CS337

Artificial Intelligence & Machine Learning

Report

Stock Market Price Prediction Using Sentiment Analysis

Submitted by

Gangula Bhuvan Reddy-200050040

Chilukuri Mani Praneeth-200050028

Paidi Venkata Ganesh-200050094

Peram Ankshitha-200050105

ABSTRACT:

In this project we attempt to implement a machine learning approach to predict stock prices. Machine learning is effectively implemented in forecasting stock prices. The objective is to predict the stock prices in order to make more informed and accurate investment decisions. We propose a stock price prediction system that integrates mathematical functions, machine learning, and other external factors for the purpose of achieving better stock prediction accuracy and issuing profitable trades.

We tried to predict stock prices using two different architectures, viz., Recurrent NNs and Transformers. We intend to compare the results of each as well as among various RNNs. RNNs and Transformers are very powerful in sequence prediction problems because they're able to store past information, each in their own way. This is important in our case because the previous price of a stock is crucial in predicting its future price.

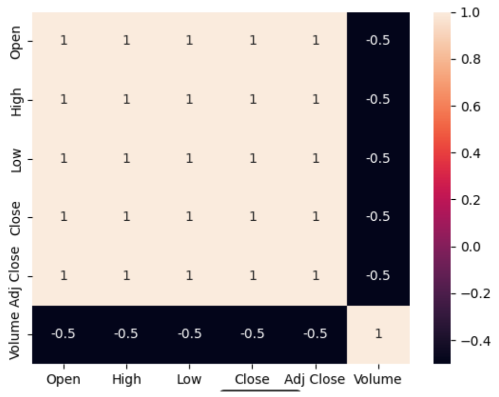
INTRODUCTION

As a high-risk and high-return market, the stock market has always been closely watched by investors, and stock forecasting has always been a research topic of great concern to researchers. Accurate prediction of stock prices plays an increasingly prominent role in the stock market where returns and risks fluctuate wildly, and both financial institutions and regulatory authorities have paid sufficient attention to it. In the early days, people have found that the establishment of mathematical models can be very good, such as the time series model, because its model is relatively simple and the forecasting effect is better. However, due to the non-linearity of stock data, some machine learning methods, such as SVMs came into use. Later, with the development of deep learning, some architectures such as RNN, LSTM, came into usage as they can not only process non-linear data, but also retain memory for the sequence and retain useful information, which is positive.

In recent years, Transformers have gained popularity due to their outstanding performance. Combining the self-attention mechanism, parallelization, and positional encoding under one hood usually provides an edge over classical LSTM and CNN models. One major advantage of the transformer architecture is that at each step we have direct access to all the other steps (self-attention), which practically leaves no room for information loss, as far as message passing is concerned. On top of that, we can look at both future and past elements at the same time, which also brings the benefit of bidirectional RNNs, without the 2x computation needed. And of course, all this happens in parallel (non-recurrent), which makes both training and inference much faster.

Dataset used

We used the **yfinance** API to download stock price data of various stocks daywise from 2007 till now. The downloaded data consisted of Various features like Opening, Closing, High, Low prices etc.. but as seen in the figure below, the correlation between the features is ~1, so we used only the '**Close**' as the feature to the Neural Network.

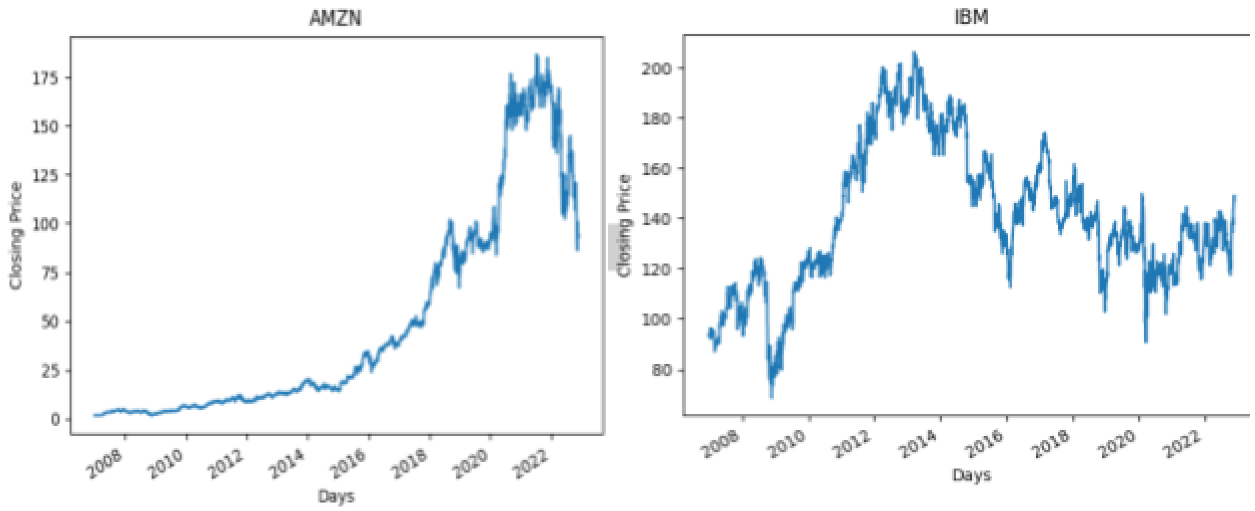


In this project we wanted to build a model , which can predict the stock price for the next **N** days, given the prices in a window of **T** days.

So we need to create a dataset out of the daily stock-price data. Each datapoint in the dataset **(x,y)** is of the form **x** ∈ R^T, **y** ∈ R^N.

From the daily-data, a dataset is made by sliding a window of **T days** across the data set, and setting the **next N days** as its label / target. So each point in the dataset is of the form $x_t = (p_t, p_{t+1}, \ldots\ldots p_{t+T-1})$; $y_t = (p_{t+T}, p_{t+T+1}, \ldots\ldots, p_{t+N-1})$. We later experimented with various values of **T,W** and compared the performances of models. More on this later.

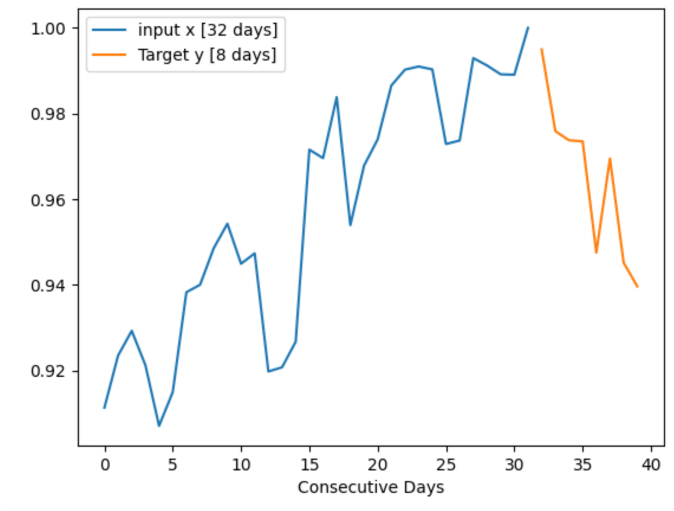
This figure shows the Stock-Price values of **AMZN, IBM** stocks from 2007



Normalization:

We used Min-Max scaling on the stock-prices while making the dataset. Also we stored the scaling factors in the model itself in all the architectures, in order to consistently scale the data for prediction on new data as well.

