CAP 6615 - Neural Networks

Programming Assignment 4 -- CNN+RNN for Time Series Prediction

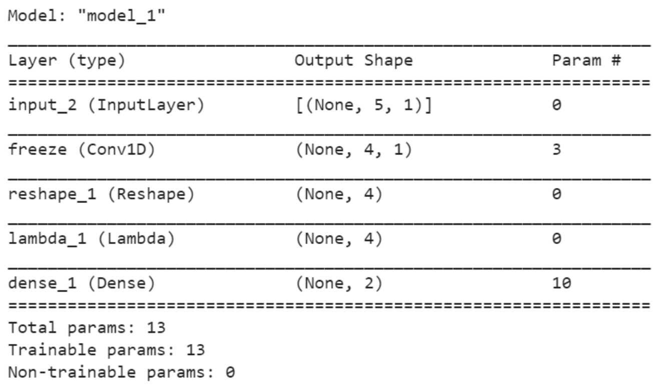
Zhounan Li, Tongjia Guo, Huaiyue Peng, Mingjun Yu, Zhiyun Ling

Spring Semester 2021, 21 April 2021

Network parameters

Wave detection model

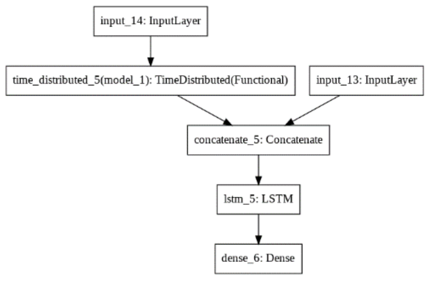
The wave detection model is used to detect the wave pattern in a 5-steps long series. Intuitively, to check the wave-pattern, we need to recognize the ‘up-down-up-down’ shape. So the CNN’s kernel size is (2, 1, 1), which is used to compute the ‘difference’. While training, the kernel’s weights is frozen. Also, we use Relu activation function, in CNN layer. After that , we cut the difference to 1 if the difference is larger than 0.



```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 5, 1)]            0
_________________________________________________________________
freeze (Conv1D)              (None, 4, 1)              3
_________________________________________________________________
reshape_1 (Reshape)          (None, 4)                 0
_________________________________________________________________
lambda_1 (Lambda)            (None, 4)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 10
=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
```

CNN-RNN model

In this model, I pick two input tensors, one for LSTM and another for the wave detection. The wave detection model is covered by a TimeDistributed Layer. For a faster training speed, we use the default activation function on LSTM layer, because if we replace the activation function with any other one, we can utilize the GPU resource, which will make the training speed super slow.


input_14: InputLayer
time_distributed_5(model_1): TimeDistributed(Functional)
input_13: InputLayer
concatenate_5: Concatenate
lstm_5: LSTM
dense_6: Dense

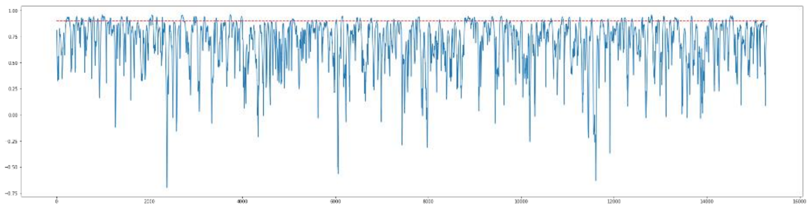
```
____________________________________________________________________________________________________
Layer (type)                        Output Shape              Param #     Connected to
====================================================================================================
input_14 (InputLayer)               [(None, 56, 5, 1)]        0
____________________________________________________________________________________________________
input_13 (InputLayer)               [(None, 56, 1)]           0
____________________________________________________________________________________________________
time_distributed_5 (TimeDistributed)  (None, 56, 2)           13          input_14[0][0]
____________________________________________________________________________________________________
concatenate_5 (Concatenate)         (None, 56, 3)             0           input_13[0][0]
                                                                          time_distributed_5[0][0]
____________________________________________________________________________________________________
lstm_5 (LSTM)                       (None, 8)                 384         concatenate_5[0][0]
____________________________________________________________________________________________________
dense_6 (Dense)                     (None, 1)                 9           lstm_5[0][0]
====================================================================================================
Total params: 406
Trainable params: 393
Non-trainable params: 13
```

