README

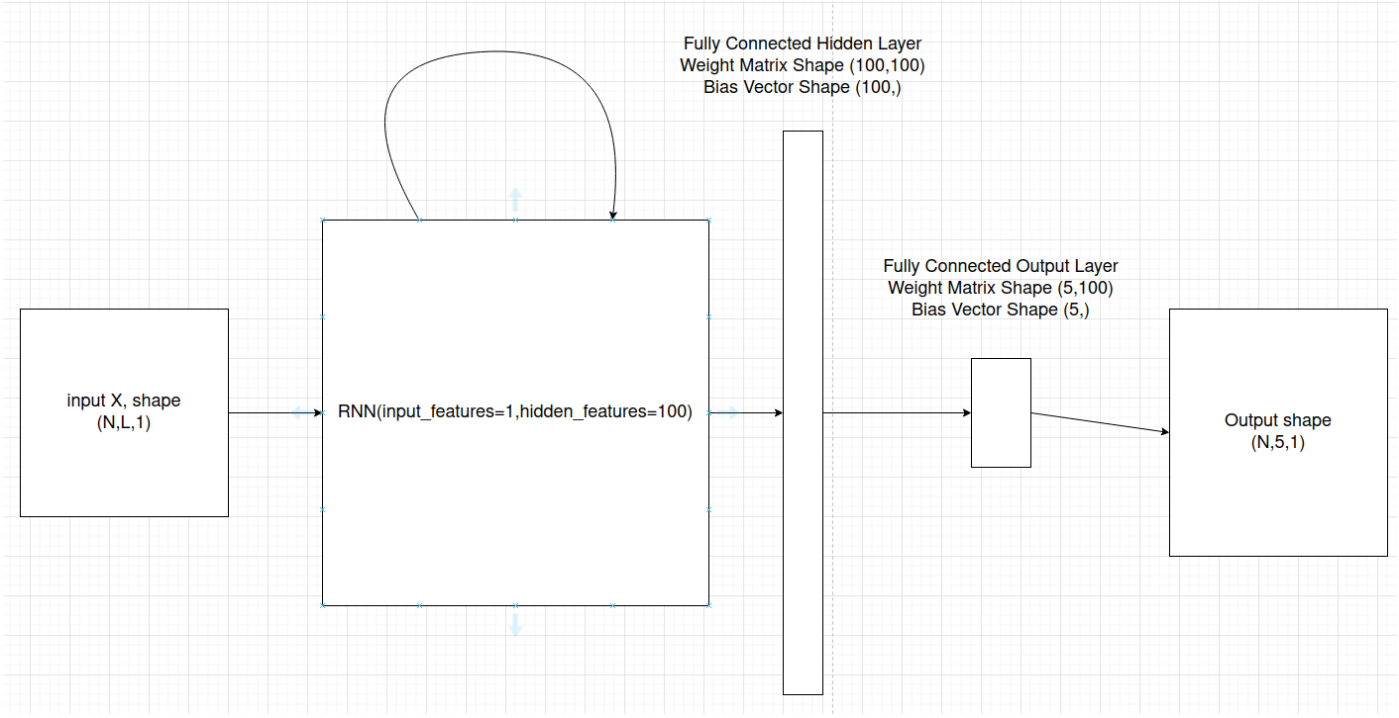
Introduction

Many individuals and groups trade the markets on a daily basis for the purpose of preserving or making money. Market participants range from individuals saving for their future, to professional trading firms aiming to maximize profits through capital gain. Engaging with the free market requires people to assume a certain amount of financial risk. However, in order to eliminate the financial risk, we aim to build a deep learning model that will predict prices of the stocks for the next 5 days (output), given that we have previous price data of the stock for the past 30 days (input).

In our project we use sequence-to-sequence architecture to produce price predictions using stock market data. For sequence processing, we use deep learning models such as Recurrent Neural Networks. With that being said, this repository contains the code for our project of exploring methods to forecast price sequences using stock market data. We use a RNN architecture to encode multi features price sequences and a generative architecture to forecast the following sequence.

A single input data tensor is of the shape (N,M); N days of M features of price data. The target data tensor is of the shape (K,L); K days of L features of price data that follows the input tensor (the N+1th day is the first day in the target tensor).

Model



The input data, X, has the shape (N,L,1) - N batched examples of length L, our optimal model uses close prices alone, so the features per sequence is 1. The inputs are first processed by a RNN, which produces a 100 dimensional encoding (states H_L) of each of the input sequences. These encodings are then processed by a 100 unit fully connected hidden layer followed by a fully connected output layer. The outputs are reshaped to produce the (N,5,1) model forecasts.

Model Parameters

The f**orecaster_fc_hidden** model contains 3 trainable layers: *encoder*, *hidden*, and *out*. The encoder layer is built using an RNN; the two variables that control the number of parameters are hidden_features and num_layers. Our RNN is typically set to only a single layer unless otherwise specified so generally the only trainable parameters in it would be of size hidden_features + bias. The next layer, fc_hidden is a linear layer and the number of trainable parameters should be encoder_hidden-features * fc_hidden weights + biases. The final output layer fc_out is another linear layer of dimensions fc_hidden x output_length * input_features this would represent the number of trainable parameters plus biases.