An Example data point is shown below for **(T,N) = (32,8) days.**

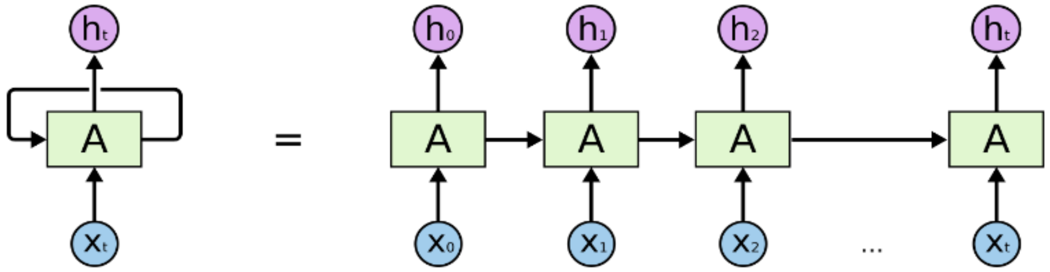


Using Recurrent Neural Networks

1. RNNs

Stock Price Prediction is a Sequential Data, and thus Normal Feed-Forward Neural Networks cannot be used for this task, as there is no notion of a *sequence* of inputs in The Standard NN. And it doesnt care about the movement of the stock price, it just treats them as independent features. So we need to use an Architecture that can **remember** the past observations in the sequence and use them to predict the next prices. So it has a **memory.**

Recurrent Neural Networks address this issue, They are Networks with loops in them, allowing information to persist.



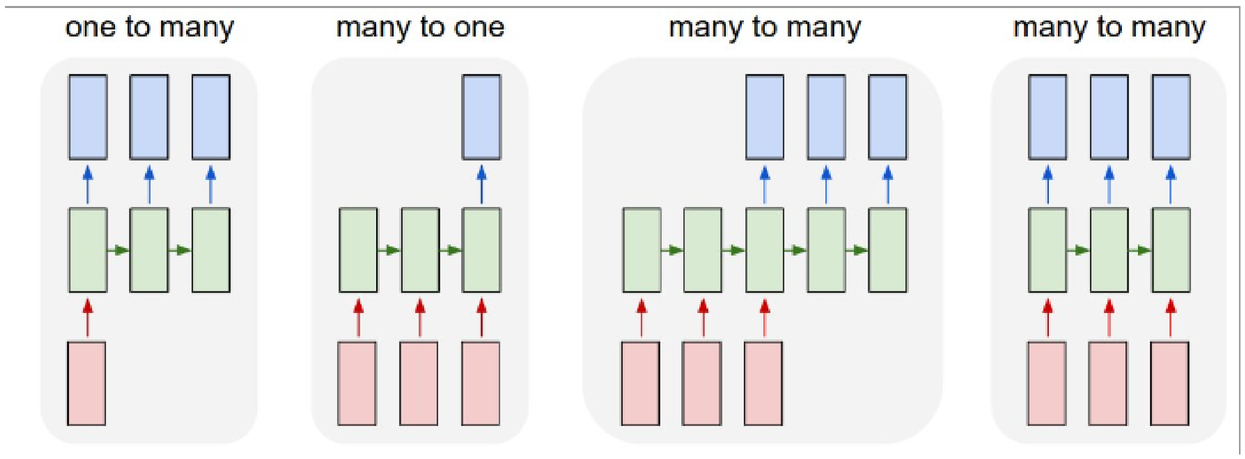
While processing a sequence, RNN takes $\mathbf{x_t}$ as input, and it stores what is called a **hidden state h_t,** which remembers the info from the previous time-steps while processing the next timestep. And that is fed as an input to the next-timestep along with $\mathbf{x_{t+1}}$. So essentially $\mathbf{h_t} = f(\mathbf{W_h h_{t-1} + W_x x_t})$, where f is some activation function like **tanh, relu** etc. We later experimented on the choice of **f .**

This recurrent relation w.r.t time is what gave RNNs that name.

The output o_t can be obtained by using another **Linear FC** layer and passing $\mathbf{h_t}$ **into it. I.e,** $\mathbf{o_t = W_o h_t}$ (with bias). Note that $\mathbf{o_t} \in \mathbf{R^N}$, **and** $\mathbf{x_t} \in \mathbf{R^T}$.**(size of h_t is our choice)**

There are many advantages of using RNN, mainly

- They can deal with sequences of any length
- They can be used in various models, as shown below.



Each of them can be used in Various Applications, like translation, image processing etc. We chose to use the **Many-to-one** Model, i.e, consider the output at the **last-time-step** only.

Choosing the Architecture

There are many parameters in the Model that we are free to choose, and the main params are

- Choice of **(N, T)**
- Size of **Hidden Layer/ State.** Also number of **RNN units (we chose 1)**
- Choice of Activation **tanh / ReLU**
- And other parameters like dropout, BatchSize, lr, epochs etc for **Training.**

We experimented with all the above choices of model, and the results are summarized in the upcoming sections.

So as now we have a model chosen, and also the Dataset is ready, Lets move to Training the Model.

Training the model

As with the normal Neural Networks, all we have to do is to update the Weights involved in the Model to fit the data. It needs **backpropagation**. But in the case of RNNs, we need to unroll the sequence back in time to find the gradients. So it is called as **BackPropagation Through Time (BPTT).**

As this process is sequential through time, it cannot be parallelized, hence BPTT is usually quite slow compared to Normal NNs

We used a MSE loss between the predicted o_t and true. And used the **Adam** optimizer in **PyTorch** with the weights as the parameters.

We did a 80:20 split on the Dataset, and trained models for various stocks data, and saved those models in the **MODELS/** folder.

Then trained the model tuning various combinations of **(lr,epochs,BATCHSIZE)** and also employed **Early Stopping,** where we saved the model version with best Validation loss. The various results obtained along with the training time are shown below.

Experiments and Results.

We varied the values of **(T,N,hidden_size)** along with the **activation f'n used.**

We chose T from [16,32, 64] and N from [1,4,8,16], hidden_size from [32,64]

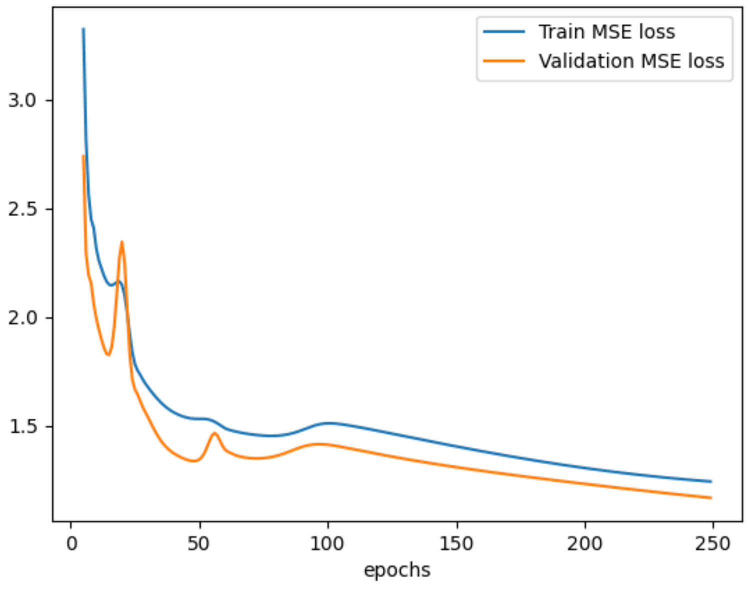
Note that MSE loss will obviously increase with N, because we are predicting for more days, which implies more uncertainty as stock prices really are unpredictable!, and so more error.

BatchSize, and Learning rate

The observations found are (which are quite obvious) that increase in Batch Size increases training time, and hence most optimal choice is **128, 256.** But **256** seemed the better choice because After choosing the best learning rate(around **[0.002-0.005],** because higher values are giving **non stable Loss vs epoch curves),** both **256,128** are giving similar **MSE losses on Train, Validation.** So we can choose **256** because it **reduces time by almost a factor of 0.5.** So the best setting found was **[256, lr = 0.005].... Trains in ~19sec on GOOG dataset.**

Some observed data; (for **(T,N,Hidden = H) = (32,1,32)**) on GOOG dataset.

