Design Overview and Testing Documentation

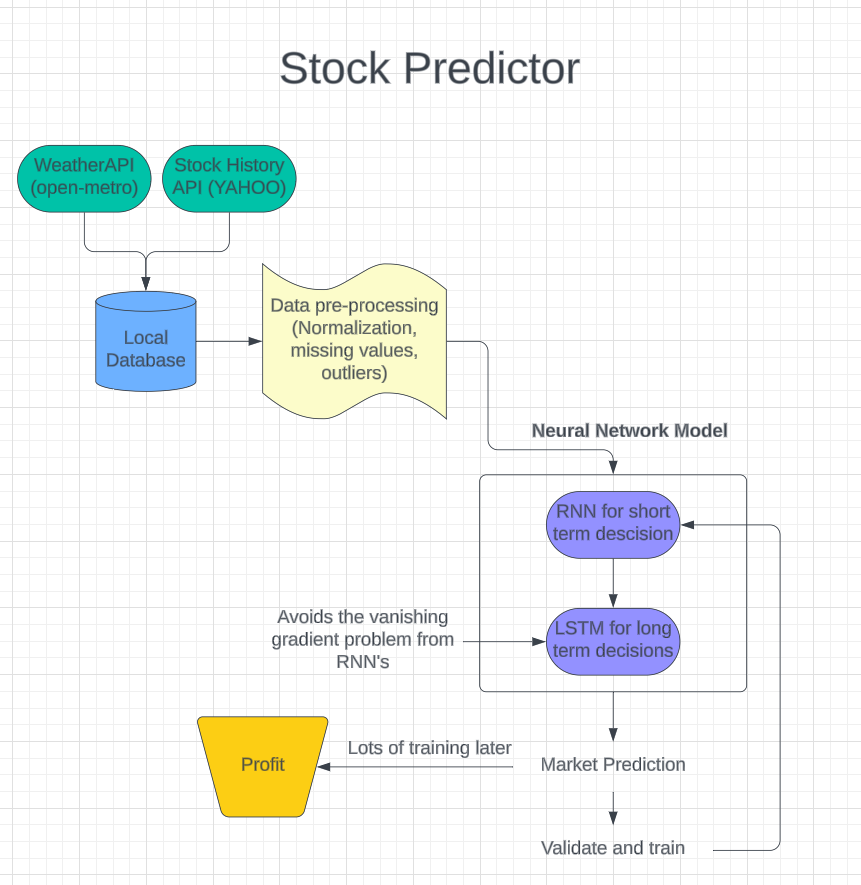
Stock Market Analysis

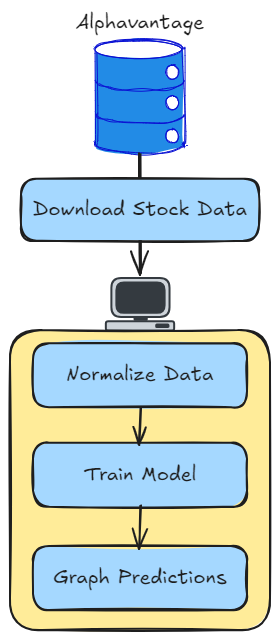
Recap

For the project, we will develop a neural network to analyze stock market trends using parallel computations to optimize the training process. Stock market trends are inherently complex, with patterns influenced by numerous factors, including market sentiment, economic data, and historical performance. A neural network offers a sophisticated means to model and predict these trends by learning from large datasets, recognizing patterns, and making projections based on historical data.