Our final model has an RNN with input_size=1 and hidden size = 100 has:

- 100 weights to encode each input of the sequence
- 100 biases for encoding the input
- 100x100 weights for calculating the effect of the previous timesteps hidden output
- 100 biases as well for calculating effect of the previous timesteps hidden output
- TOTAL = 10’300 parameters

The fully connected hidden layer has:

- 100x100 weights and 100 biases
- TOTAL = 10’100 parameters

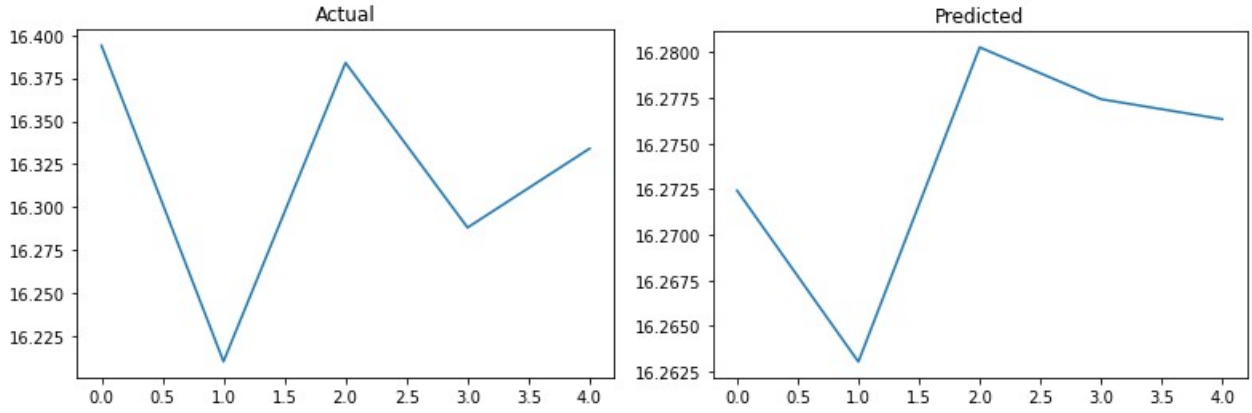
The output layer has:

- 5x100 weights and 5 biases
- TOTAL = 505 parameters

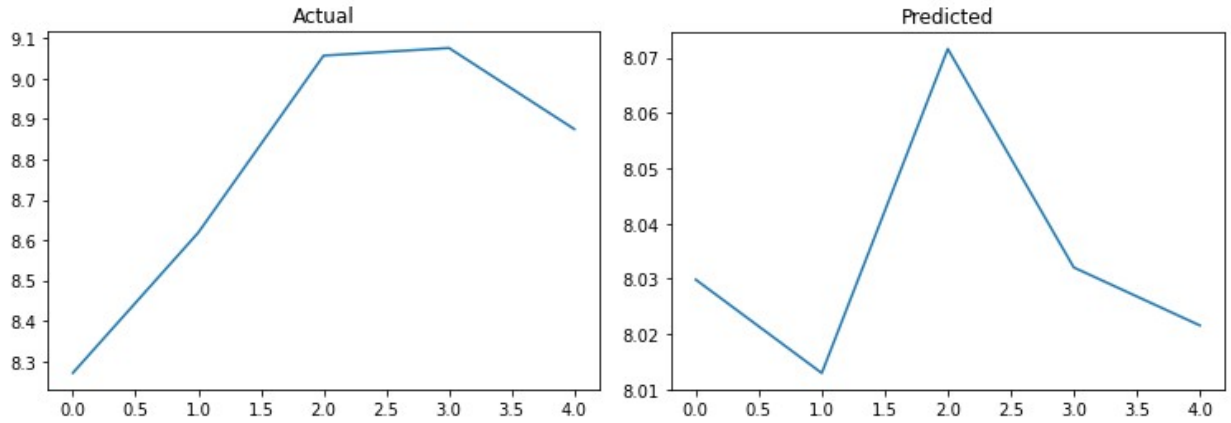
The final model has 10’300+10’100+505=20’905 trainable parameters.

Model Examples

Successful test example



Unsuccessful test example



Data

Data Source

Kaggle is a collaborative online platform which allows machine learning engineers to find, publish, and explore data sets as well as different build models. In fact, the primary source of data that is used in this project is the Huge Stock Market Dataset. This is a raw dataset that we extracted from Kaggle, licensed under the CC0: Public Domain License. With the Public Domain License, it allows free usage of this data for educational purposes which was leveraged in this project. To access the data set please refer to the link below: https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs.

Data Summary

The Huge Stock Market dataset contains daily price data for 1344 exchange traded funds (ETFs) and 7195 stocks trading up to the year 2017 from the NYSE and NASDAQ stock exchanges. Each day for a trading instrument has 6 data features: Open, High, Low, Close, Volume, and OpenInterest.

Previous statistics from our project proposal: The average 5 day nominal price difference of the target sequences is -67.77. The average 30 day nominal price difference is -488.91. The average intraday price move (Close price - Open price) is -5.99. These averages are important to note for the dataset, as it could hint to dataset bias that prices have generally moved lower over time, the model might learn this instead of the price patterns behind the trends.

Our task is to forecast a sequence of 5 days of price data in the future, the average 5 day nominal difference is important to note for this as a biased model may simply predict prices to go lower, which is not always the case.