Batchsize	256	128
Best lr	0.005	0.003
Avg Time taken	20 sec	37sec
Avg Train MSE	1.25	1.27
Avg Validation MSE	1.01	1.10



The figure shows the Loss plotted vs number of epochs.

While running the code we can tune all 3 <B,lr,epochs> and it saves the model with the best Validation error.

Higher learning rate values make this graph oscillate and diverge too much.

Testing the Obtained Models

Now we have models trained with (T,N,H) = 32,1,32 on various stocks like **GOOG, AMZN, MSFT, IBM etc..** And we also tested the model obtained from training one stock to other stock data as well (Because we utilized all data for Train + Valid). And because the variations of stock prices are similar in these companies, they gave good results on one-another.

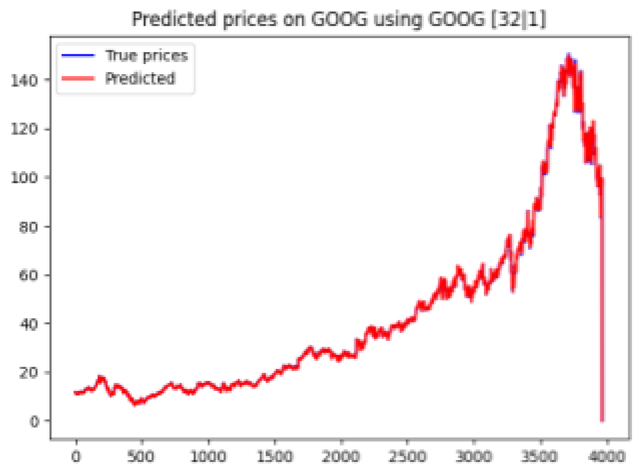
Some stocks didn't vary much(like DELL) etc, and using those models to predict on recently- high varying - high price stock like TSLA, MSFT gave bad results. But training on MSFT stock gave excellent results on other stocks.

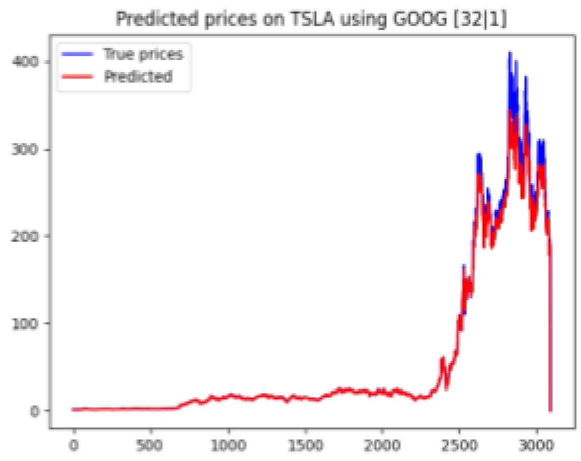
Model Trained vs MSE loss	**GOOG**	**TSLA**	**IBM**	**AAPL**	**MSFT**
GOOG	1.04	95.87	5.62	1.64	55.41
GOOG**	1.32	38.9	5.33	1.98	15.65
MSFT	1.28	21.78	4.64	1.89	5.43
TSLA	4.01	22.33	4.32	4.72	7.14

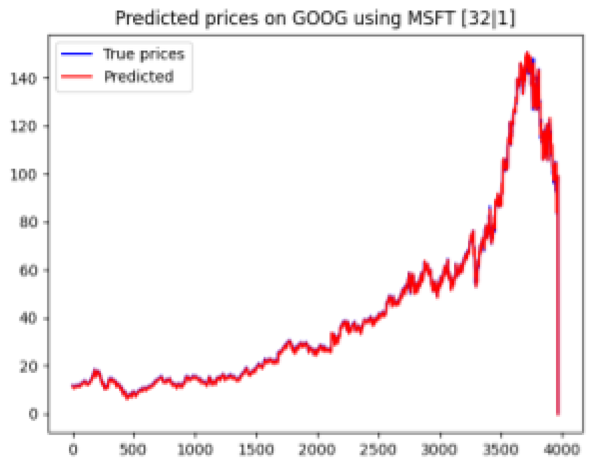
Because the google stock max price is lower... if we scale MinMax wrt google data, then while prediction on high price stocks like TSLA, MSFT when scaling with GOOG stock , the scaled prices go beyond 1. If we instead scale the training of google data wrt $TSLA_{max}$ **(indicated by GOOG** in the table)** then the training data of GOOG never reaches 1 it will be < 1. So it can't learn the trends of stock at high values.

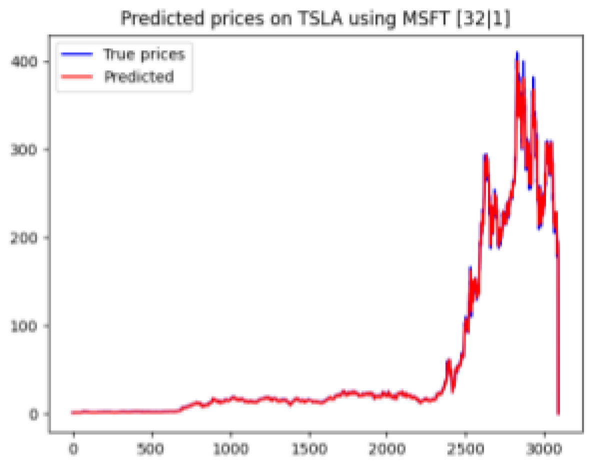
Thus we experimented on All Stocks vs All stocks and MSFT model was better to generalize because all kinds of variations **(slow increase, sudden ups, sudden downs etc....)** are present in the stock data as well as the price of stock is high, and hence the model was able to learn all the variations better and also deal with the price ranges of all other Stocks.

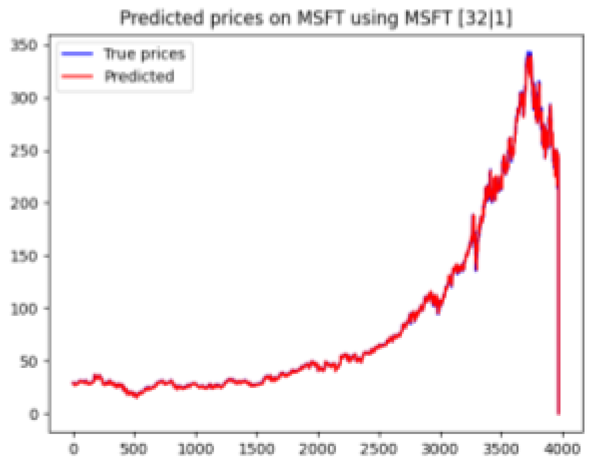
And the Graphs to visualize the performance all plotted below. NOTE: these are plotted by taking y_pred vs y_true values of the next day in all the data points in a timely fashion. It does not indicate a continuous prediction throughout the whole range. Our model just predicts the next day value given a 32 days input [32|1]