1 Design Overview

1.1 General design diagram





From the above diagram the flow of the model will be

1. Using APIs to gather stock data
2. Pre-processing the data (as necessary) for the Neural Network Model
3. Training the model

1.2 Data Collection

Data collection will be done using APIs such as Kraggle or Alpha Vantage. Kraggle does not require any data preprocessing where Alpha Vantage does. Below is an example of pulling in data using either AlphaVantage or Kraggle.

data\_source = 'kaggle' # alphavantage or kaggle

if data\_source == 'alphavantage':

# ====================== Loading Data from Alpha Vantage ==================================

api\_key = '<your API key>'

# American Airlines stock market prices

ticker = "AAL"

# JSON file with all the stock market data for AAL from the last 20 years

url\_string = "https://www.alphavantage.co/query?function=TIME\_SERIES\_DAILY&symbol=%s&outputsize=full&apikey=%s"%(ticker,api\_key)

# Save data to this file

file\_to\_save = 'stock\_market\_data-%s.csv'%ticker

# If you haven't already saved data,

# Go ahead and grab the data from the url

# And store date, low, high, volume, close, open values to a Pandas DataFrame

if not os.path.exists(file\_to\_save):

with urllib.request.urlopen(url\_string) as url:

data = json.loads(url.read().decode())

# extract stock market data

data = data['Time Series (Daily)']

df = pd.DataFrame(columns=['Date','Low','High','Close','Open'])

for k,v in data.items():

date = dt.datetime.strptime(k, '%Y-%m-%d')

data\_row = [date.date(),float(v['3. low']),float(v['2. high']),

float(v['4. close']),float(v['1. open'])]

df.loc[-1,:] = data\_row

df.index = df.index + 1

print('Data saved to : %s'%file\_to\_save)

df.to\_csv(file\_to\_save)

# If the data is already there, just load it from the CSV

else:

print('File already exists. Loading data from CSV')

df = pd.read\_csv(file\_to\_save)

else:

# ====================== Loading Data from Kaggle ==================================

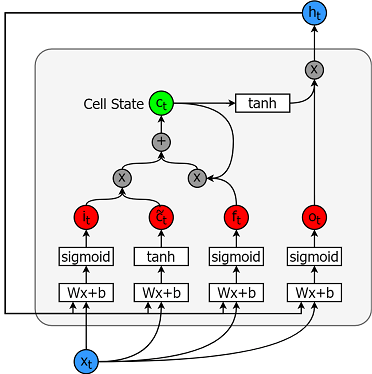
# You will be using HP's data. Feel free to experiment with other data.

# But while doing so, be careful to have a large enough dataset and also pay attention to the data normalization

df = pd.read\_csv(os.path.join('Stocks','hpq.us.txt'),delimiter=',',usecols=['Date','Open','High','Low','Close'])

print('Loaded data from the Kaggle repository')

1.3 LSTM Cell Design



The above diagram provides an overview of the LSTM cell that will be used to build the model.

