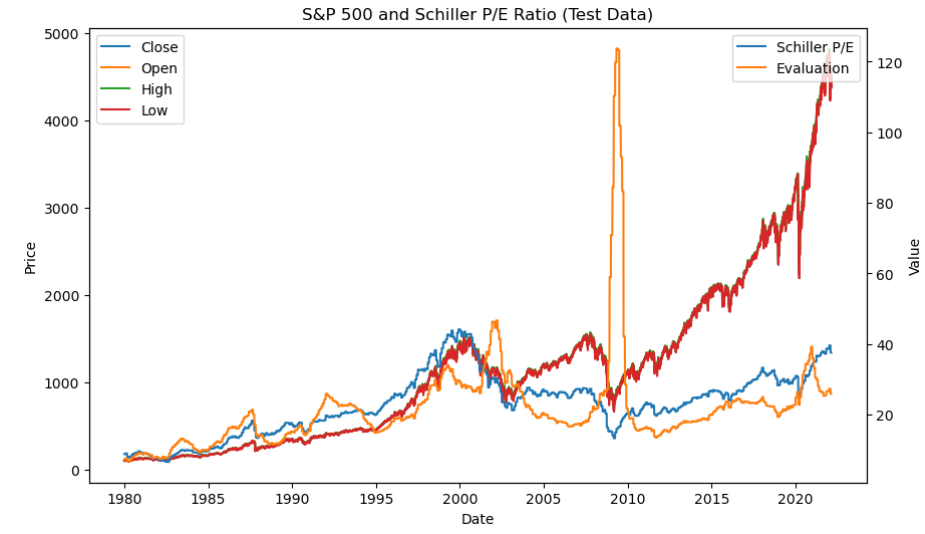
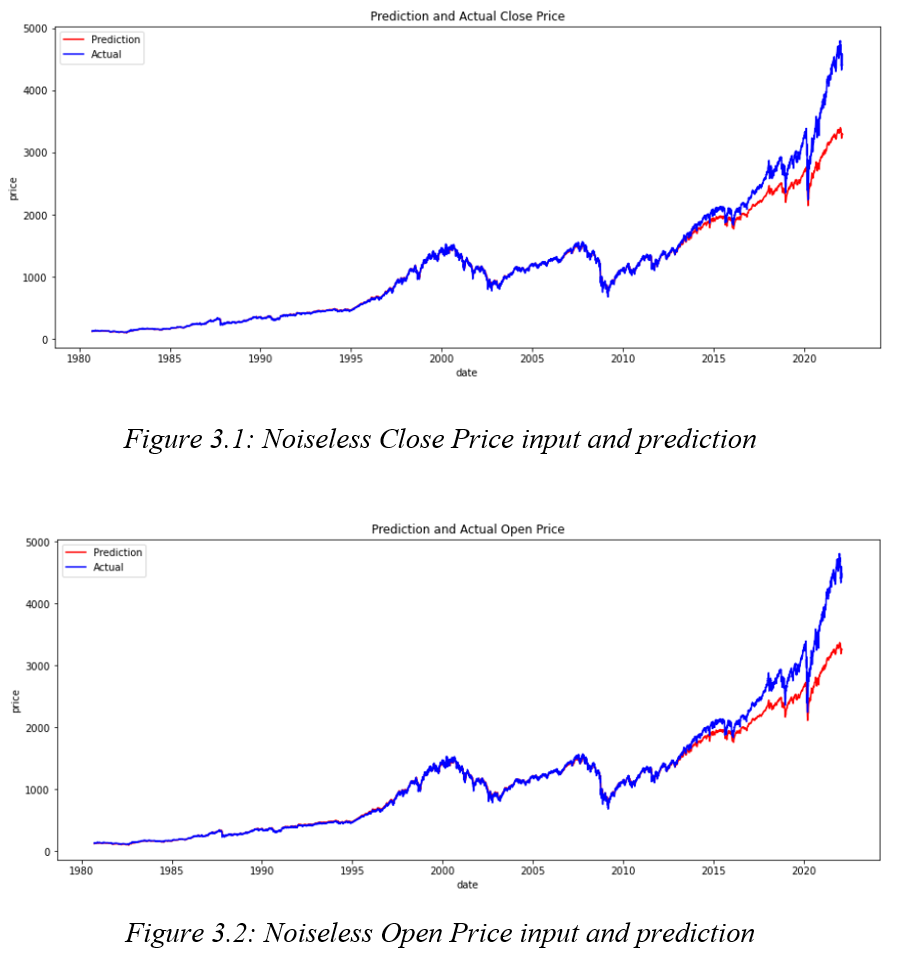
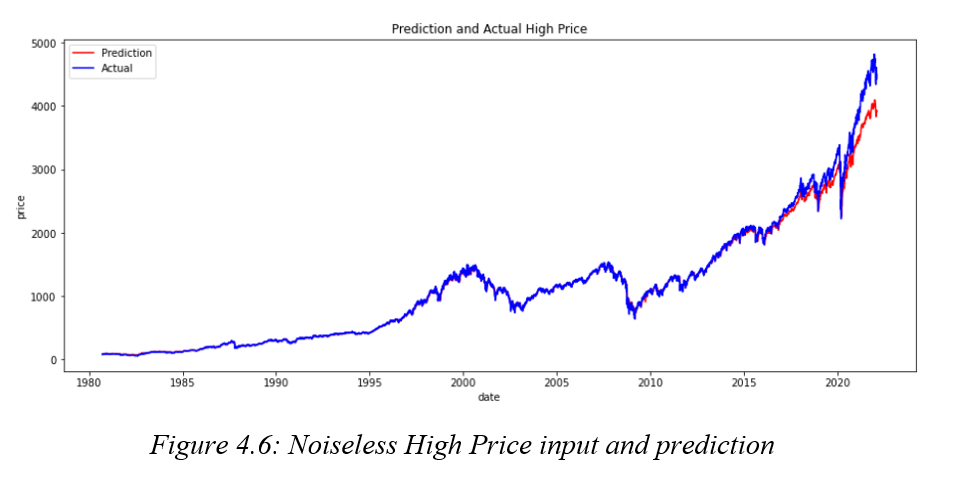
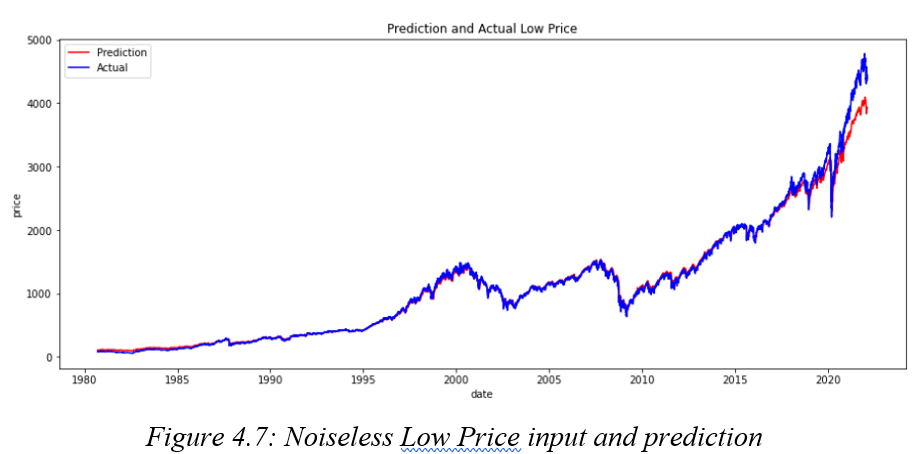