In this repository, we train our model on ETFs price data, which has 2.5 million examples in total if considering inputs of length 30 and target outputs of length 5. Additionally, we focus on the price features of the data, Open, High, Low, and Close. We remove Volume and Open Interest features when processing the data. Not all 6 data features are useful for training the model since the Volume and the OpenInterest feature do not create a significant impact. Thus, the model is mainly trained on four features: Open, High, Low, and Close or a single feature: Close.

With that being said, one critical finding is that the model's performance was significantly improved with using a single feature rather than using four features.

Data Transformation

The data was processed into tuple pairs holding an input and target price sequences. Each data point is an (x,t) tuple pair where x is a tensor of shape (L,M) and t is a tensor of shape (Q,M). The target sequence consists of the timesteps following the last timestep of the input sequence.

The data was unzipped into multiple CSV text files, each one holding the dated daily price data for a trading instrument. To process the data into usable input and output tensors for model training, we read each file into a numpy array and separate out the dates from the rest of the data. The dates array is kept as a string datatype, the data array is cast as float32.

The data is then passed to processing functions that splice out a timeseries from a specific date range, which are then iteratively processed into (x,t) tuple tensor pairs. The shape of x is (L,M) and t is shape (Q,M), where L is the input length and Q is the target sequence length, M is the number of features for each day, chosen when initially loading the CSV files into numpy array through a use defined function argument (see function **load_price_data_into_numpy_array**).

See the functions **date_make_train_val_test_data** and **data_split_symbol_and_date** that produce the training, validation, and test sets based on either specific date ranges, or separate trading instruments for each set and date ranges within those.

We also augment the training data by generating new price sequences that match the interday proportional change of training examples (see the functions **augment** and **translate_price**). For example, if the close price of a training point is $10 for day 1 and $15 for day 2, an augmented version of this datapoint might be $20 for day 1 and $30 for day 2 (the 50% price increase between day 1 and day 2 is preserved). We augment in this way so that the model learns patterns from many price ranges even if the market has not traded at some price ranges before.

Lastly, we also included functions in the data_process.py file that normalized the processed data. Please refer to the functions **normalize_single_stock**, **normalize_as_avg_price**, and **normalize_train_data**. Though we found that non-normalized influenced the model to perform better in comparison to normalized data. Thus, we did not use normalized data across the training examples provided.

Data Split

Performing data split is an essential task that helps identify how well a model generalizes and how precise the generated predictions are. There are two main approaches that this repository showcases when splitting the data and that the models use while training data.

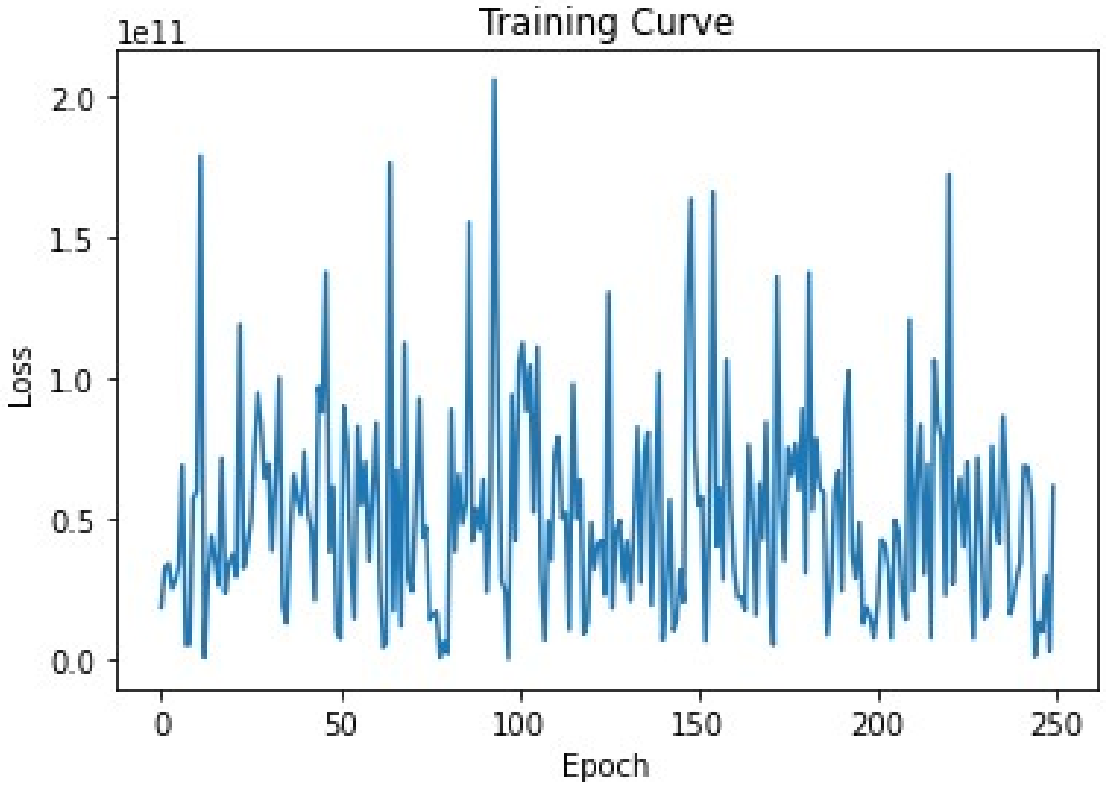
The first approach was to split the dataset based on date ranges. We trained the model on all ETF price data from the date 2010-01-01 to 2013-01-01 as the training set. Then the price data from 2013-01-01 to 2015-01-01 as validation data, and 2015-01-01to 2030-01-01 as test data (end dates exclusive). The idea behind this approach was so that the model would learn from all available price data for a given range, then be tested on data that would simulate future performance to best represent how the model would behave if it were deployed today.