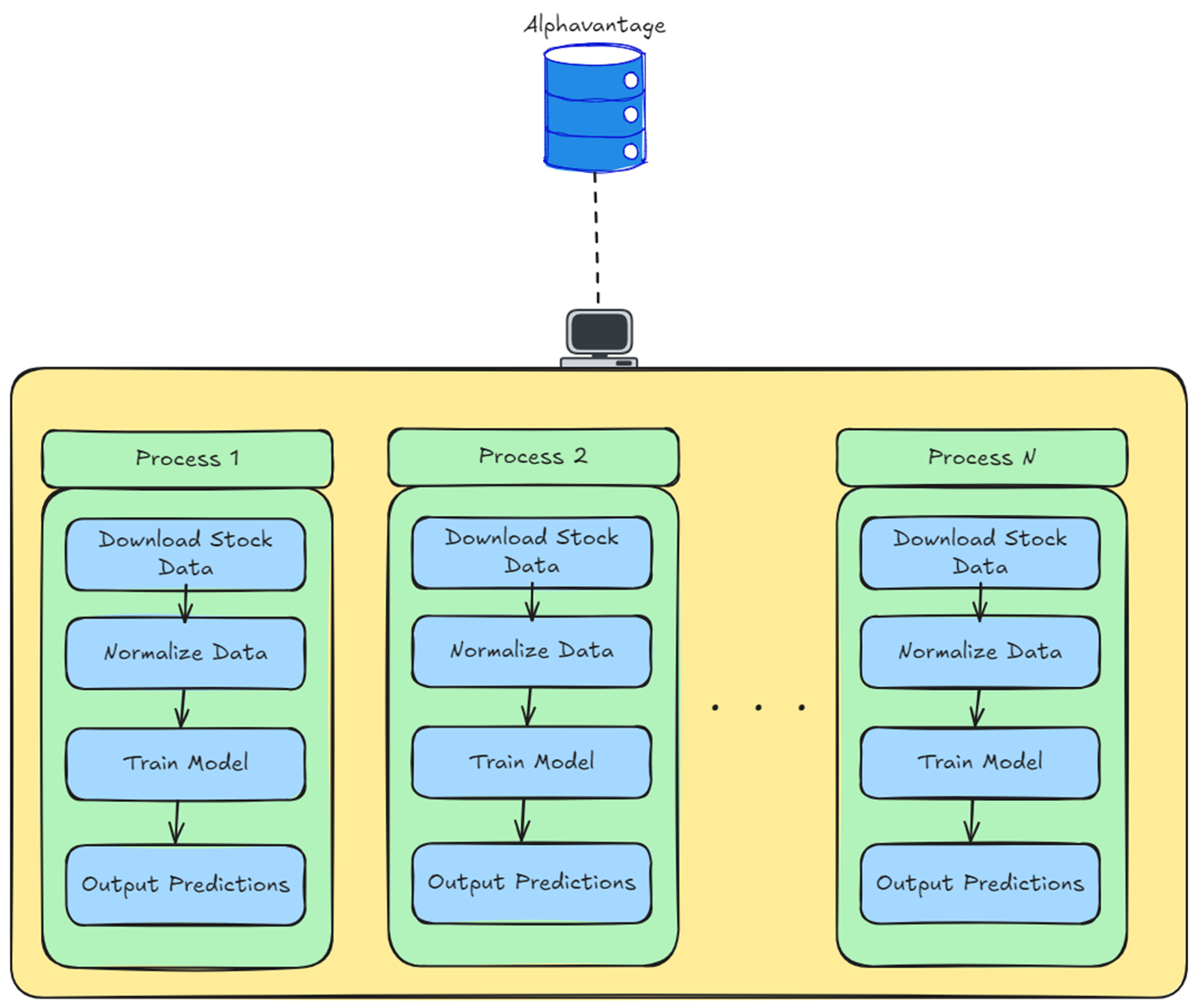
* Cell state (ct) - This represents the internal memory of the cell which stores both short term memory and long-term memories
* Hidden state (ht) - This is output state information calculated w.r.t. current input, previous hidden state and current cell input which you eventually use to predict the future stock market prices. Additionally, the hidden state can decide to only retrive the short or long-term or both types of memory stored in the cell state to make the next prediction.
* Input gate (it) - Decides how much information from current input flows to the cell state
* Forget gate (ft) - Decides how much information from the current input and the previous cell state flows into the current cell state
* Output gate (ot) - Decides how much information from the current cell state flows into the hidden state, so that if needed LSTM can only pick the long-term memories or short-term memories and long-term memories

1.4 Parallelization

Parallel computations will be used to improve the efficiency of the model. Specifically parallel computations can be used to reduce training time and data processing time. The diagram in section 1.1 shows a serial design of the model.

1.4.1 1 Stock per process

The first possible parallel implementation is to split the serial model to analyze one stock data set across multiple processes. Each process would run with its own CPU resources. Below is an example of what this model would look like.