Python Codes

Correlation algorithm

```python
def cor(s1, s2):

        m1 = np.mean(s1)

        m2 = np.mean(s2)

        # # remove the standard deivation, because PE is too small compared to SP500 value
        std1 = np.std(s1)
        std2 = np.std(s2)

        # print(f"mean1: {m1}, mean2: {m2}, std1: {std1}, std2: {std2}")
        s1_new = s1 - m1
        s2_new = s2 - m2

        s1_new /= std1
        s2_new /= std2

        cov = np.sum(s1_new * s2_new)

        dev1 = math.sqrt(np.sum(s1_new * s1_new))
        dev2 = math.sqrt(np.sum(s2_new * s2_new))

        return cov / (dev1*dev2)
```

the function above is used to compute the correlation between two series.



After computing the correlation, we set the threshold as 0.9, then the point above the red line are the ‘sell signals’

Wave detection model

```
model_ipt = Input(shape=(Tx_wave, 1))

cnn_layer_opt = Conv1D(filters=1, kernel_size=(2, ), strides=1, padding='valid',activatio
n='relu', kernel_initializer=tf.keras.initializers.Constant(
    value=np.array([[[-1]], [[1]]])
), name='freeze')(model_ipt)
reshape_layer_opt = Reshape((-1, ))(cnn_layer_opt)
cut_layer_opt = Lambda(lambda x: tf.math.minimum(x, 0.1))(reshape_layer_opt)
model_opt = Dense(2, activation='softmax')(cut_layer_opt)
wave_detect_model_new =Model(model_ipt, model_opt)
```

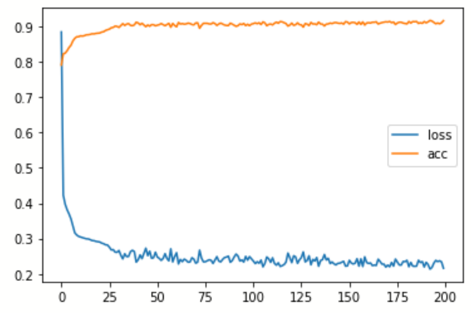
```
wave_detect_model_new.get_layer('freeze').trainable=False
```

Above is the code for building the wave detection model. In the code, I use the tf initializer to initialize the CNN's kernel. And I use the Lambda layer to cut the difference to at most 0.1. Before training, I froze the CNN layer to prevent its kernel weights to be changed.

For training the model, I would like to use supervised learning. So I made the training set for this model by myself by checking the 'peak' and the 'valley'.

```
# detect summit and valley in this range
idx_summit = []
idx_valley = []
# if it is a wave pattern, there will be two summits and only one valley
for j in range(i-Tx_wave+1, i-1):
if sp[j-1] < sp[j] > sp[j+1]:
                idx_summit.append(j)

if sp[j-1] > sp[j] < sp[j+1]:
                idx_valley.append(j)

#  and 0.7<(sp[idx_summit[1]] - sp[idx_valley[0]]) / (sp[idx_summit[0]] - sp[i-Tx_wave]) < 1.43
if len(idx_summit) == 2 and len(idx_valley) == 1 and sp[idx_summit[1]] > sp[idx_summit[0]]:
data_Y_CNN.append(1)
else:
data_Y_CNN.append(0)
```

As a result, the dataset is an unbalanced dataset, so use over_sampling to get more positive samples. The training process is below.



By testing the recall, accuracy, precision score on the original dataset, the performance is good. (0.9902200488997555, 0.9740065146579805, 0.50625). The recall and accuracy scores are super high, while the precision a little bit lower. But it doesn't matter, because the number of positive samples is too small, although we predict some negative samples to be positive, it doesn't matter.