Figure 4.3: Test Data

4.2 Dataset Configuration with Prediction Comparation







After we get rid of the 0 prices, the accuracy of the prediction model improves really a lot. Comparing to not removing 0 records, the predicting line becomes fit, and be nearer to the original price line.

Besides, The MSE loss has a more in the training process shows that after data selecting, the MSE loss gets a little higher than before at the beginning, whereas it still reach 0 in the end.

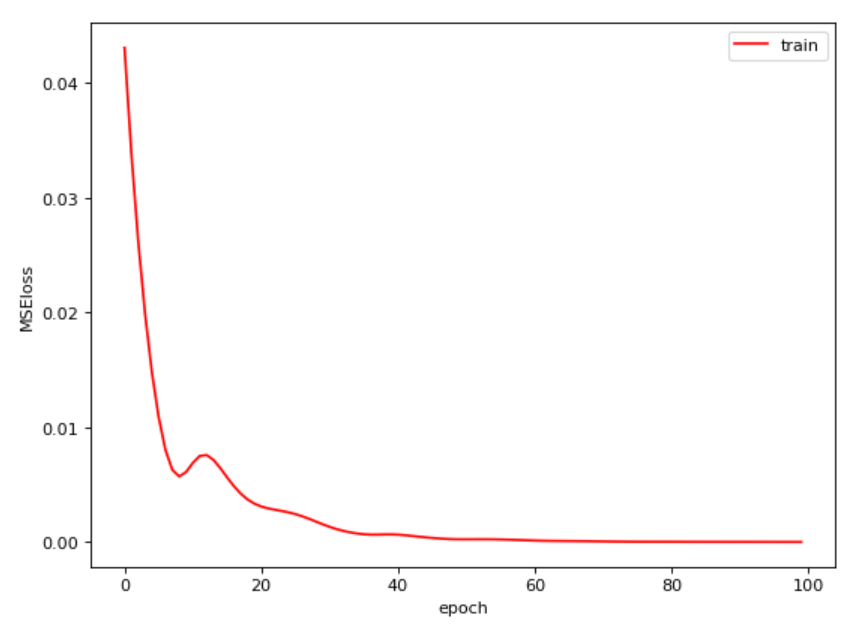
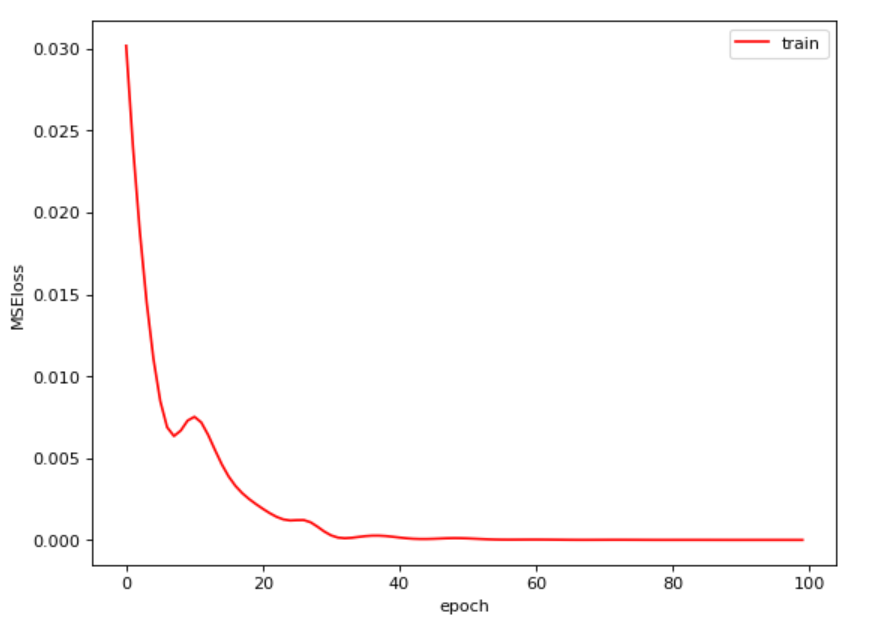


Figure 4.8: MSE Loss before and after Data optimizing

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 comes down later and primarily always lower than before.