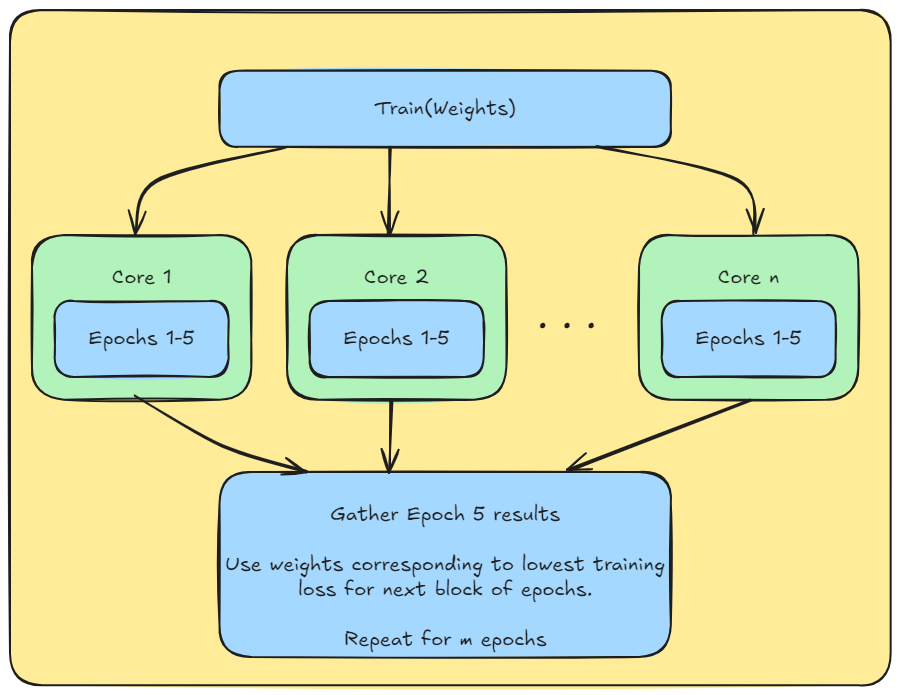


Each process is responsible for retrieving its own stock data, preprocessing the data, training with the data, and outputting predictions. Across a larger pool of stocks this approach would yield efficient speedups.

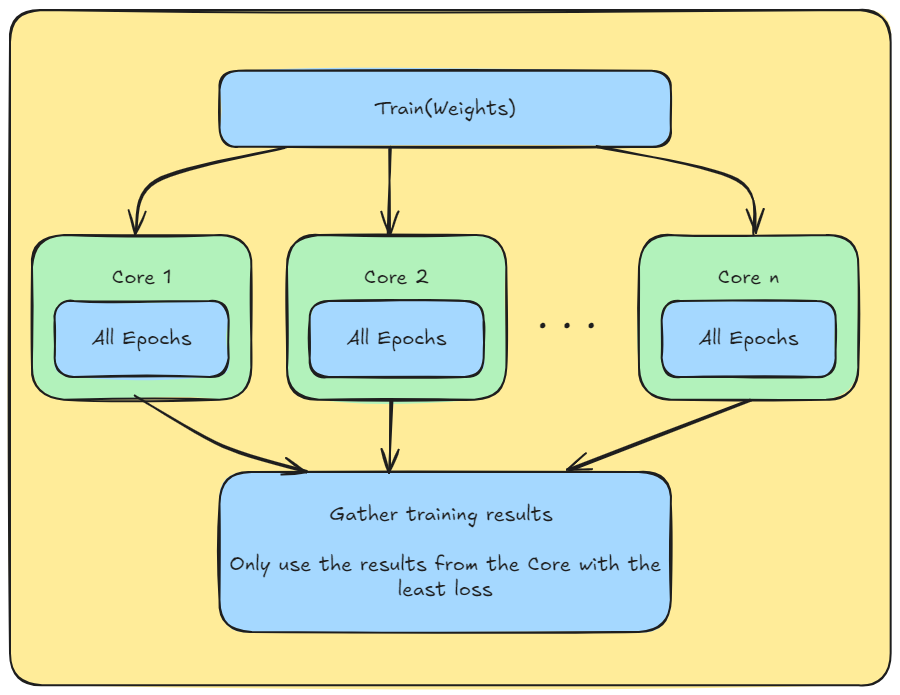
There are additional parallelizations that can be done for each process. The data normalization & pre-processing involves scanning a CSV file and making adjustments without data dependencies on other sections of the file. This allows for an OpenMP for loop to split the task among multiple cores.