CNN _ RNN model

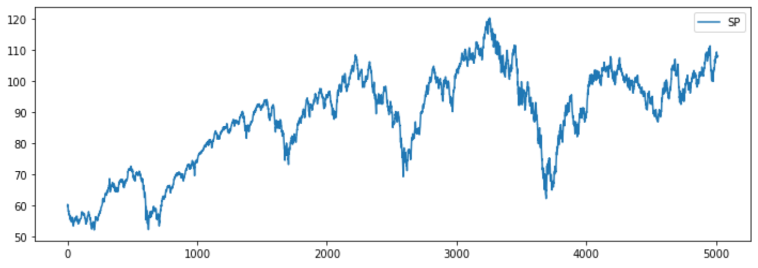
```
rnn_ipt = Input(shape=(Tx-4, 1))

wave_detect_ipt = Input(shape=(Tx-4, 5, 1))
wave_detect_opt = TimeDistributed(wave_detect_model_new)(wave_detect_ipt)
concat_opt = Concatenate(axis=-1)([rnn_ipt, wave_detect_opt])

lstm_opt = LSTM(8, return_sequences=False)(concat_opt)
opt = Dense(1)(lstm_opt)
model = Model([rnn_ipt, wave_detect_ipt], opt)
wave_detect_model_new.trainable=False
```

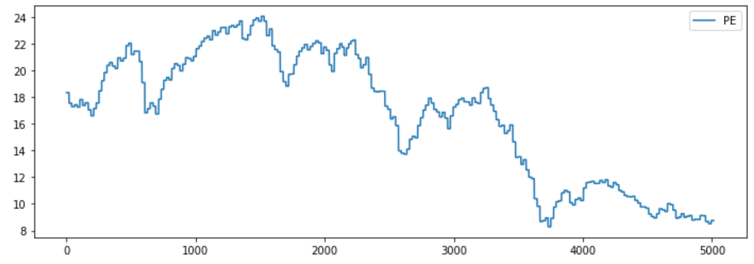
I set the wave_detect model to be untrained before training the model.

Here I just put the model build codes. The training details will be shown later.

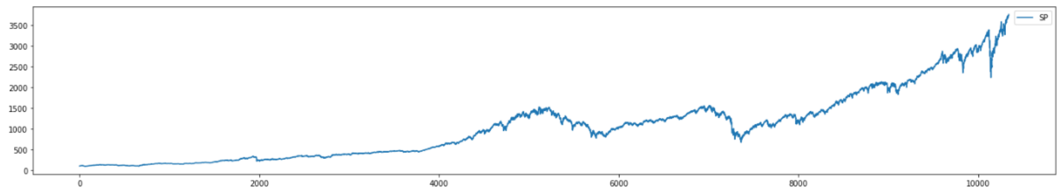
Train and test set

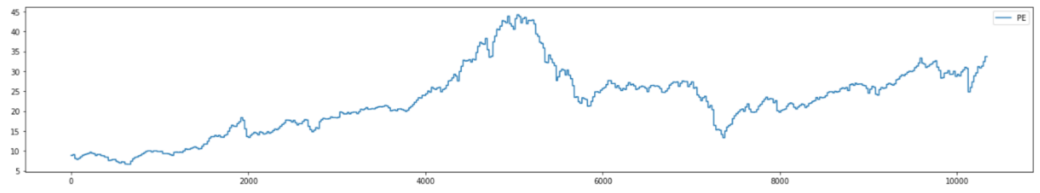
According to the requirement, the dataset after 1980 should serve as the test dataset. So split the train and test set using 1980.





Above is the training set of pe and sp





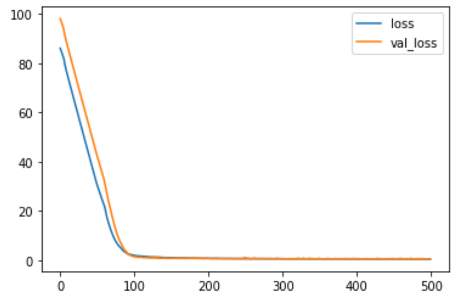
Above is the test set of pe and sp

To make the dataset for training, I use the slide window with a size of 60 to make the dataset.

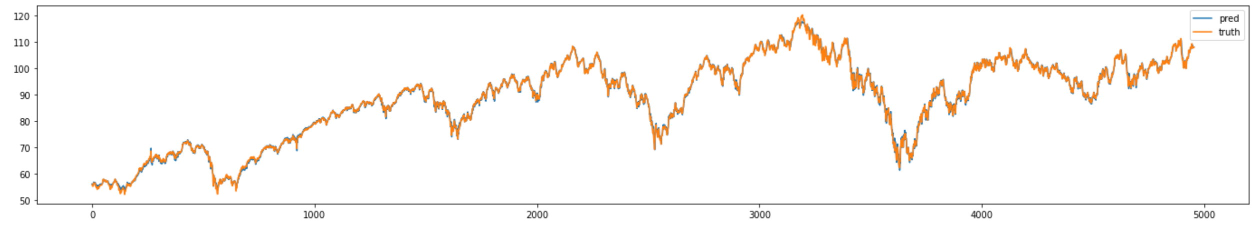
```
data_x_rnn = []
data_x_wave = []
data_y = []

for i in range(Tx, len(sp_train)):
    data_x_rnn.append(sp_train[i-Tx+4:i])
    data_y.append(sp_train[i])
    temp_wave = []
    for j in range(i-Tx+5, i+1):
        temp_wave.append(sp_train[j-5:j])
    data_x_wave.append(temp_wave)
```