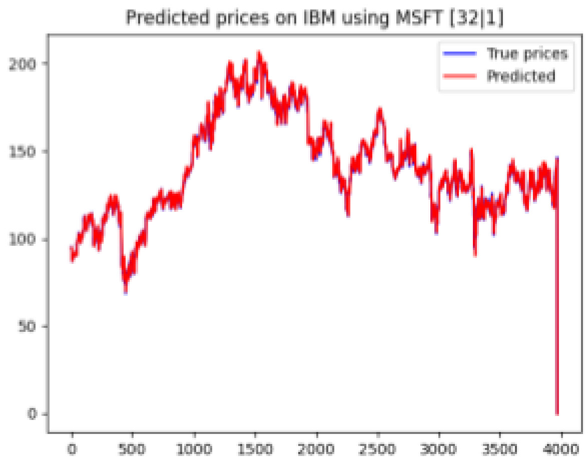

Predicted prices on GOOG using GOOG [32|1]
True prices
Predicted


Predicted prices on TSLA using GOOG [32|1]
True prices
Predicted


Predicted prices on GOOG using MSFT [32|1]
True prices
Predicted


Predicted prices on TSLA using MSFT [32|1]
True prices
Predicted


Predicted prices on MSFT using MSFT [32|1]
True prices
Predicted

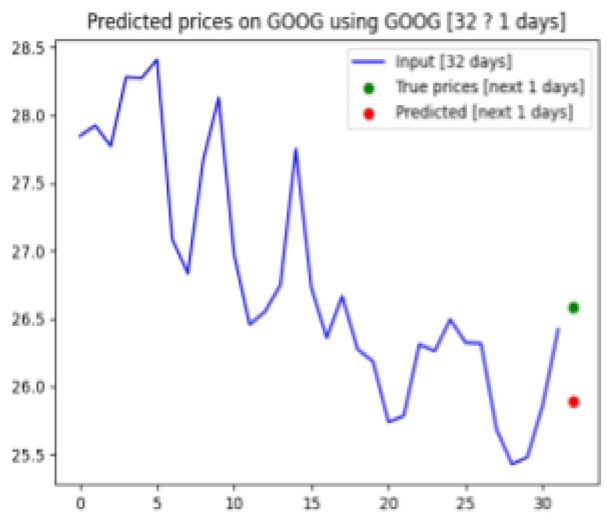

Predicted prices on IBM using MSFT [32|1]
True prices
Predicted

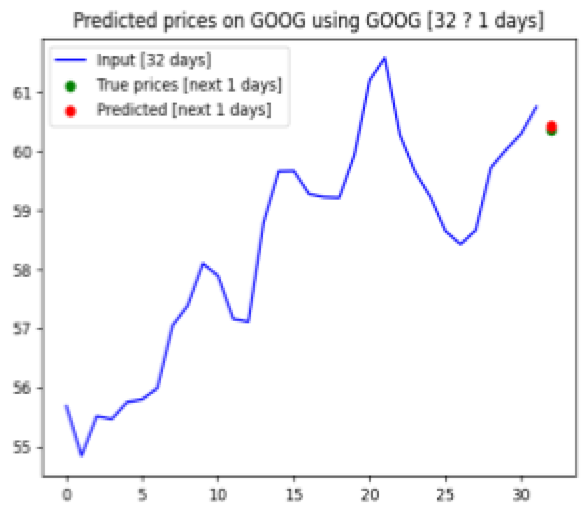
Individual datapoint predictions:

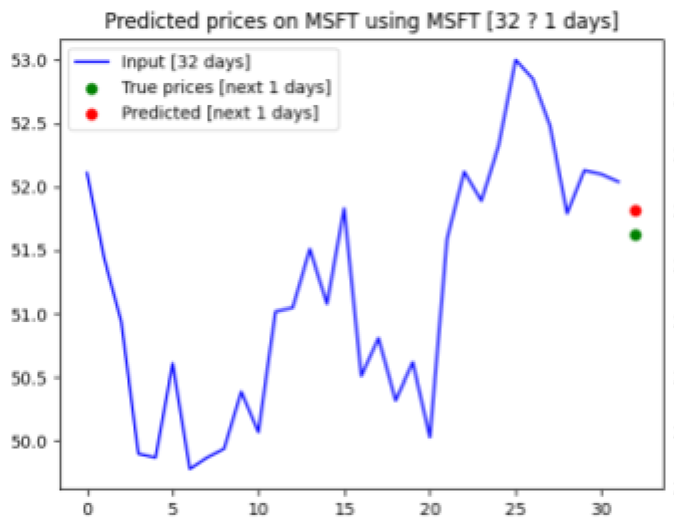
Here some results of predictions on Individual points are given, i.e given the prices of **T** days, what is the predicted price on next **N days.** Again these are obtained for various stock models.

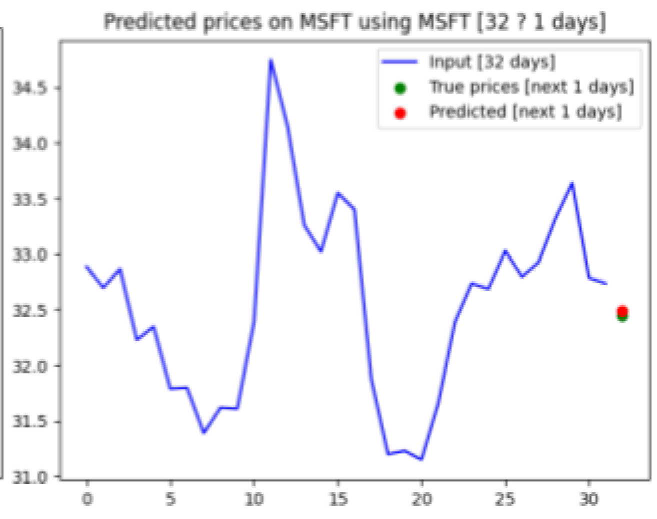
We fixed **N** = **1** in the previous analysis for simplicity, Now we also consider a case where **N** > **1.**

FOR N = 1







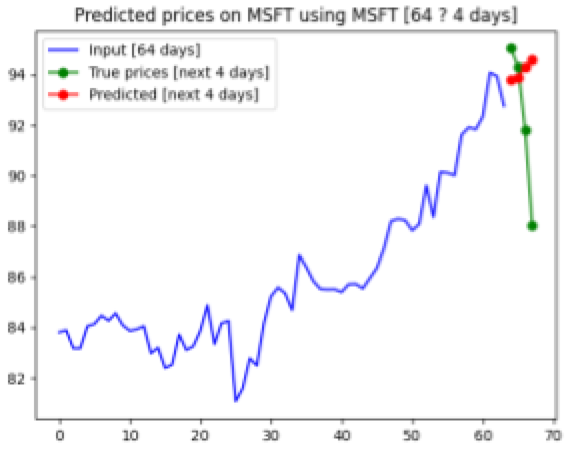


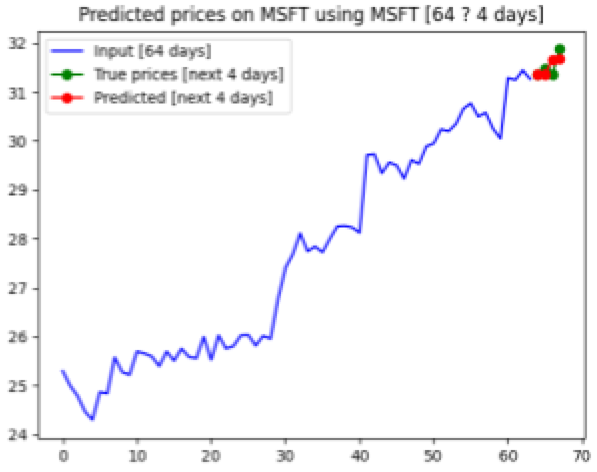
Plots for N = 4, T = 64(H = 32)

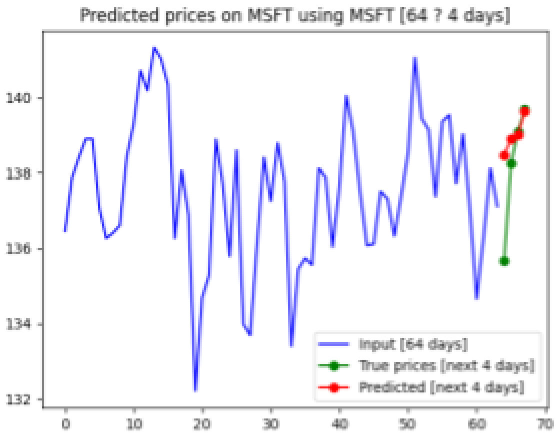
The following are the MSE loss obtained on GOOG, MSFT trained models on all other stocks. NOTE that the MSE losses this time will be higher because we are predicting more outputs here.