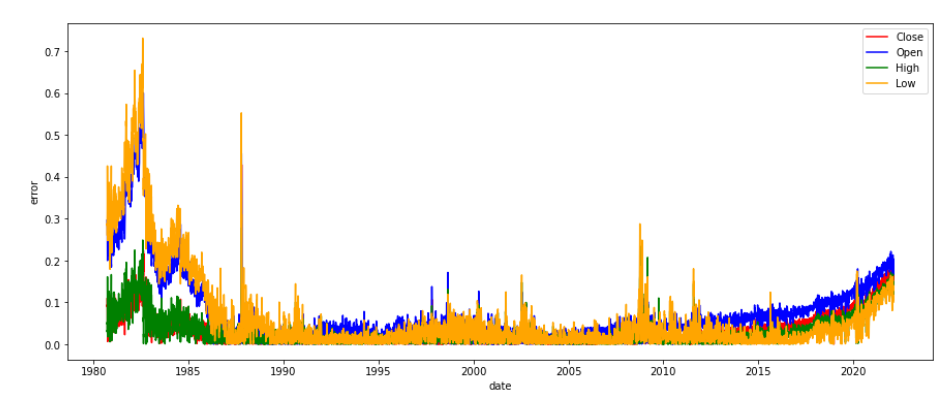


Figure 4.9: Error Percentage

4.3 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.