Unoptimized Network

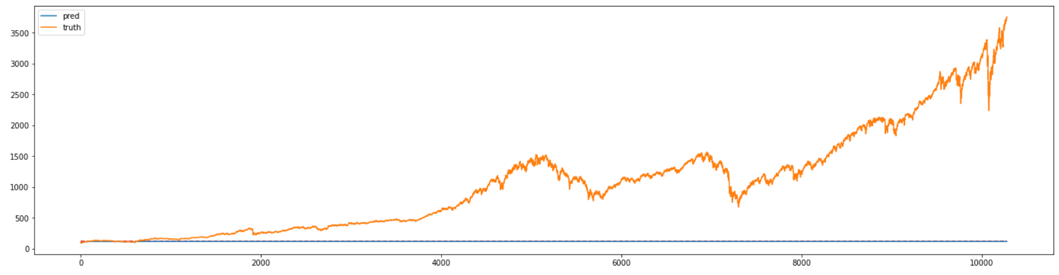
In the unoptimized model, I just used the training that I made in last step to train the model for 500 epochs. The validation dataset in split from train set.



The training process is smooth. And the predict result on the train and validation set is below.



The performance seems to be fine. But it doesn't work on the test dataset.

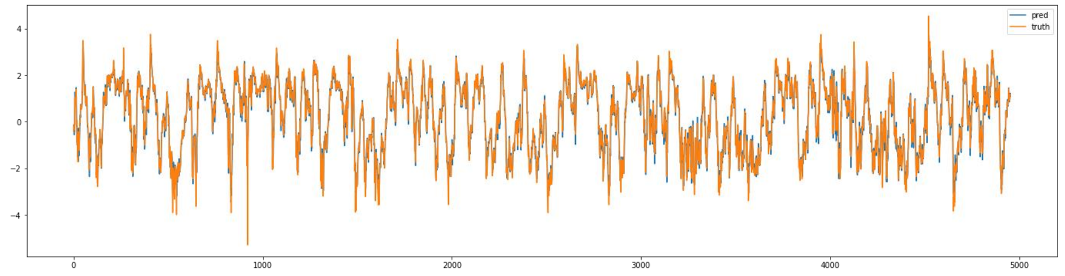


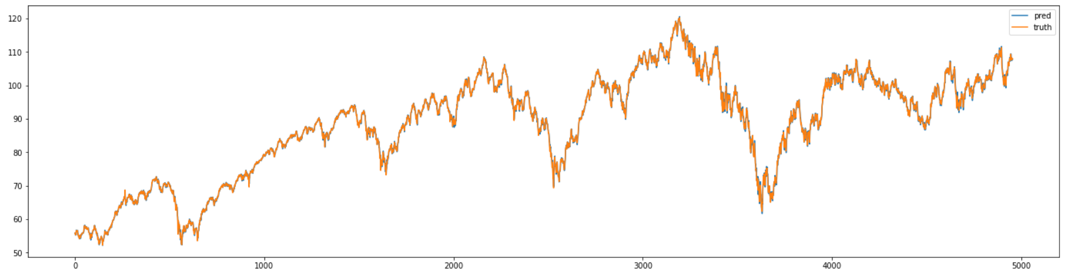
The prediction result is totally wrong. The max value in the predictions is 119.75696. So, the reason here is that the model has never met a value larger than 120 in train set. As a result, it doesn't want to predict a value larger than 120.