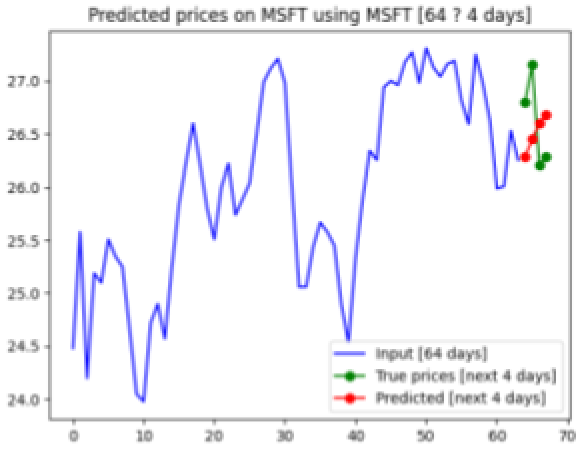
Model Trained vs MSE loss	GOOG	TSLA	IBM	AAPL	MSFT
GOOG	3.01	271.81	21.88	5.027	168.44
MSFT	2.78	48.78	11.99	4.11	11.17

Also some Random Datapoint predictions are plotted below.







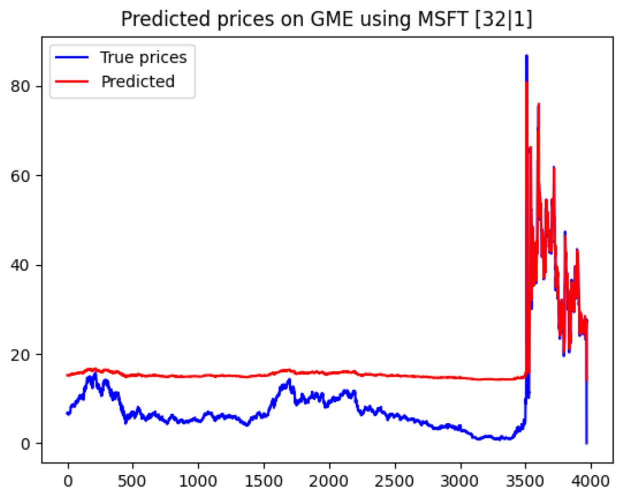


Tanh vs ReLU

So which is the better activation/ non_linearity among these 2 choices? For finding out, we trained the RNN model on **(T,N,H) = (32,1,32)** with these 2 settings. The training results obtained were:-

Train Metric	Train MSE	Val MSE	Time
Tanh	5.18	5.21	25
ReLU	6.15	4.18	22

So the Training results are almost same, or better for ReLU. But when tested on other stocks ReLU had some issues, it gave more error than the Tanh model.



The issue seems to be that the model is always giving predictions Above a certain Min value, so all the test points below that value seems to get wrongly predicted. Which kindof suggests some thing related to the nature of ReLu.

But Traditionally, the advantages Tanh has over ReLU are following:

- It normalizes the **h_t to [-1,1]** thus avoiding any problem of exploding / vanishing
- It is smooth + it combats extremes by being a middle ground.

So ReLU outputs value > 0, hence it has the inability to give -ve values in the Hidden State. Limiting the output it produces to some extent.

NEXT: Long Short Term Memory NNs... LSTMs........ Addresses the problems with RNNS.

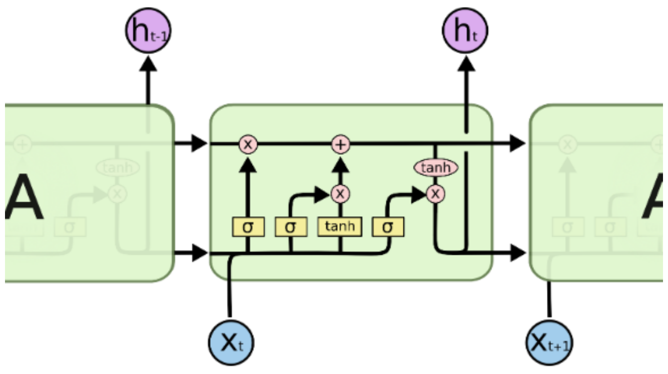
2. LSTMs

LSTMs are a kind of **Recurrent NNs**, these address the problems of RNNs.

The Major issue with RNN is the issue of **Vanishing/ Exploding Gradients.** While doing **BPTT,** we need to multiply with a factor of **W**$_h$ when unrolling back in time. So this repeated multiplication might Blow up / Diminish the gradient and hence **RNNs arent that good when handling longer sequences.** So to deal with the issue of LOnger sequences, **LSTMs are used. They can handle larger sequences.**

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

The recurrent Nature is same as RNNs except that they have a different repeating unit inside, as well as **it stores <cell state, hidden state>** both.



As shown in the figure, LSTMs have **4 Gates** which are controlled (**amount to pass: 0 = block, 1 = pass everything)** by sigmoid signals. And the update the **h**$_t$**,c**$_t$ **in the way shown in figure.**

LSTM has Forget Gate, Output Gate, Input Gates. The model can learn which data to keep, and which to forget. And thus be able to remember long term data. **LSTM**s have lot of gates, paths in the cell, so it also has a lot of Weights to be learned, and hence LSTMs train significantly slower than RNNs

Choosing the Architecture

There are many parameters like RNN we are free to choose in the LSTM

- Choice of **(N, T)**
- Size of **Hidden Layer/ State.** Also number of **LSTM units (we chose 1)**
- And other parameters like dropout, BatchSize, lr, epochs etc for **Training.**

We experimented with all the above choices of model, and the results are summarized in the upcoming sections.

Training the model

The training is similar to RNNs, and BPTT is employed. But here we have an additional cell state that propagates through time, so more work needs to be done to find the gradient, hence more training time.

Again we used a MSE loss between the predicted o_t and true. And used the **Adam** optimizer in **PyTorch** with the weights as the parameters. **SGD is slower, so Adam is preferred.** We did a 80:20 split on the Dataset. Then trained the model tuning various combinations of **(lr,epochs,BATCHSIZE)** and also employed **Early Stopping**

Experiments and Results.

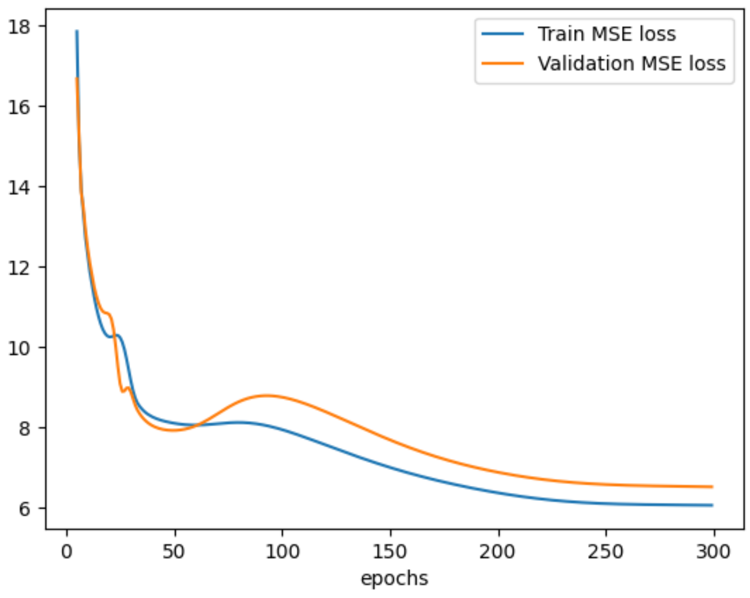
We varied the values of **(T,N,hidden_size), T from [16,32, 64] and N from [1,4,8,16], hidden_size from [32,64]**

BatchSize, and Learning rate

Since training is slower with LSTMs, the choices are 256 or even higher (say 512). But **256** seemed the better choice again because, there was no significant time improvement with **512**, maybe because of increased computation in each batch. And also both of them were giving similar errors.

Some observed data; (for **(T,N,Hidden = H) = (64,1,32)**) on **MSFT** dataset.