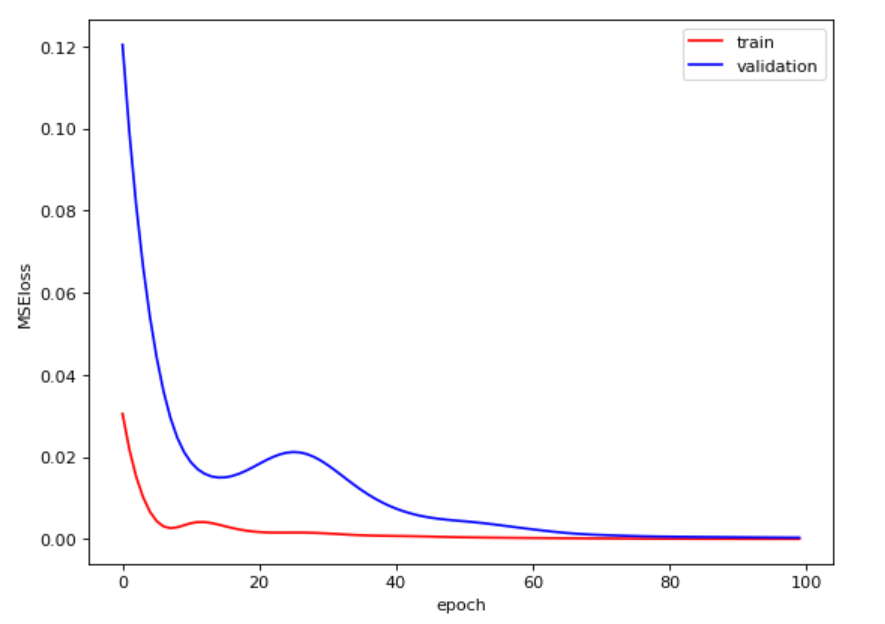
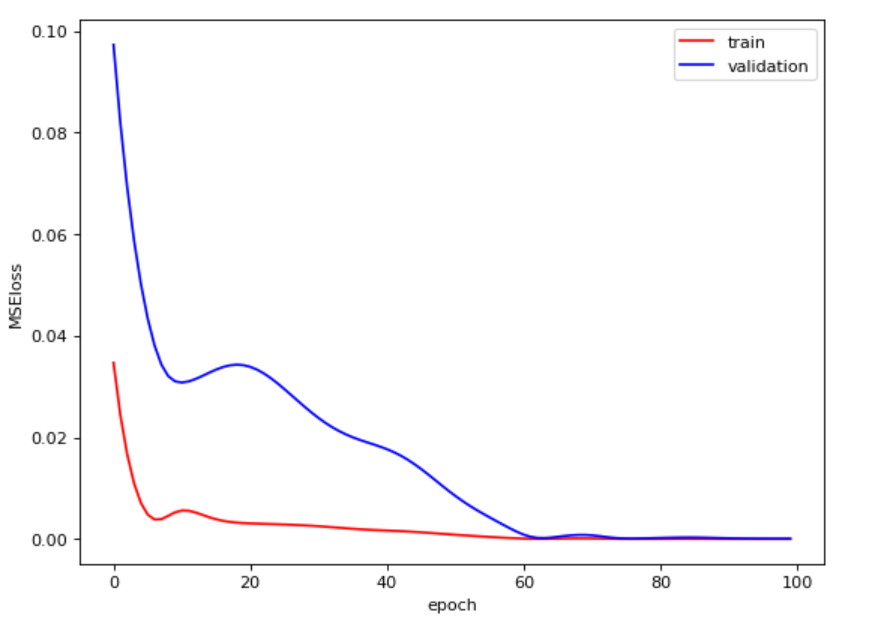
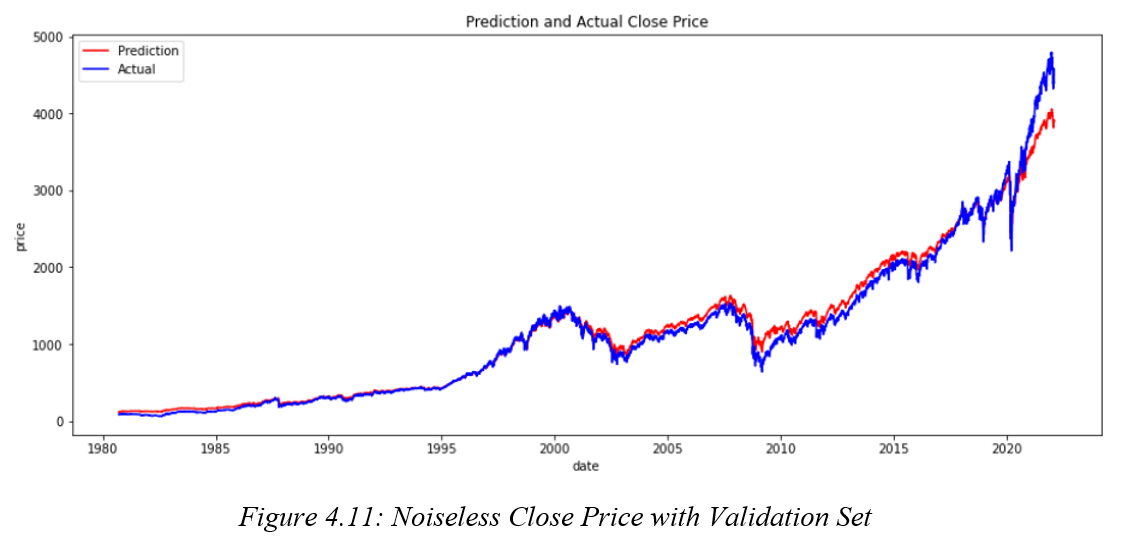
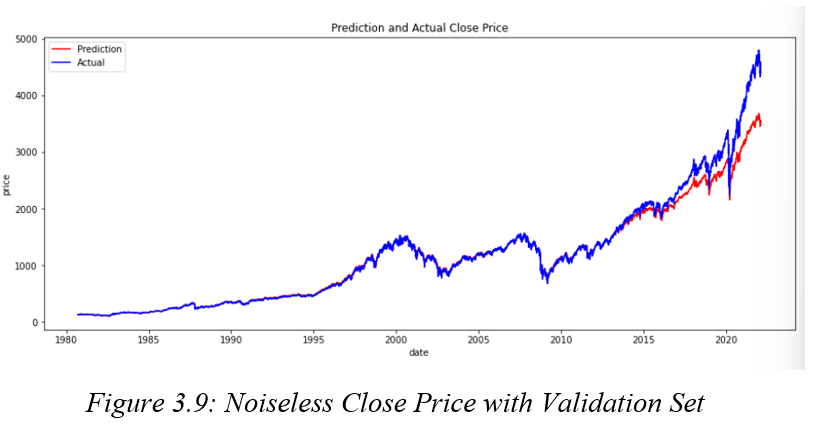