Optimized Network

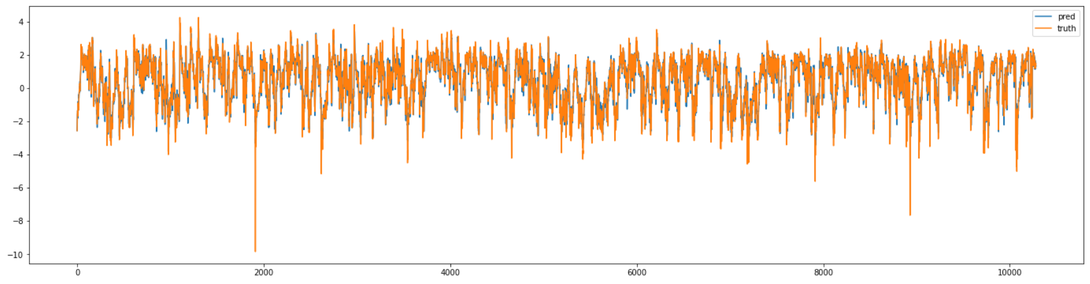
To optimize the network, I didn't modify the network's structure or parameters, because there aren't many parameters that we can changes. And for this prediction task, the input is not rich, I assume a more complex model may cause over-fitting.

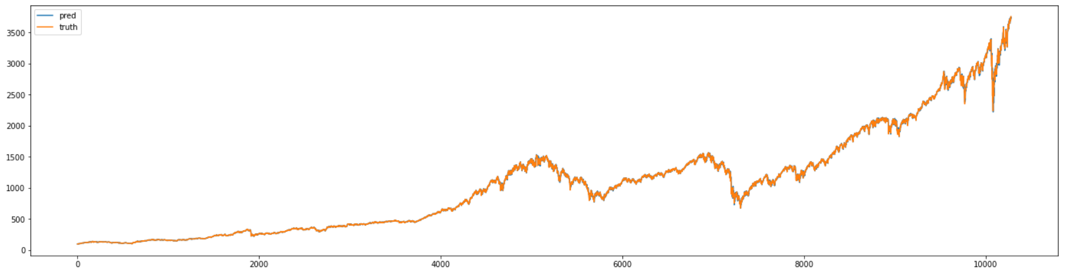
To fix the problem in last step, I do normalization operation in each slide window and record the mean and std for remapping the target-value in the future.





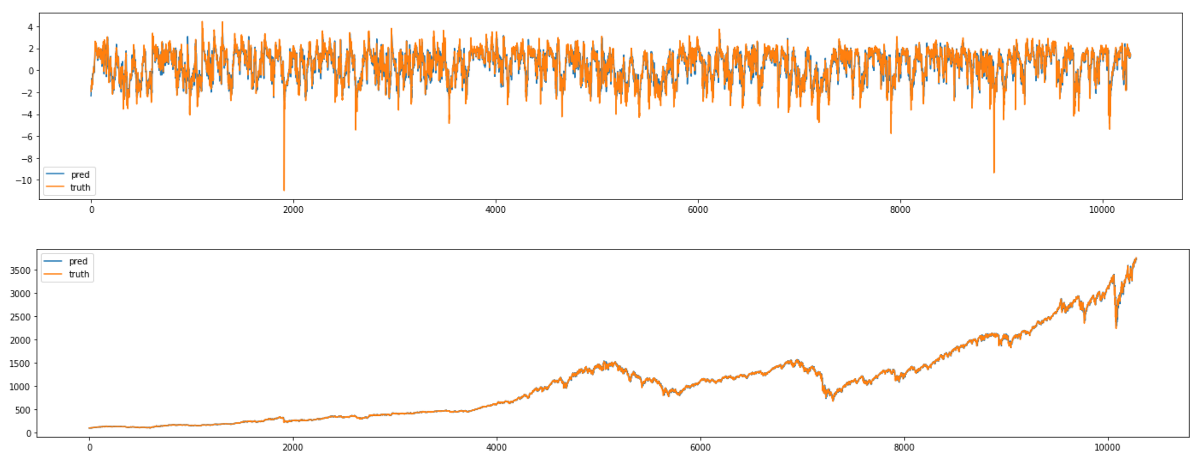
Above is the prediction result on training set. The first image is the predicted normalized value. The second image is the remapped target value.