Batchsize	256	512	256(GOOG)
Best lr	0.005	0.006	0.006
Avg Time taken(300 epochs)	160 sec	150 sec	160 sec
Avg Train MSE	~6.00	6.40	1.42
Avg Validation MSE	~5.5 +- 0.5	~6.6	1.15



The figure shows the Loss plotted vs number of epochs.

While running the code we can tune all 3 <B,lr,epochs> and it saves the model with the best Validation error.

Higher learning rate values make this graph oscillate and diverge too much.

Testing the Obtained Models

Now we have models trained with (T,N,H) = 64,1,32 on various stocks like **GOOG, AMZN, MSFT, IBM etc..** And we also tested the model obtained from training one stock to other stock data as well (Because we utilized all data for Train + Valid). And because the variations of stock prices are similar in these companies, they gave good results on one-another.

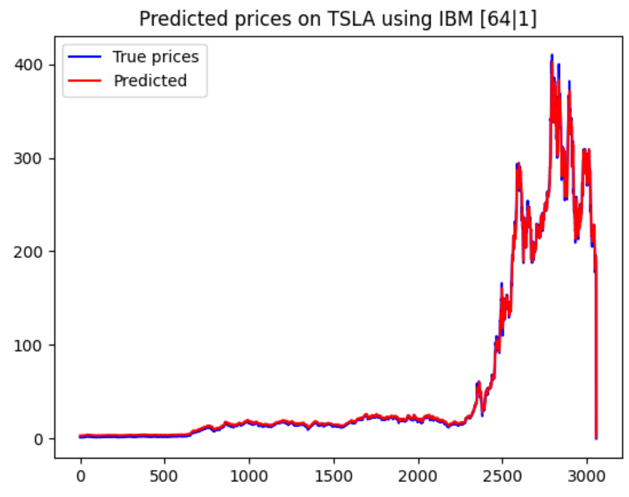
Model Trained vs MSE loss	GOOG	TSLA	IBM	AAPL	MSFT
GOOG	1.19	59.62	6.83	1.99	32.44
MSFT	2.67	22.90	7.44	3.34	6.63
IBM	2.09	22.60	4.33	2.97	5.52

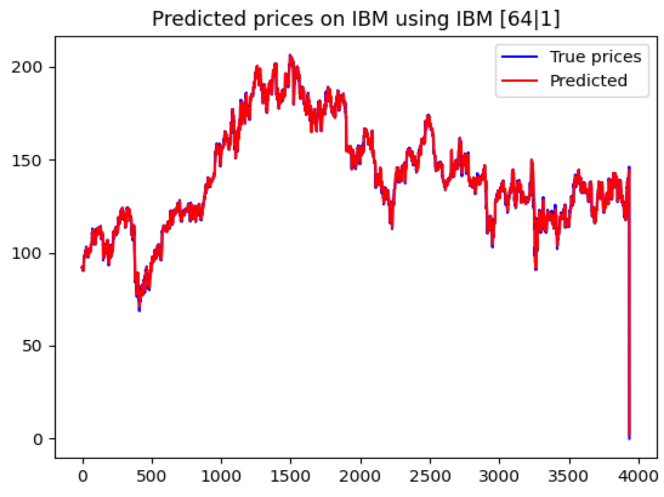
Now IBM was the the best stock to train for **long term sequence inputs.** It is clearly able to perform very well on other stocks as well.

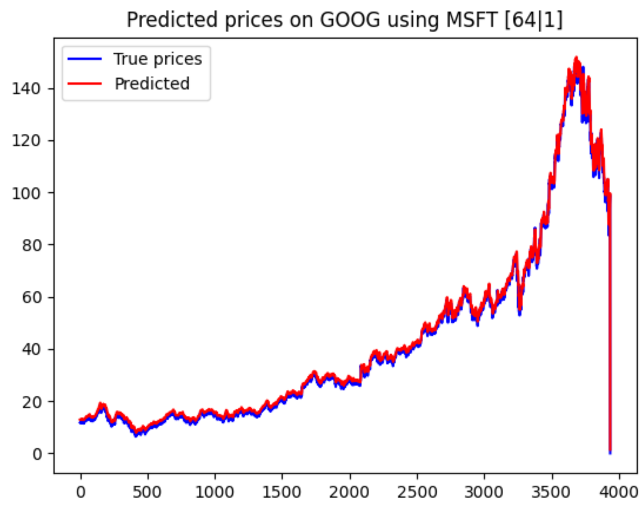
Maybe we can improve the GOOG, MSFT models by better tuning and lr adjustments but in the few tweeks we tried, this IBM model performed the best.

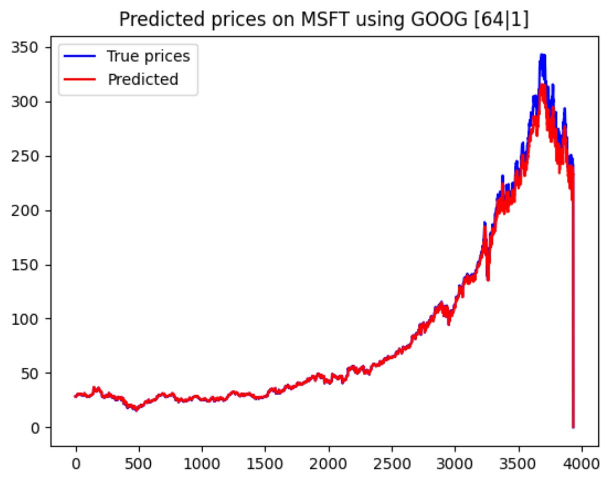
It might be because IBM data is very volatile from early years itself, so maybe that's the reason why IBM data captures long term dependencies more accurately than MSFT etc where a lot of data is non volatile.

And the same few Graphs to visualize the performance all plotted below. NOTE: It does not indicate a continuous prediction throughout the whole range. thOur model just predicts the next day value given a 64 days input [64|1]





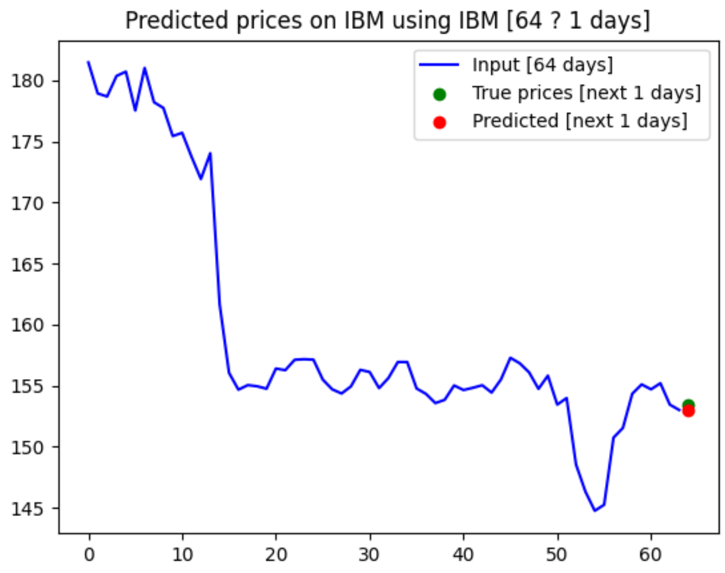


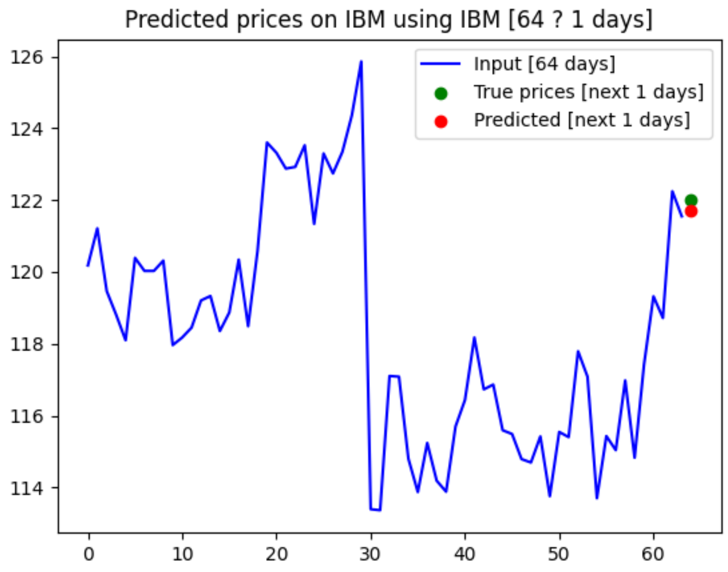


Individual datapoint predictions & >1 day Predictions:

We can plot the prediction on single data points similar to the previous RNN case, as they are just random points, they look similar, but they dont give any extra info. So just a few are included.