1.4.2 Divide training among cores

It is possible to parallelize the training process, that will require some more careful design. The Neural Network training process involves training cycles called epochs that adjust weights every epoch according to a loss function. The output of one epoch influences the inputs of the next epoch. This creates a data dependency that does not allow for full parallelization. It is possible to allow P cores to run a small number of epochs each and take the best result to influence the next sequence of training, but this just results in P-1 \* epochs being thrown away.

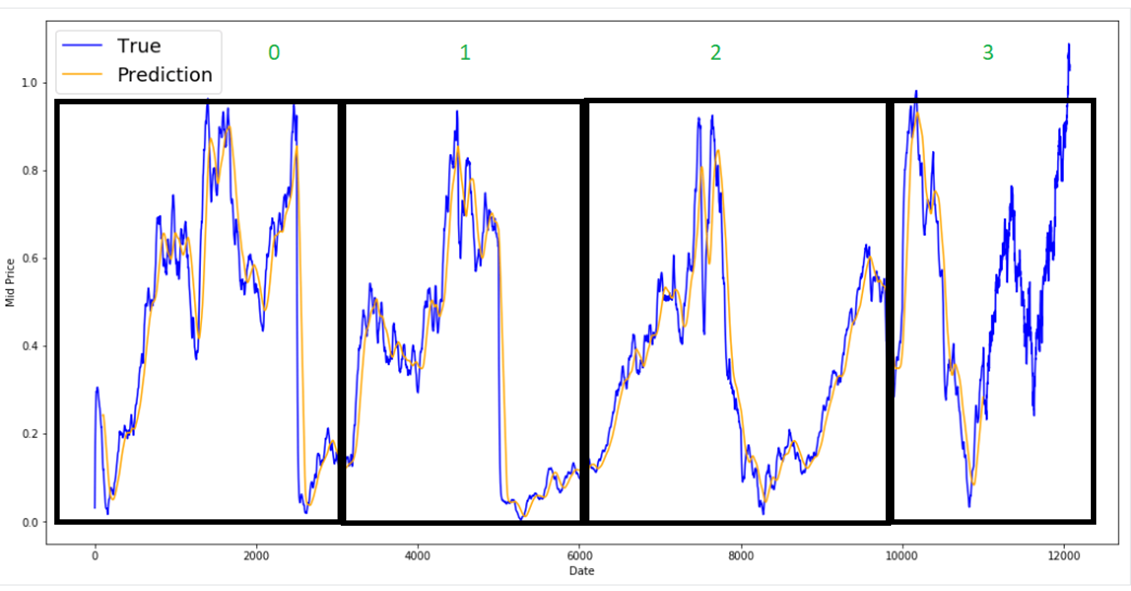


As you can see from the proposed model, only 1 sequence of epochs training is kept. Additionally, the model requires synchronization after each block of epochs. This introduces some serial code which will reduce the speedup. The alternative is to not perform block training and just allow each core to run every epoch and just use the best result.



This does allow more efficient training as we increase the probability of a lower loss at the last epoch. However, if we are using the model to analyze a large number of stocks in parallel the CPU resources can be better used to run each stock analysis on a single core in parallel.

1.4.3 Divide stock data among cores



From the above graph you can see a graph of how a stock price fluctuates over time. In the serial implementation the model would train 80% of this data & validate with the remaining 20%. A parallel approach in this case could be to divide the data set of this stock by providing each core p with a partial data set.

This approach has multiple issues. The first is that a smaller data set means less reliable training & validation. Reduced model accuracy as the LSTM is more efficient when training over large sets of time series data.

However, it is worth investigating this strategy as it is possible that considering Alphavantage has 80+ years of stock data, 1 core would not be able to handle stock data over decades.