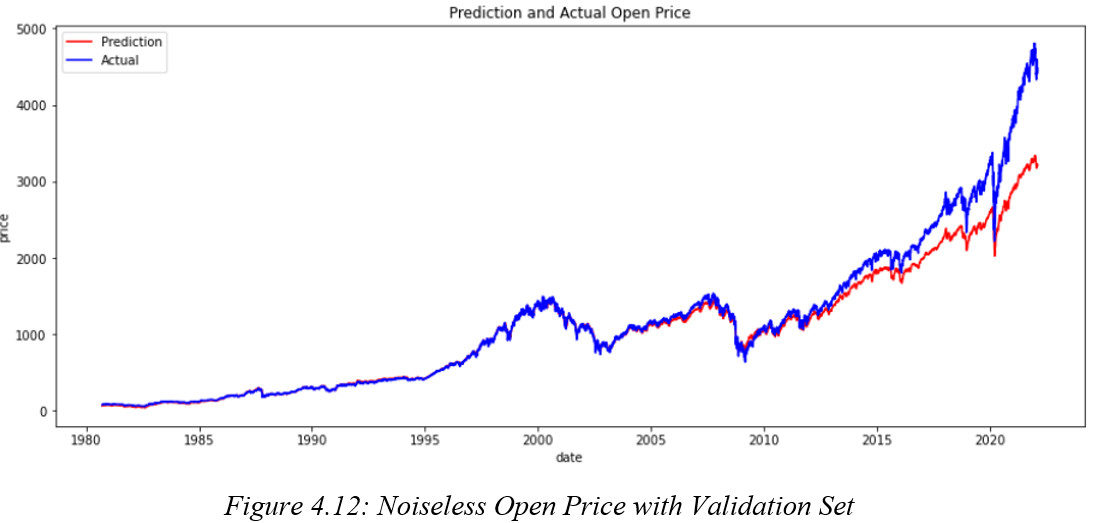
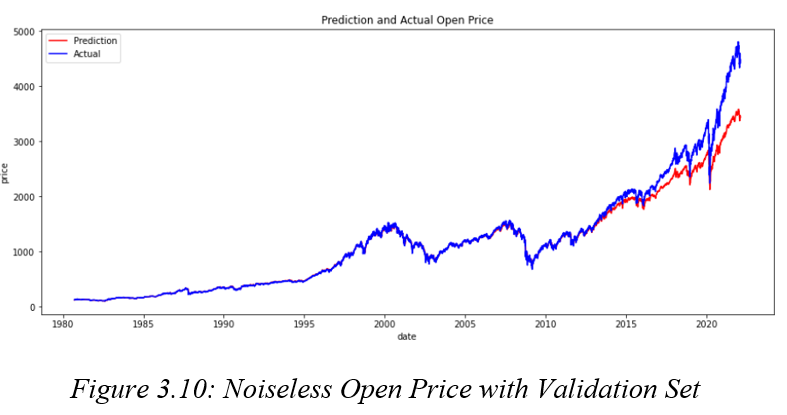
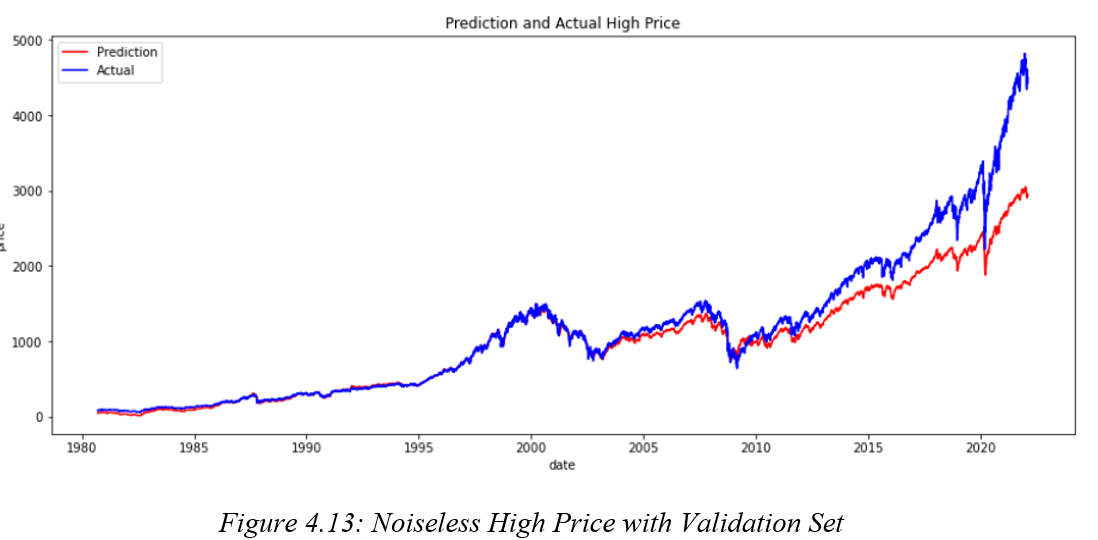
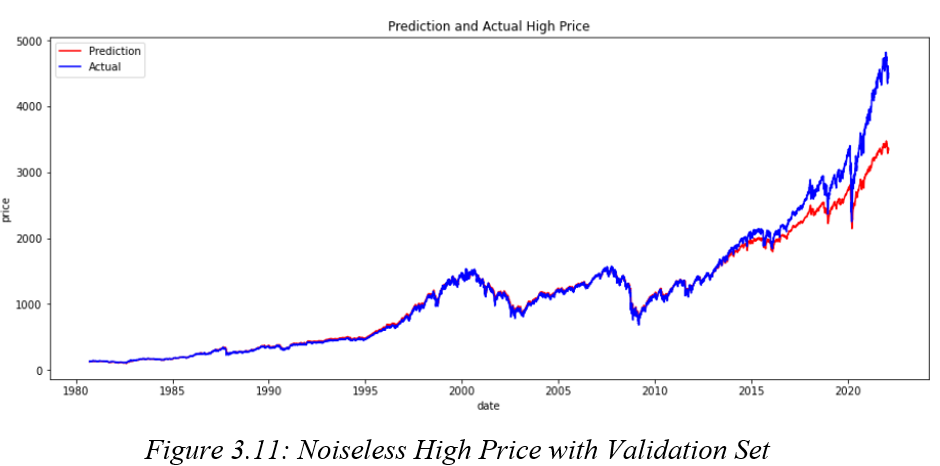