FOR N = 1





But a notable observation by seeing many random above such plots was that the **amount of error** usually decreased and the prediction is being closer to the True.

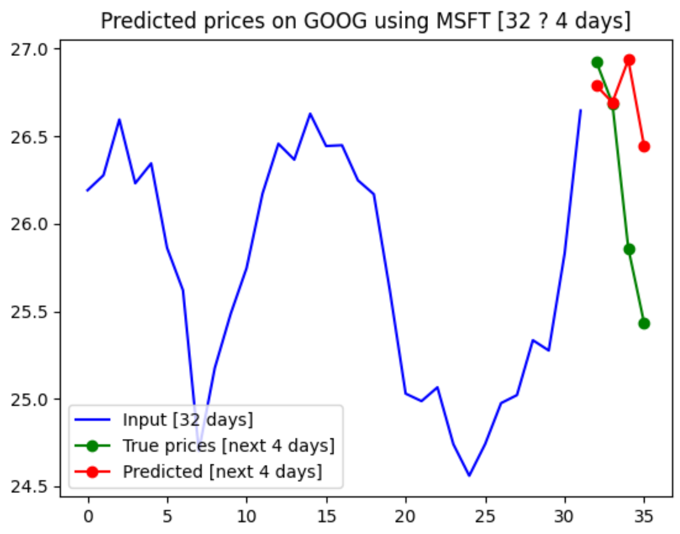
Plots for N = 4, T = 32 (H = 32)

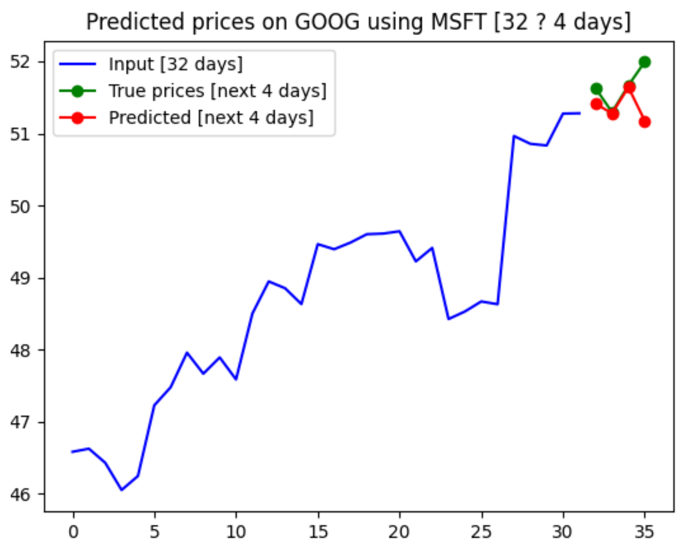
One Notable thing was that Training Time decreased a lot when the input window size was reduced to 32. So (T,H) have a large effect on the TRaining complexity of LSTM, other than BatchSize.

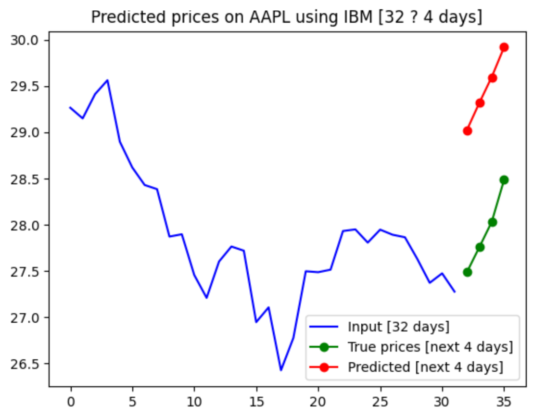
The following are the MSE loss obtained on IBM, MSFT trained models on all other stocks. NOTE that the MSE losses this time will be higher because we are predicting more outputs here.

Model Trained vs MSE loss	GOOG	TSLA	IBM	AAPL	MSFT
IBM (~80sec)	4.88	57.42	9.59	5.82	18.96
MSFT	2.78	47.25	9.99	4.11	10.91

Also some Random Datapoint predictions are plotted below.







3. GRUs

Gated Recurrent Units(GRUs) are also recent kind of **Recurrent NNs**, which are pretty similar to LSTMs. The **LSTMs stores <cell state, hidden state>** both, but GRUs get rid of the **cell state. It only stores the hidden state.**
GRUs have only 2 Gates. A **RESET** and an **UPDATE** gate. Which are controlled by sigmoid signals.

As the number of Parameters is low, they train faster than LSTMs, but still they have ability to hold long term info just as LSTMs.

Rest of the analysis of GRUs is pretty similar to LSTMs. Including the **Selection of Architecture, and the Training Process.**

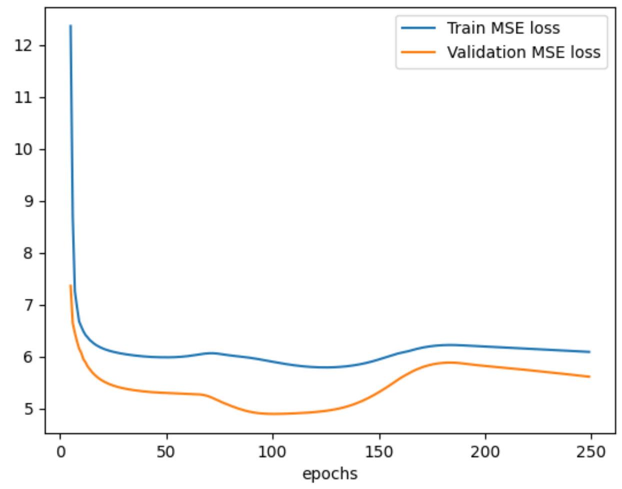
Experiments and Results.

BatchSize, and Learning rate

I used the BATCH SIZE to be 256 just like LSTM and tuned the learning rate to obtain the following results.
Some observed data; (for **(T,N,Hidden = H) = (64,1,32)**)

Batchsize	256(MSFT)	256(GOOG)	256(IBM)
Best lr	0.005	0.005	0.006
Avg Time taken(250 epochs)	100 sec	100 sec	111
Avg Train MSE	5.89	1.18	4.51

Avg Validation MSE	**4.89**	**1.01**	**4.88**



The figure shows the Loss plotted vs number of epochs.

While running the code we can tune all 3 <B,lr,epochs> and it saves the model with the best Validation error.

Higher learning rate values make this graph oscillate and diverge too much.

Testing the Obtained Models

Now we have models trained with (T,N,H) = 64,1,32 on various stocks like **GOOG, AMZN, MSFT, IBM etc..** And we also tested the model obtained from training one stock to other stock data as well (Because we utilized all data for Train + Valid). And because the variations of stock prices are similar in these companies, they gave good results on one-another.