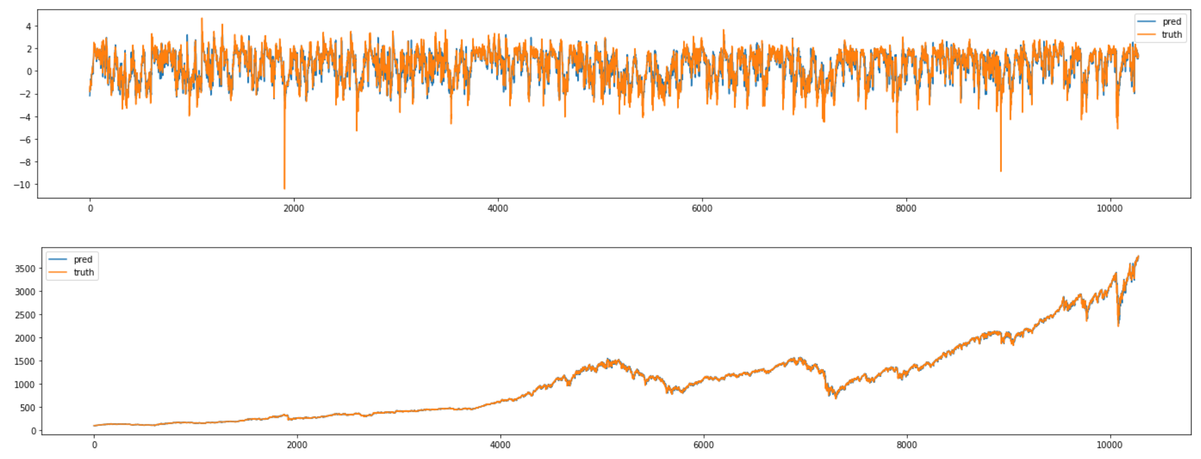


Above is the prediction result on test dataset (after 1980). The performance seems to be good.

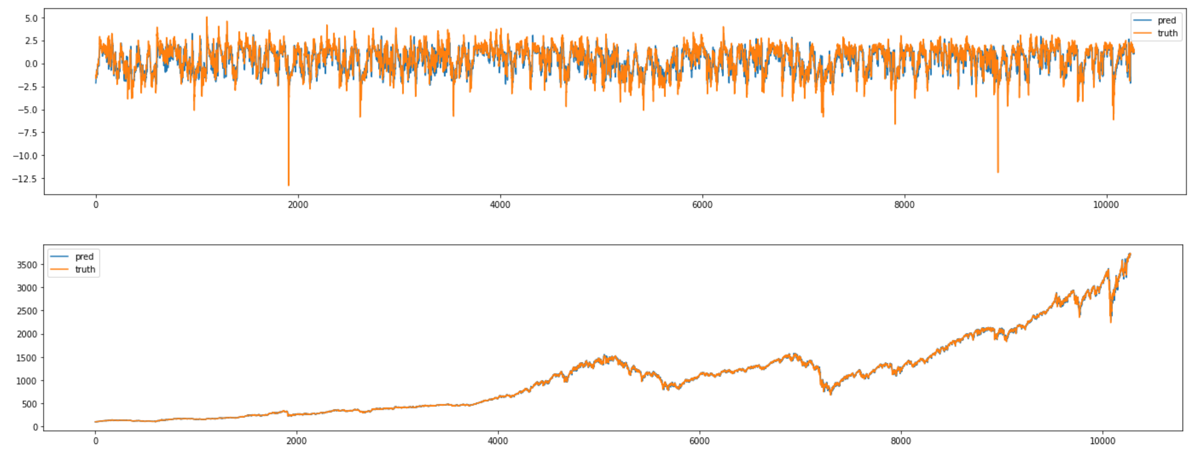
Then test the performance while predicting the next 2, 3, 4 values.