2 Training

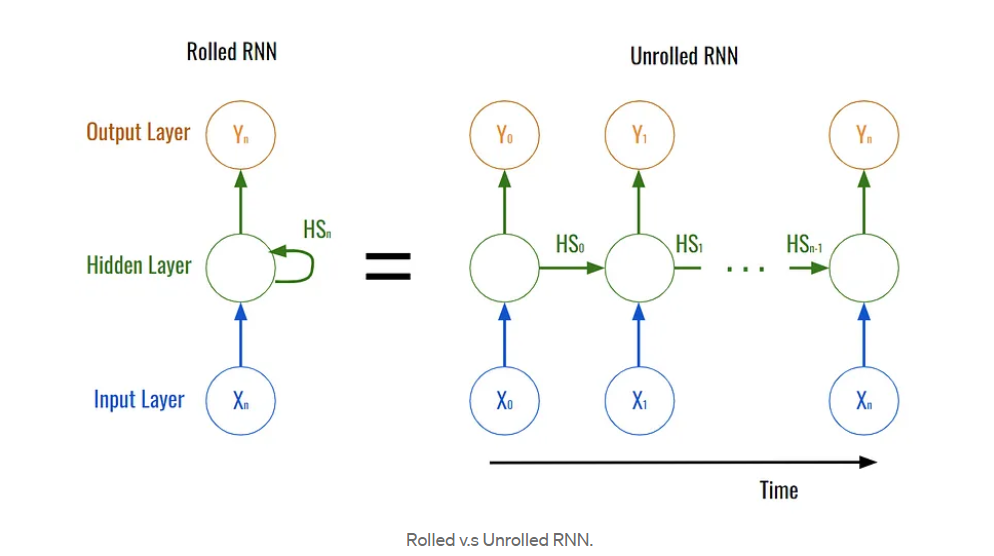
Building the set to train the model will consist of three separate steps, firstly we will pull data from the open-meteo weather API for a specific day in history, after that we will pull in the stock data for that day using the YAHOO stock history API. Once both of these steps have completed we will pull all of the collected information into a local database where it will be processed. Processing in this case basically just means that we are adding the weather and stock data as feature variables with whether or not the stock will increase or decrease as the variable the network is trying to predict.

Since we are using both an LSTM and RNN we will need to train two models in parallel, to do this we will start by splitting the data that we collected from a training set as well as the actual test set. Once this is done each network will be trained at the same time using different methods which are listed below

RNN and LSTM training

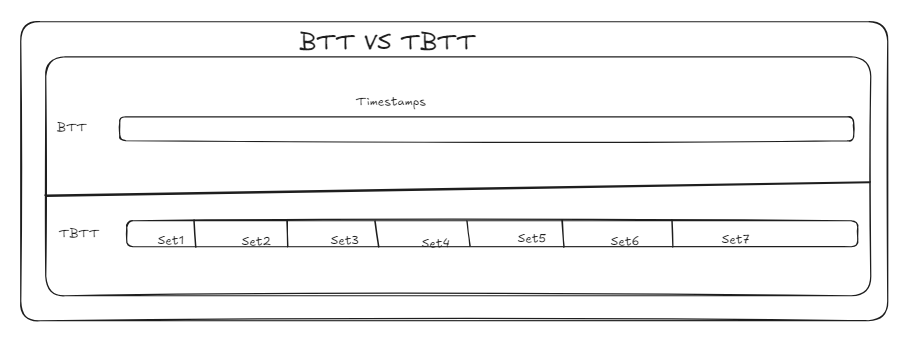
There are two ways that we can train our RNN, firstly we could use Backpropagation Through TIME (BTT) or Truncated Backpropagation through time> Benefits, disadvantages and how they work will be explained below

**BTT:** BTT works on a modified version of the standard Backpropagation algorithm, one of its main differences from standard backpropagation is that it is primarily used on time sequenced data like the data we are predicting in this project. To implement BTT we would first start by presenting the network a series of timestamped data as the training set, these timestamps consist of an input, a copy of the network and an output. Once all of these timestamps are passed into the network we would unroll the network and calculate error across each timestamp and put them into a list. Finally we would roll the network back up and use the accumulated error from before to update the network weights. It is important to note that all of the timestamps can be seen as separate layers in the network, with this being said unrolling the network is simply sending the data through every timestamp and rolling the network is simply sending the timestamps through the original layers of the network, an image of this can be seen below