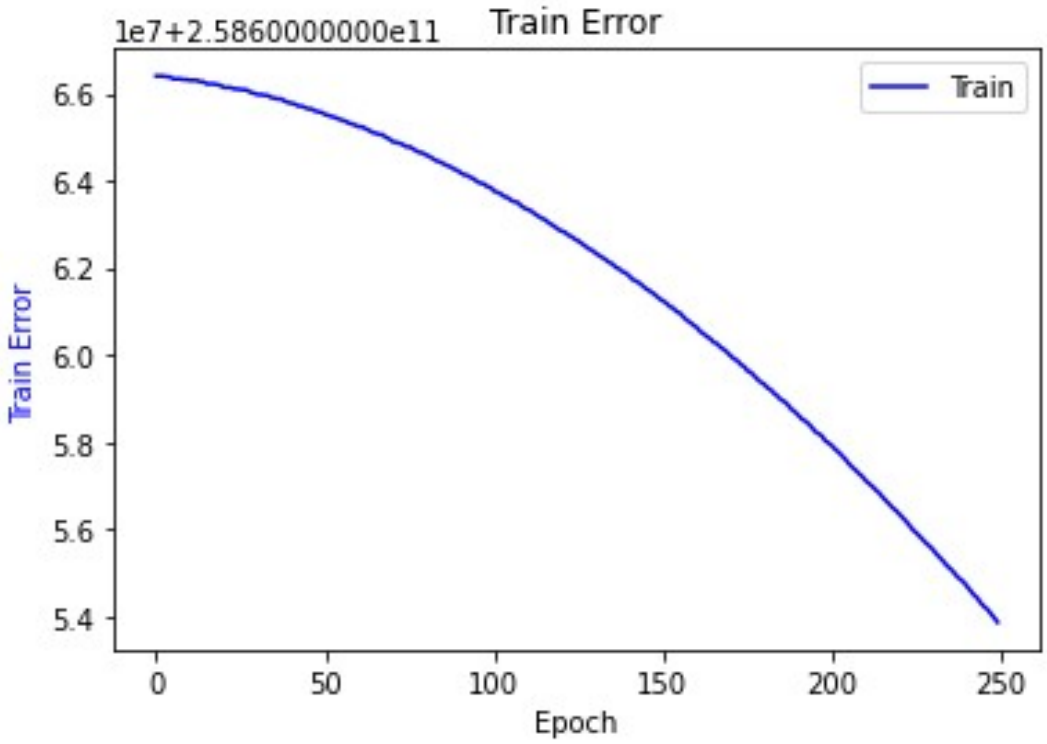
The second approach that was taken in order to split the data was through using the function **split_etfs** in the data_process.py file. This took in the entire ETFs data and split the data into three sets: test, validation, and training set. With that, 60% of the stocks were randomly allocated to the training set, 20% to the test set, and 20% to the validation set. To further specify, each set was passed through a processing function that spliced out the specific time-series from a specific date range. This allowed each set to include distinct stocks with distinct date ranges. By splitting the data by trading instrument and date range, we explored if learned price patterns are applicable between ETFs.

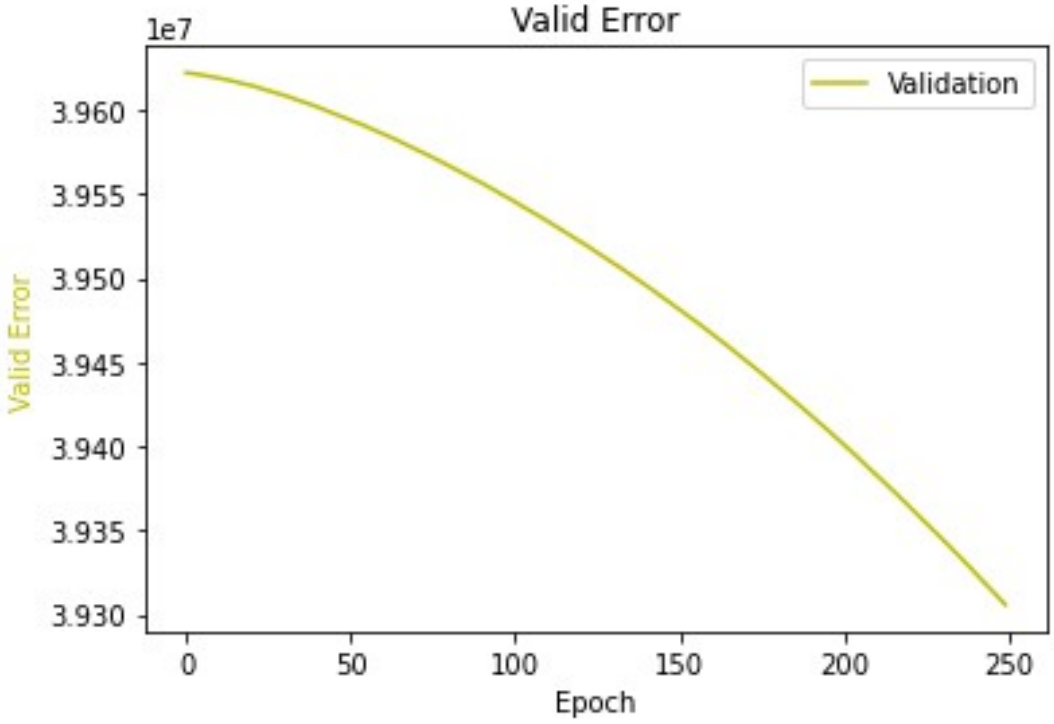
We found that both of the splitting approaches were beneficial and worked efficiently on small scaled data. On the other hand, neither worked effectively on a large scaled data. The former were due to the fact that the model performed better on small scaled data as in comparison to a large data set.