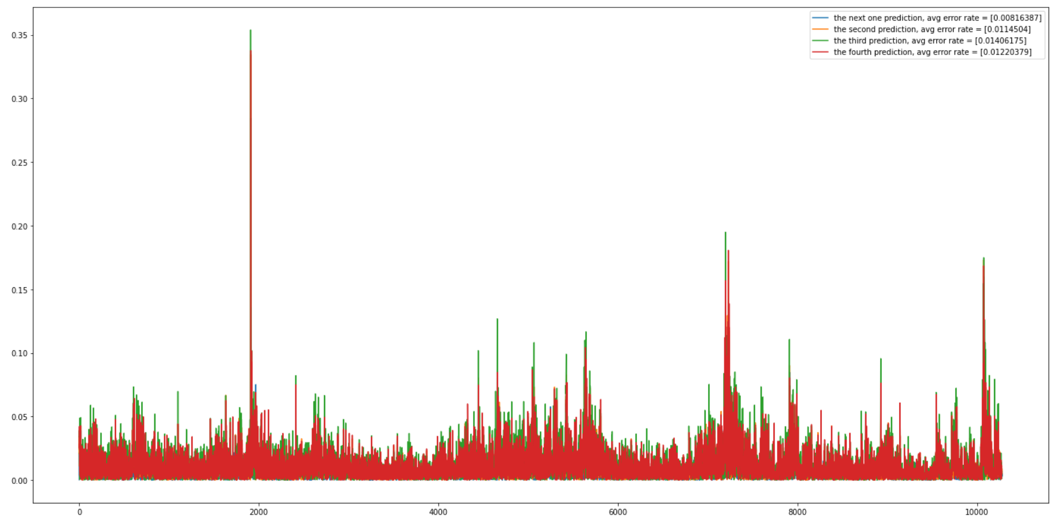
The next 2nd values predictions.



The next 3rd values prediction



The next 4th values prediction.



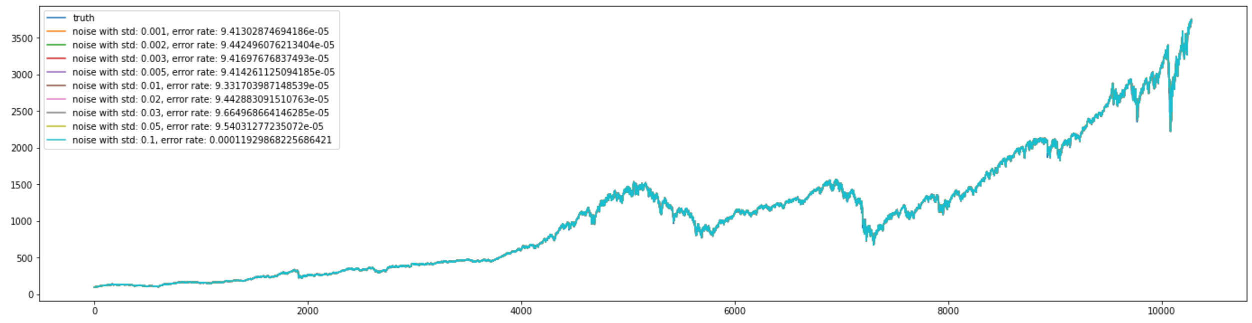
Here shows the error rate while predicting the next values.

With noise

Because we do the normalization before, so it is easier to add noise to the slide window.

```
for n_std in noise_std:

        new_sp_arr = []
        for i in range(len(data_test_x_rnn_arr)):
                sp_arr = data_test_x_rnn_arr[i].copy()
                indices = np.random.randint(Tx-4, size=int(Tx/10))
                for idx in indices:
                noise = np.random.normal(scale=n_std)
                sp_arr[idx, 0] += noise
                new_sp_arr.append(sp_arr)
        noise_sp_arr.append(np.array(new_sp_arr))
```

Do the prediction on those dataset with noise, and plot the predictions.



Discussion

The CNN model is used to detect the wave pattern, so key is to pay attention to the difference between two continuous values. To make sure the CNN can compute the difference, I set the kernel weights as [-1, 1] with a shape of (2, 1, 1) and keep it unchangeable. After training the CNN, we add it to the final model. Without the CNN, we just use the past sp500 value to make future prediction. With the

CNN added, besides the past values, we can get an extra information about the past values pattern. After training the LSTM, we can know the affection of the wave pattern on the single step output and furthermore the effect on the final prediction. We can not detect the wave pattern with a single of several CNN layers added before the LSTM, because the 'slide window' need to be overlapping. So I generate another input tensor and use TimeDistributed layer to map the wave pattern detection model onto this input tensor.

Besides the model's structure, how to deal with the data is also super important. As we can see, the sp value is increasing, so only training the model with the data 40 years won't work. Because the model didn't meet the larger values. Also, we can do the normalization on the whole dataset, because this will cause data leak, which mean we know some information about the future. So, to solve that, I did normalization in each slide window. In this way, the model is not paying attention to the future, and it can get an 'unscaled' training process. (By saying 'unscaled', I mean the parameters are not affected by the scale of the input data).