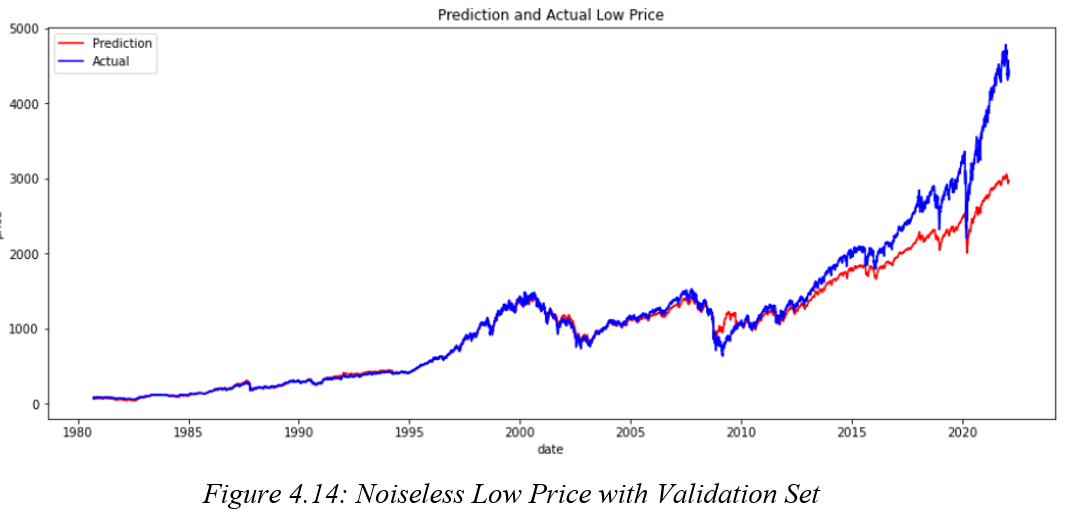
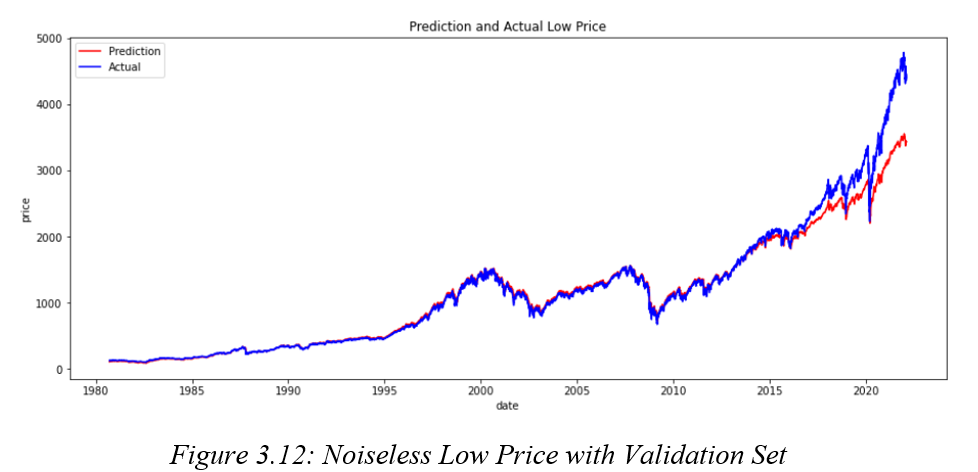
Figure 4.10: MSE Loss before and after