Training Graphs

Final Model – Forecaster_fc_hidden(1,100,100,1)

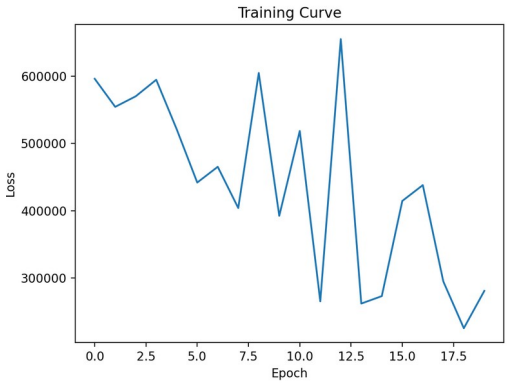


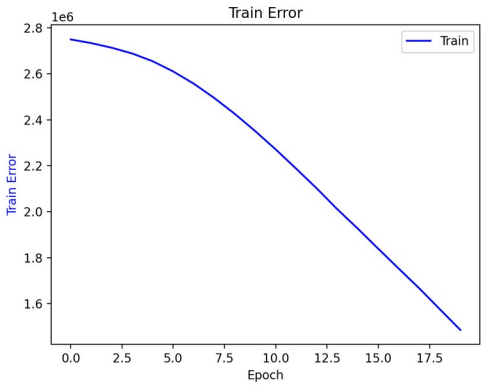

Train Error
1e7+2.5860000000e11
Train
6.6
6.4
6.2
6.0
5.8
5.6
5.4
Train Error
0
50
100
150
200
250
Epoch

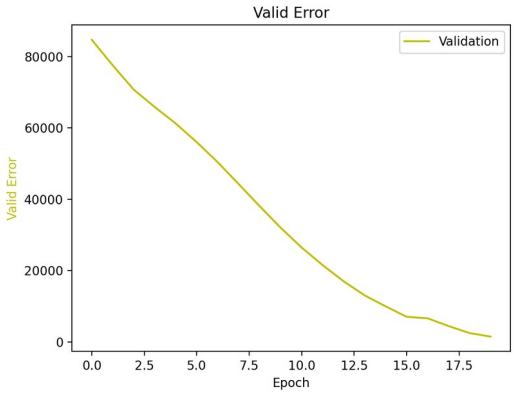

Valid Error
1e7
Validation
3.960
3.955
3.950
3.945
3.940
3.935
3.930
Valid Error
0
50
100
150
200
250
Epoch

Learning curves from models trained on small dataset

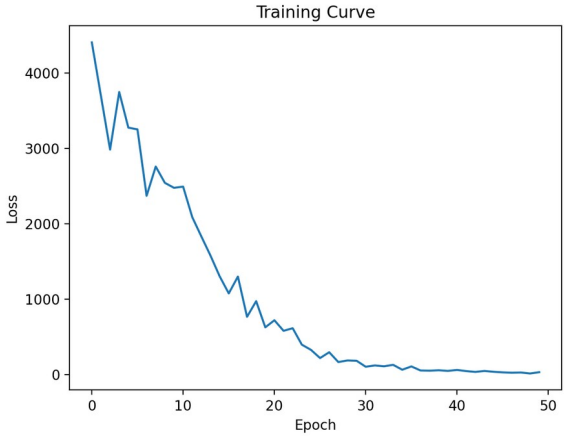
Forecaster_fc_hidden(1,150,75,1), learning_rate=0.001