The biggest disadvantage with the BTT algorithm is just how costly it is to actually use, since each timestamp acts as a separate layer when unrolling the network things can quickly balloon to a scale that is too large to test in a timely manner. As an example if we passed in one thousand timestamps to the network then we would have one thousand derivations for a single weight update, this can be mitigated with the Truncated Backpropagation Through Time algorithm discussed below

**TBTT:** TBTT is an extension of the standard BTT algorithm, the main thing that TBTT changes is how the weights are updated. Instead of unrolling and rolling the network every time a timestamp runs through it the TBTT algorithm only rolls and unrolls the network to update weights after a user specified number of timestamps have passed through the network, here is a more detailed explanation of how the algorithm runs. Firstly we present a sequence of k1 timestamps to the network, next we would unroll the network and let k2 timestamps go through it, calculate error for each timestamp and save it to a list, finally we would roll up the network and use the error to calculate the new weights. This sequence will be repeated until the testing set has run out of timestamps to test. As an example of how this algorithm would work let's take a look at the one thousand timestamp long example from above, in this algorithm you could break that one thousand element long list and split it into fifty individual training cases, these cases would each be about 20 characters long. This would greatly sped up training but may have an effect on how well the network sees dependent data like the weather. A picture can be seen below of how these models handle larger datasets



As we can see from this picture the BTT algorithm does not break up sets of timestamps and updates the weights with every timestamp given while the TBTT algorithm breaks up the larger input into more manageable chunks and then updating the weights at the end of every chunk, the one we use will ultimately be determined by how large our dataset ends up being in the end, Everything that has been mentioned in the training section will also work for the LSTM as well since the LSTM is a more complex version of the already existing RNN

Uses for RNN and LSTM

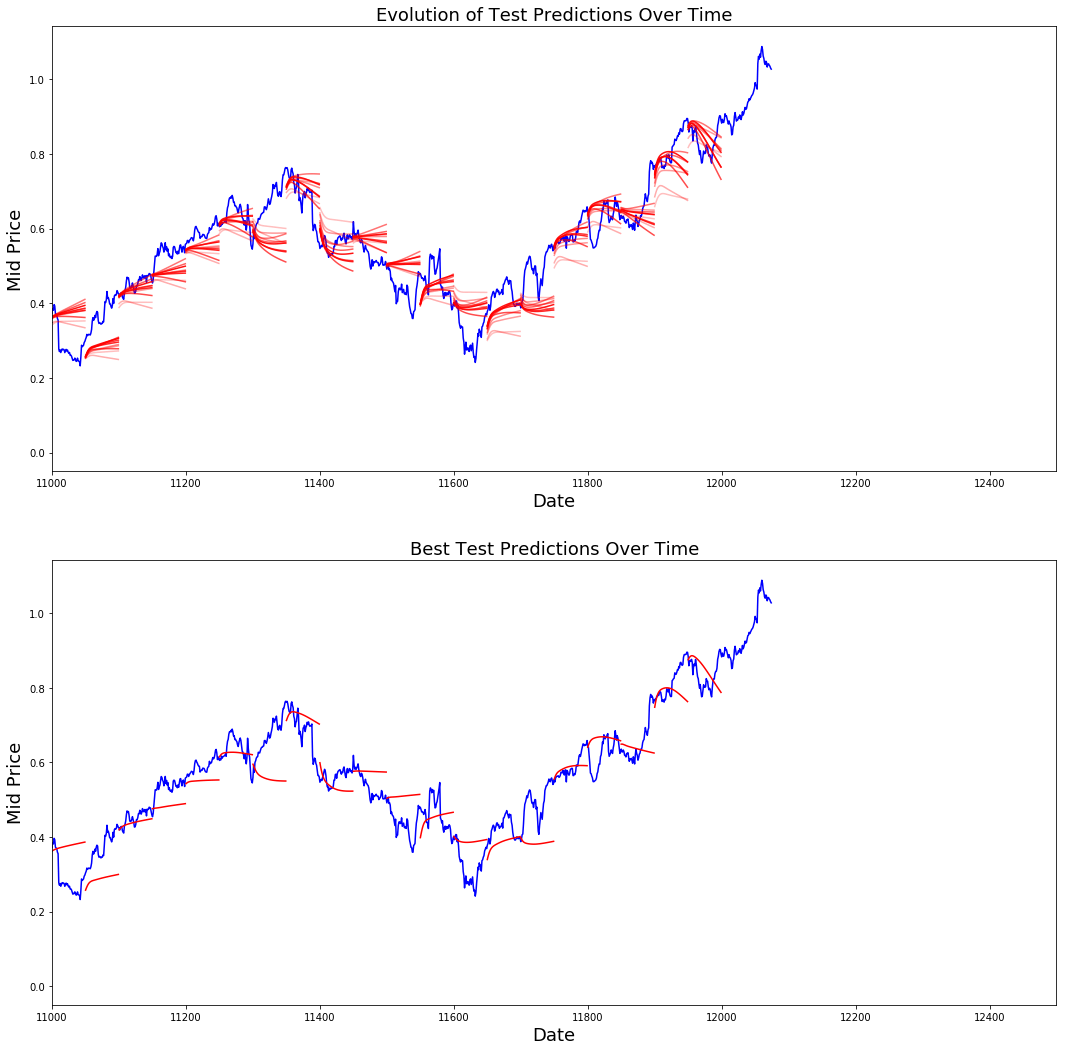
Both of these neural networks are very useful for different applications, the RNN for example is very good at taking data and using it to predict things in the short term, the RNN struggles for long term data predictions though due to things like the exploding gradient problem. The exploding gradient problem occurs during the training of the neural network and can happen when the gradient of the weights in respect to the loss function get to be to large, this can cause serious issues with how the network predicts things, in contrast the LSTM does not have to deal with the exploding gradient problem. The LSTM excels at long term data prediction due to its use of long term dependencies that help associate specific factors in the data to the output node, these long term dependencies make the network far harder to implement than an RNN. Another downside of the LSTM is that it is extremely easy to overfit the data to the network, this can lead to inaccurate predictions over a longer period of time.