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

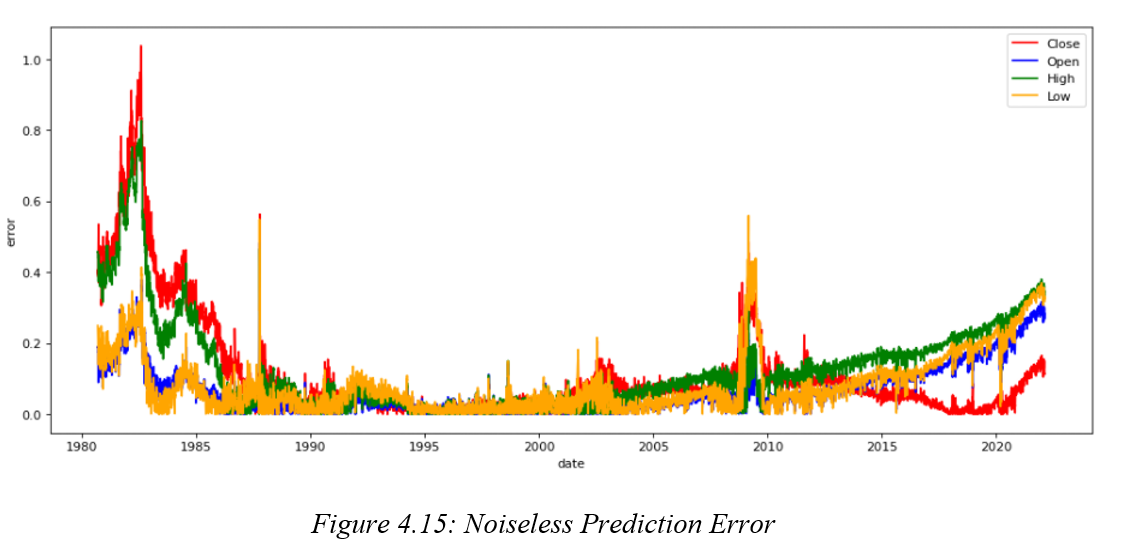
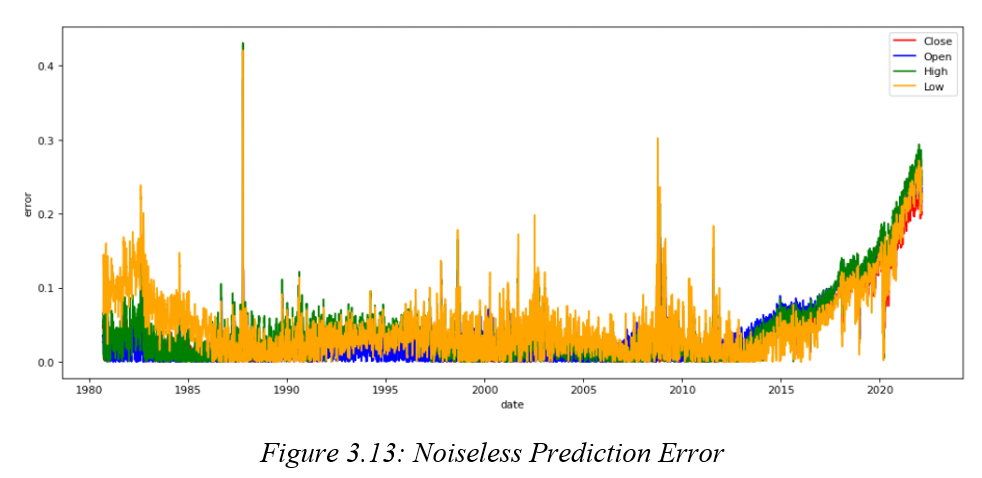