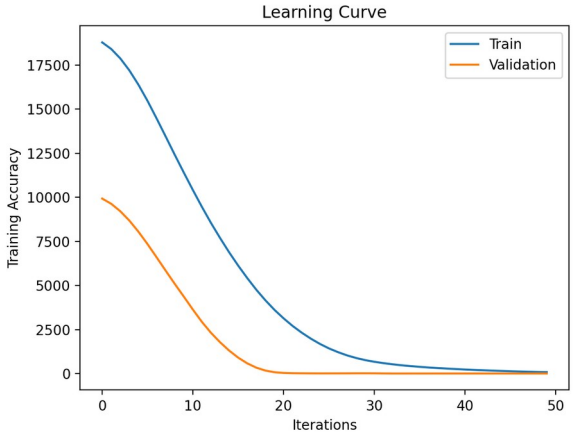




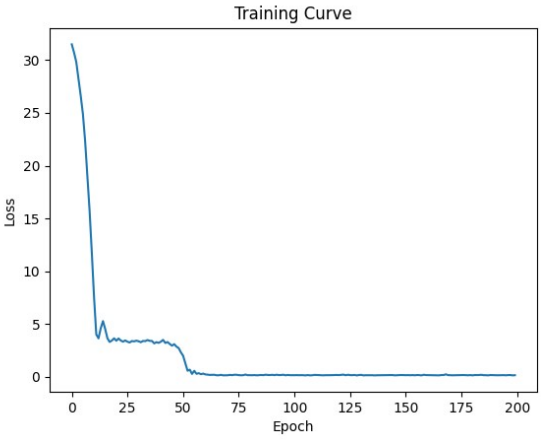


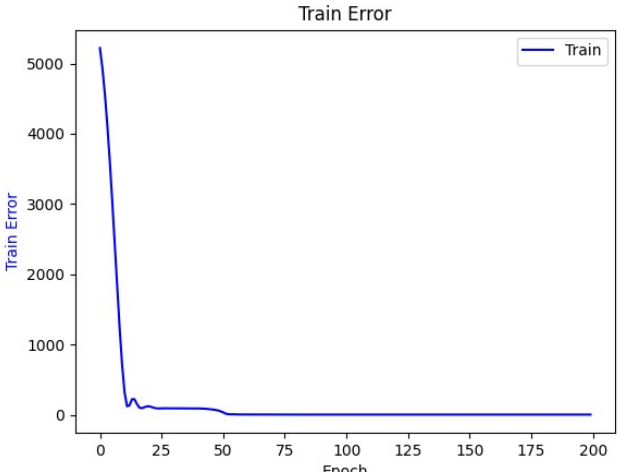
Forecaster_fc_hidden(1,100,150,1), learning rate=0.001

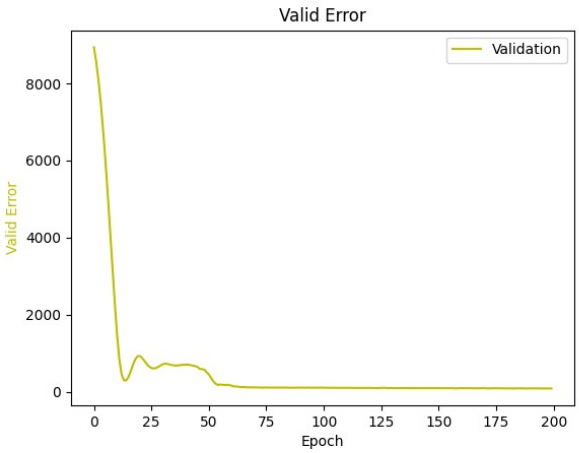




Example of model trained using only a single stock







Hyper-parameter Tuning

The four hyperparameters that were tuned in the Forecaster_fc_hidden model were the hidden layer size, number of layers, learning rate, and batch size.

For the hidden layer size we tried many sizes including 5, 10, 40 but found that they were too small to encode enough information and ended up giving very poor results. We ended up settling on a hidden layer size of 100 as we found that it was capable of containing enough information for it to make good predictions.

For the number of layers we used one RNN layer then two fully connected layers. We found that any less than two fully connected layers resulted in a model that didn't have enough expressive power.

For the learning rate we tried numbers from 0.01 to 0.000001 was too slow of a learning rate as we found that it did not result in convergence even after 250 epochs. The larger learning rates resulted in instability during training. We found that learning rates between 0.001 and 0.0001 worked

sufficiently well for small scaled data whereas the learning rate of 0.00001 worked slightly better for larger scaled data.

For batch size there was a large range of numbers that worked well. We found that anywhere from 1000 to 10000 worked reasonably well though lower batch sizes tended to begin with less validation error.

Results

Quantitative Measure

When looking at the results of our model, we used a number of factors to determine how well it was performing and where it might be failing. Our first choice was within the model itself, we began using an MSE loss function as that is what other RNN examples we studied had seemed to use. However while it did work we noticed our model's loss began at a very high value. This was likely due to our use of medium to large input values with a decent amount of variation causing the loss function to produce much larger values than it typically would with other sets of data. Upon further research and testing we determined Smooth1Loss in PyTorch produced much lower loss values and began to use it instead.

When testing our model we reviewed the normal statistics associated with neural networks, those being model loss, training accuracy and validation accuracy. Again due to the large magnitude of our input these values were generally larger than in most networks however both training accuracy and loss dropped to reasonable levels in later stages of parameter tuning.