3 Testing

3.1 Neural Network Testing

After training the model on a sample data set, the testing process will involve using the historical data of other stocks across different markets to verify the accuracy of the model. Using python packages we can create graphs of the actual price changes of the stocks and overlay them with the predicted prices from our model. This allows us to visualize the accuracy of the model.

Two relevant graphs are showing the predictions over time (through each epoch) versus a best prediction graph. Below is a sample from the datacamp article.



Next, we will import stock data across different time periods & different markets and validate the models accuracy across a wider data distribution. Using the MSE for the predictions of a given data set we can plot the accuracy of the model across different factors. Using linear regression we can determine if either the market or time periods has a significant effect on the model's accuracy. After determining the accuracy of the model and making adjustments, we can then formulate an investment strategy and test the model against today’s market using fake money.

3.2 Parallel Testinge

Training the parallel computations involves implementing the approaches defined in section 1.4 & testing the speedup / efficiency over multiple cores. Results will be documented using plots. Additionally, since we have proposed prototyping the model in python & implementing the final design in C to allow OpenMP usage we can test the times of running the model in both cases.

References

<https://machinelearningmastery.com/gentle-introduction-backpropagation-time/>

<https://k21academy.com/datascience-blog/machine-learning/recurrent-neural-networks/#:~:text=To%20train%20an%20RNN%2C%20the,of%20a%20cost%20function%20C>.

<https://medium.com/@CallMeTwitch/building-a-neural-network-zoo-from-scratch-the-recurrent-neural-network-9357b43e113c#:~:text=The%20above%20photograph%20shows%20how,intermediary%20steps%20of%20the%20network>.

<https://apmonitor.com/pds/index.php/Main/LongShortTermMemory>

<https://deepai.org/machine-learning-glossary-and-terms/exploding-gradient-problem#:~:text=The%20exploding%20gradient%20problem%20is,(weights)%20become%20excessively%20large>.

<https://www.datacamp.com/tutorial/lstm-python-stock-market>