The prediction errors vary apparently in [Close, Open, High, Low] prices, and even be much higher than before in most time slots. Especially at the beginning of time series like 1980s, the Close price even gets the error near 1.0.



4.4 Noise-corrupted

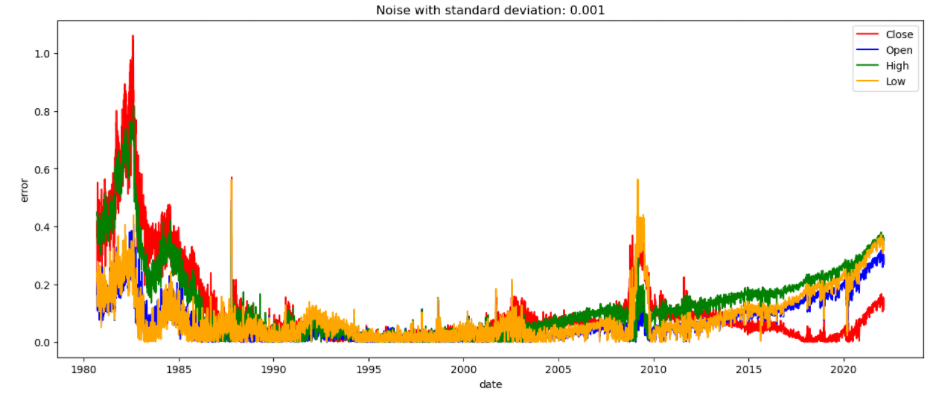


Figure 4.16: Noise-corrupted, SD 0.001

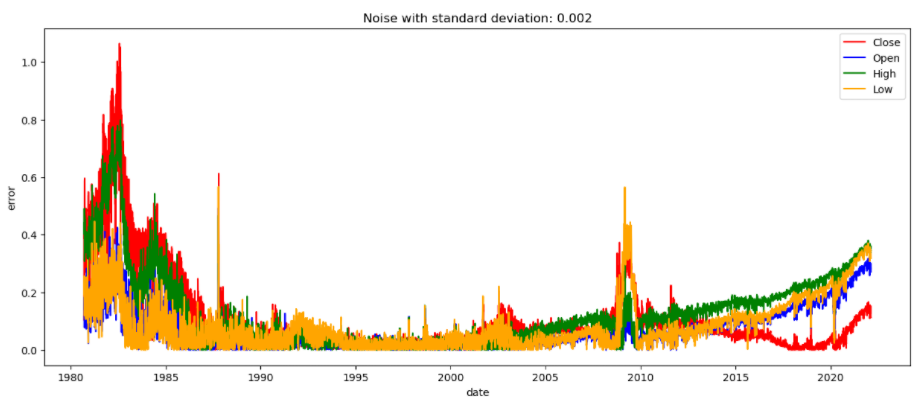


Figure 4.17: Noise-corrupted, SD 0.002



Figure 4.18: Noise-corrupted, SD 0.003

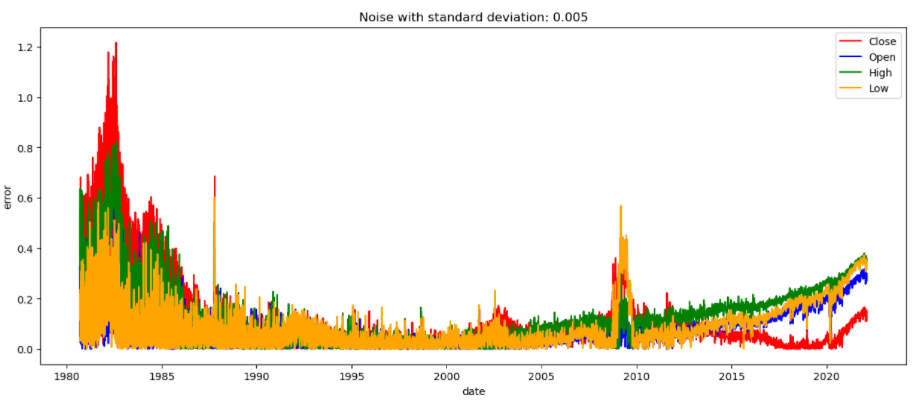


Figure 4.19: Noise-corrupted, SD 0.005

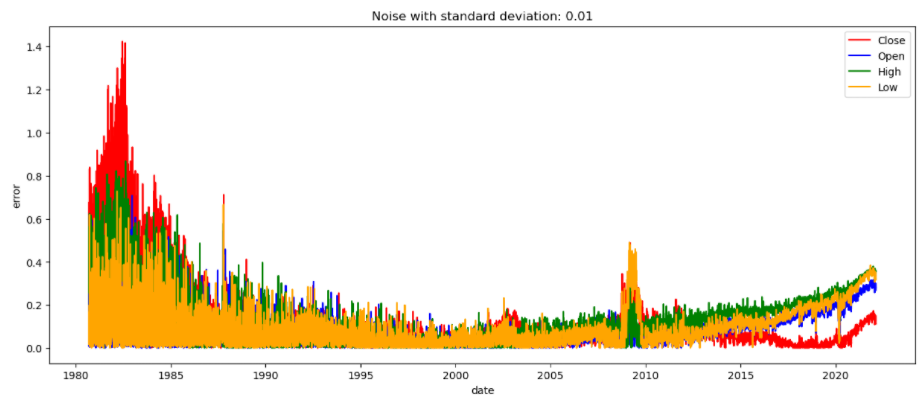


Figure 4.20: Noise-corrupted, SD 0.01

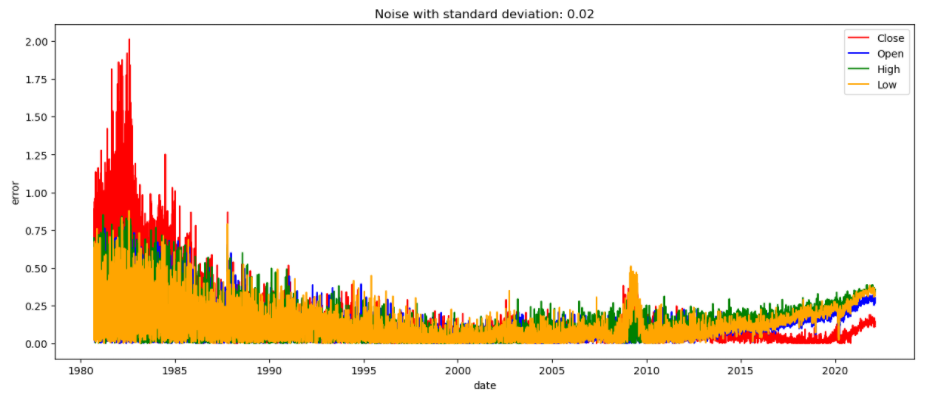


Figure 4.21: Noise-corrupted, SD 0.02

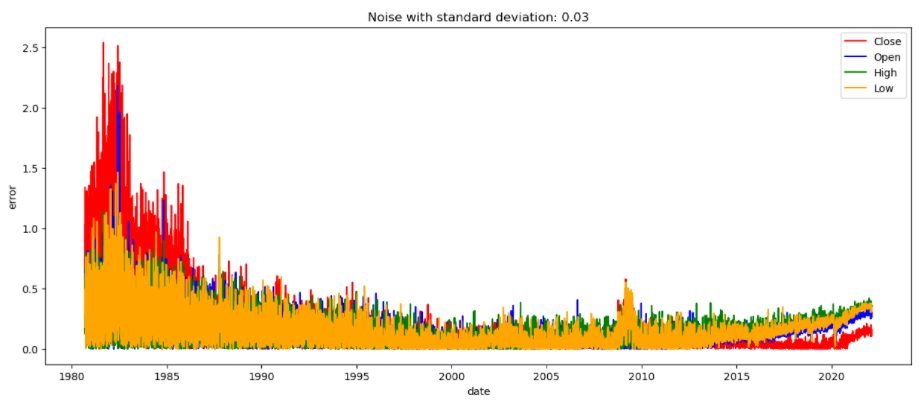


Figure 4.22: Noise-corrupted, SD 0.03



Figure 4.23: Noise-corrupted, SD 0.05



Figure 4.24: Noise-corrupted, SD 0.1

1. Pseudocode and Python code for algorithms to compute *Fh & Ffa*
   1. Pseudo Code
   2. Python Code
2. Discussion
   1. Workflow of Typeset Recognition
   2. Questions & Answers in Step 4c 16x16 32x32 (10 images comparison)
   3. Workflow of Gesture Recognition
3. Extra: 2-Stage Deep Neural-Network（2DNN）
   1. Parameter

|  |  |
| --- | --- |
| DNN: Stage 1 | |
| Parameter name | **Value** |
| Batch size | 26 |
| Learning Rate | 0.01 |
| Input Size | 256 |
| Number of classes | 1 |
| Number of Epoch | 200 |
| Hidden layer size 1 | 512 |
| Hidden layer size 2 | 256 |
| Hidden layer size 3 | 128 |
| Hidden layer size 4 | 64 |
| Hidden layer size 5 | 32 |

Table 3 2DNN: Stage 1 Parameters

|  |  |
| --- | --- |
| DNN: Stage 2 | |
| Parameter name | **Value** |
| Batch size | 26 |
| Learning Rate | 0.001 |
| Input Size | 1 |
| Number of classes | 256 |
| Number of Epoch | 2500 |
| Hidden layer size 1 | 128 |
| Hidden layer size 2 | 256 |
| Hidden layer size 3 | 256 |

Table 4 2DNN: Stage 2 Parameters

* 1. Python Code: 2DNN
  2. Training Set Configuration
  3. Visualization of *Fh*& *Ffa: Noiseless* 
     1. Dataset#1: *Fh*& *Ffa*
     2. *Dataset#2: Fh & Ffa*
  4. Pseudocode and Python code for computing Fh & Ffa

8.5.1 Pseudocode

8.5.2 American Sign Language gestures: Python Code

* 1. Visualization of Fh & Ffa and results: Noise-corrupted
  2. Discussion