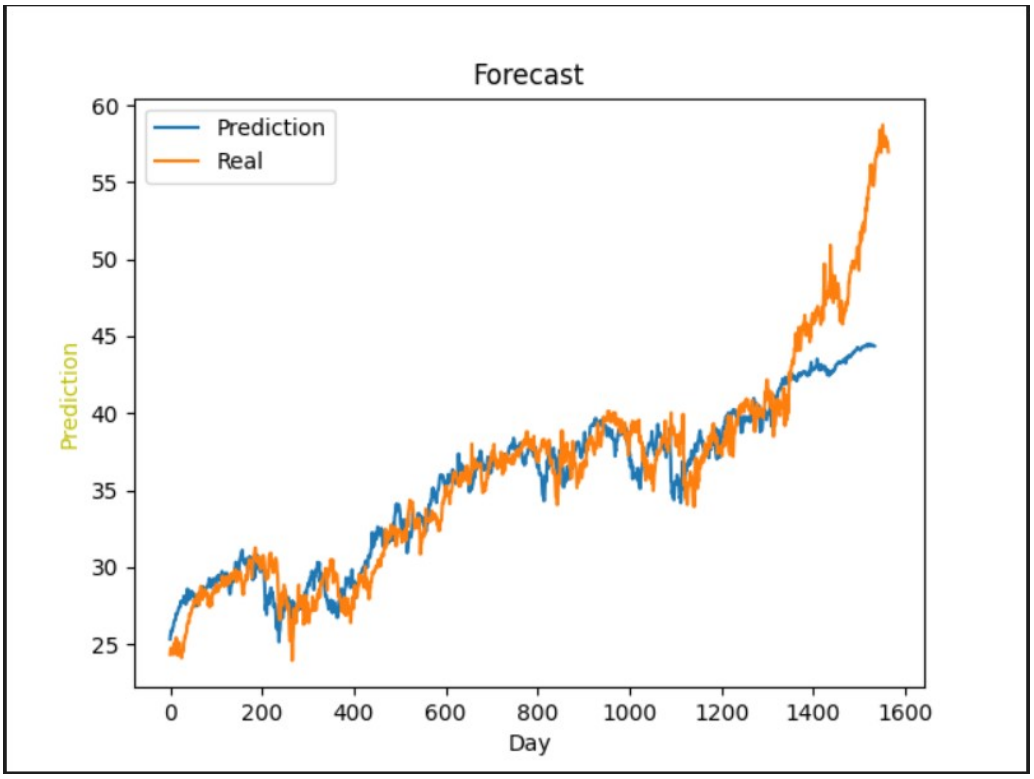
In addition to these factors we also tested and plotted the data and our models predictions after training had completed. While less strict it gave us insight into how our model was functioning and how it behaved on data it had not seen before. It allowed us to answer a number of questions about the model i.e. was it able to follow trends in data? Was it predicting in the correct value range? And overall gave us a broader picture of our model capabilities.

Quantitative and Qualitative Results

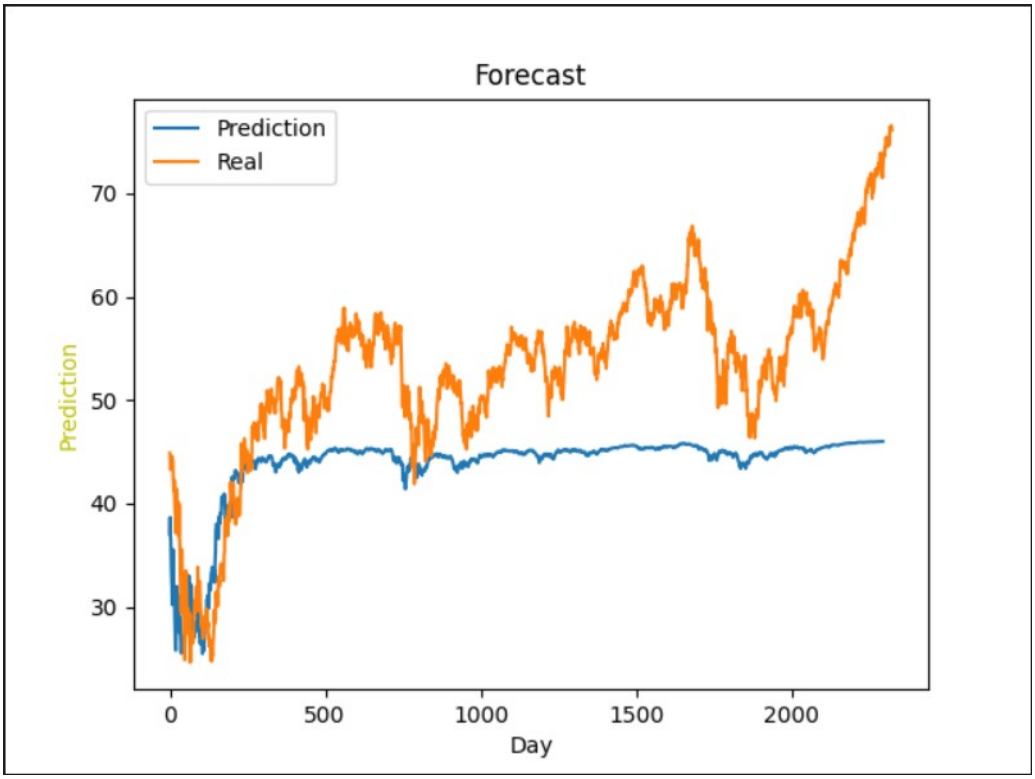
When training small subsets of the data, our model had enough capacity to memorize the training set and produce a smooth loss curve to convergence. However the loss curves created from training on the full dataset were very noisy with high nominal magnitude. We suspect the average error was so high on the full training dataset because the model was targeting 5 output features per input feature, each one contributing a squared difference to the loss for the given input. When the price difference between output and target was large, this increased significantly the average loss.

Despite the high error of the model, it was sometimes effective in forecasting what the correct overall pattern of a price series.

Magnitude corrected model forecast (great divergence at the end)



A completely failed example (even magnitude corrected, model pattern forecast is incorrect)



Our conclusion: the model worked partially:

- Worked: first predicted day was mostly in the price range of the input sequence.
- Didn't work: predicting subsequent days was less reliable.

Quantifiably from the model was not successful based on MSE loss, and convergence was not achieved on the full test data. However, the model was successful when training only on a single ETF or for small datasets where both the train error and validation error decreased. This shows some merit for price forecasting small examples from past data.

Despite the high model MSE, the model still had some predictive power at least for the first forecasted day following the input data. The price forecasted is at least somewhat close to the actual price for human standards.

Justification of Results

Initially, we tried an LSTM with batch normalization layer, and an RNN encoder to RNN decoder architecture. These models did not work very well and failed to overfit small training data sets. We hypothesize this was because the batch normalization layer made it difficult for the model to translate price back to the original price magnitude. We hypothesize the RNN-RNN encoder-decoder architecture was difficult to train because both portions of the network were RNN components, where processing occurs sequentially. The RNN-RNN encoder-decoder architecture struggled to learn in reasonable amounts of time.

Forecaster_hidden_fc - final model

The model in the end was able to perform reasonably well given the complexity of the data. While we encountered magnitude errors when predicting stocks of much larger dollar values the overall pattern and trend of the data was to some extent able to be predicted reasonably well by the model. This can be seen in the two above graphs in the qualitative and quantitative section. The training data is mapped very well and the test, while incorrect due to being much lower, the ups and downs were still accurate in time. The data summary shows the large price differences between the stocks which can be correlated with our models issue with high and low magnitudes in stocks.

Features play a key role when training models. In this repository we have trained either using one feature or using four features. After close analysis we conclude that using one feature, the close price, was more effective than using all of the 4 features: Open, High, Low, Close.This is because using more features increases the complexity of the model. To elaborate more, since the complexity increases, we also have more features that get encoded by the RNN layer. The more features, the higher complexity, thus causes higher error values.

As previously mentioned in the hyper-parameter tuning section, for our final model we chose to implement our model with 100 encoder_hidden_features, 100 fc_hidden layer, the learning rate of 0.00001, and the batch_size of 10,000.

While training our model, we have used both the Mean Square Error loss function and the Smooth loss function. Though they perform in a similar manner, our findings indicated that the initial magnitude of the MSE loss function was greater than the initial cost of the smooth loss function. Though, MSE seemed to be proportionally larger than the smooth loss function. So, despite the higher initial values, we continued to use the MSE loss function to train our data since it produces smaller cost values when the data is closer to its prediction in comparison to the smooth loss function.

Hypothesis for why the model does not perform well on the full training set, but does on small/single price series: The stock market as a whole is noisy data, trends and patterns exist (or at least seem to) when investigating select examples or the history of a single trading instrument. The model has sufficient capacity to memorize small datasets.

Ethical Considerations

Despite the somewhat lack of success of the model forecasting when trained on the full dataset, methods used in this repository may still one day find success in analysing stock market data. If such models were to be deployed, it could change the way individuals and institutions participate in the market, though with some ethical pitfalls to consider. The methods we deploy here primarily consider price features, the model itself does not know whether a stock or etf is managed in an ethical way, if individuals similarly do not consider the management either, they could end up financially supporting unethical practices or companies with values they disagree with.

Additionally, if an individual or group decides to trade based on the projected forecasts of the model here, there is no guarantee they will make or lose money on the trade, and no guarantee they will have mitigated risk based on confidence in the forecast.

If the methods deployed in this repo were successful, it is likely they would benefit stakeholders with access to computational resources much more than those without. This could lead to some stakeholders having a better edge when trading as they would be able to train their models faster, therefore incorporating and acting on new data sooner.

A notable limitation of this model is that it can only be trained on past price data, regardless of how good the performance may be on past data, there is no assurance that the future will hold similar price patterns. Thus the model's performance in the future will always be uncertain until the future becomes the past, then we could see how it performed retroactively, which still couldn't create confidence that the model would work on new future data.

Authors and Work Division

This repository is the product of collaborative effort between **Patrick Hogeveen**, **Maria Karim**, **Zehan Liu**, and **Arnaud Michel**.

The work break-down for model class and training code:

- Initial model and train code - written by Patrick and Zehan, reviewed by Maria and Arnaud.
    - Developed throughout the life of the project by Patrick, Maria, Zehan, and Arnaud.

The work break-down for data processing code:

- Data processing code - written by Maria and Arnaud, reviewed by Patrick and Zehan.

Hyper-parameter tuning was performed by all group members: Patrick, Maria, Zehan, and Arnaud.

The work breakdown for the README write-up:

- Introduction: Maria
- Model Figure: Arnaud and Patrick
- Model Parameters: Arnaud and Patrick
- Model Examples: Arnaud and Patrick
- Data Source: Maria
- Data Summary: Maria
- Data Transformation: Arnaud, Maria
- Data Split: Arnaud, Maria
- Training Curve: Arnaud, Maria, Patrick, Zehan
- Hyper Tuning: Zehan
- Quantitative Measures: Arnaud, Patrick
- Quantitative and Qualitative Results : Arnaud, Patrick
- Result Justifications: Arnaud, Maria, Patrick, Zehan
- Ethical Consideration: Arnaud
- Authors: Maria
- Advanced Concepts: Arnaud, Maria

Advanced Concept Used

Data Augmentation

As described in the data processing section, data augmentation was used to produce price sequences with different nominal values but the same inter-day proportional change as data points within the test set.

Sequence-to-Sequence Architecture

As mentioned in the introduction and model figure section, we have used sequence-to-sequence architecture. This includes our model accepting sequence inputs and outputting a prediction of sequence outputs. Additionally, the encoder mechanism is also embedded in our architecture which takes the sequence of inputs and encodes them.