Model Trained vs MSE loss	**GOOG**	**TSLA**	**IBM**	**AAPL**	**MSFT**
GOOG	1.13	111.13	5.83	1.78	62.8
MSFT	1.28	22.67	4.75	1.90	5.48
IBM	3.01	35.98	4.62	4.12	11.24

But GRUs give similar results to LSTMs and also training time is lesser, so GRUs are preferred

Individual datapoint predictions & >1 day Predictions:

The results here would be pretty similar to LSTMs so not shown again. But can be seen while Comparing all 3 RNNs.

Comparison Among RNNs.

All the results obtained are by **Training MSFT stock** and testing it on **various** stocks. Varying input sequences are taken **[16,32,50,64] with output fixed to be 1 for comparison simplicity.**
All the errors reported are MSE loss only. (on the whole test-stock dataset)

RNN:

INPUT size	16	32	50	64
Val MSE	5.79	5.43	3.86	5.73
AAPL	4.52	1.77	2.15	2.76
IBM	6.51	3.99	5.41	4.85
AMZN	6.57	2.76	3.34	4.09
TSLA	24.5	21.51	21.41	22.92

GRU: (H = 32)

INPUT size	16	32	50	64
Val MSE	5.37	5.68	4.41	5.34

AAPL	1.70	1.87	1.80	1.68
IBM	4.34	4.42	4.41	4.29
AMZN	2.71	2.94	2.86	2.64
TSLA	20.06	21.31	21.58	20.09

LSTM: (H = 32)

INPUT size	16	32	50	64
Val MSE	6.33	5.78	5.55	5.08
AAPL	2.36	2.63	2.71	1.88
IBM	5.11	4.98	6.50	4.48
AMZN	3.28	3.65	3.89	2.96
TSLA	20.64	20.35	20.77	20.71

INSIGHTS:

- Among a given Model and a test stock, as we increase the Input size, the error should decrease as we provide more data, but in case of RNNs, as we make INPUT size go near 64, the accuracy rather decreases mostly. So it shows that RNNs can't deal with longer Data. But in case of LSTM, RNN as we can see, Error usually decreases in a row. And they give their best results when Sequence length is higher.
- If we compare across models with the same test-stock, input size. Then VERY CLEARLY GRU IS THE WINNER. Also GRU has less training time. SO GRU IS THE BEST MODEL BY THE DATA OBSERVED.
- LSTM might perform better on even bigger sequences, but we did not perform such analysis because of higher training time costs.

USING TRANSFORMERS

We have tried to implement a Transformers based architecture to predict stock market prices. In recent years, Transformers have gained popularity due to their outstanding performance. A Transformer is a neural network architecture that uses a self-attention mechanism, allowing the model to focus on the relevant parts of the time-series to improve prediction qualities. Combining the self-attention mechanism, parallelization, and positional encoding under one hood usually provides an edge over classical LSTM and CNN models. One major advantage of the transformer architecture is that at each step we have direct access to all other steps (self-attention), which practically leaves no room for information loss, as far as message passing is concerned. On top of that, we can look at both future and past elements at the same time, which also brings the benefit of bidirectional RNNs, without the 2x computation needed.

DATA:

For this implementation, we have used google stock data downloaded using yfinance API. The stock data starts on the date 19-08-2004. For every training day, we have the Open, High, Low, and Close price as well as the trading Volume of the stock.

DATA PREPARATION:

The price (Open, High, Low, Close) and volume features are converted into daily stock returns and daily volume changes, a min-max normalization is applied and the time-series is split into a training, validation and test set. Converting stock prices and volumes into daily change rates increases the stationarity of our dataset.

The training, validation, and test sets are separated into individual sequences with a length of seq_len (=128) days each. For each sequence day, the 1 price feature (Close). During a single training step, our Transformer model will receive batch_size (=32) sequences that are seq_len (=128) days long and have 1 feature per day as input.

TIME EMBEDDINGS:

As the first step of our Transformer implementation, we have to consider how to encode the notion of time when processing our stock prices. Without Time Embeddings, our Transformer would not receive any information about the temporal order of our stock prices which leads to a stock price from 2020 having the same influence on tomorrows' price prediction as a price from the year 1990.

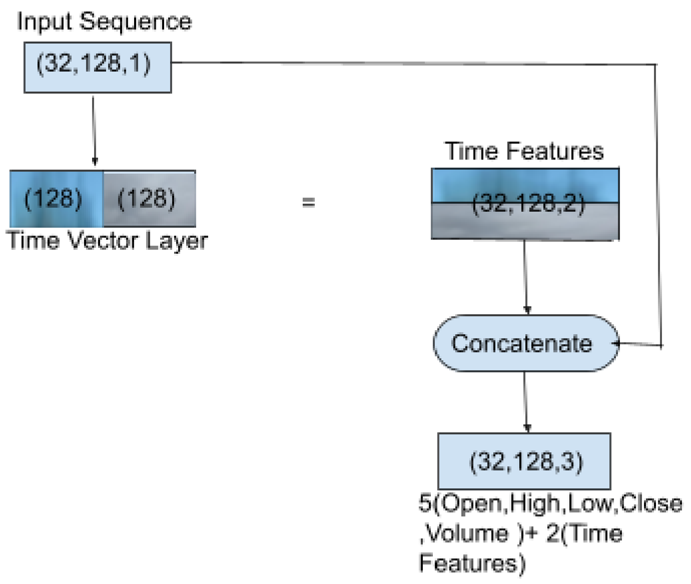
Time2Vector LAYER:

We implemented the approach described in the paper "Time2Vec: Learning a Vector Representation of Time" which represents time using vectors. This vector representation can be thought of as a normal embedding layer that can be added to a neural network architecture to improve a model's performance. The paper has two main ideas:

1. Time has to include both periodic and non-periodic patterns.
   a. The non-periodic pattern is "**wt + φ**"
   b. The periodic pattern is "**sin(wt + φ)**"
2. Representation should have an invariance to time rescaling.

FEEDING THE TRANSFORMER:

A combination of time vector representation with price and volume features is used as input for our transformer.

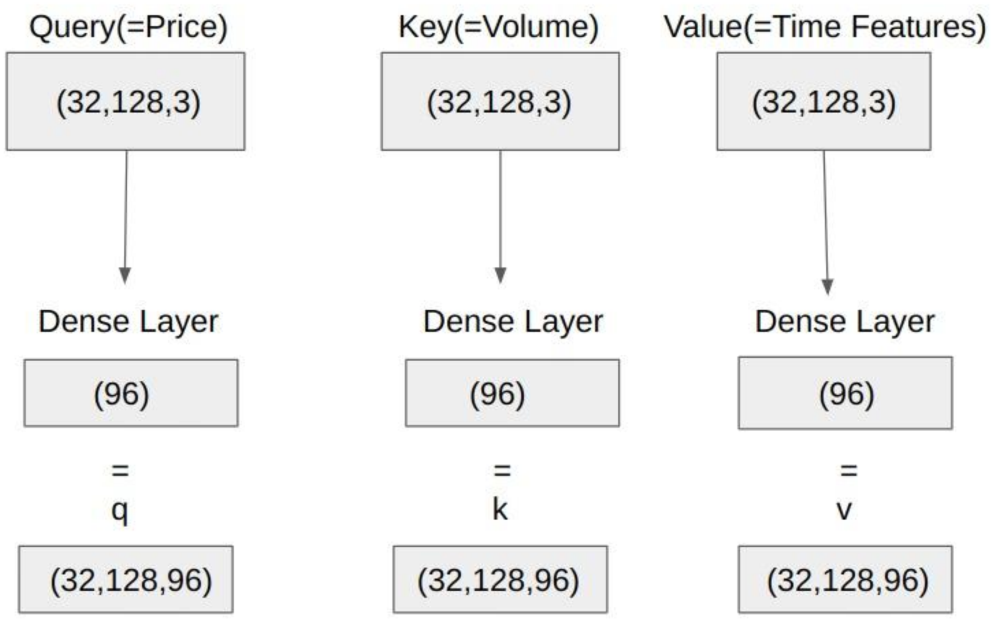


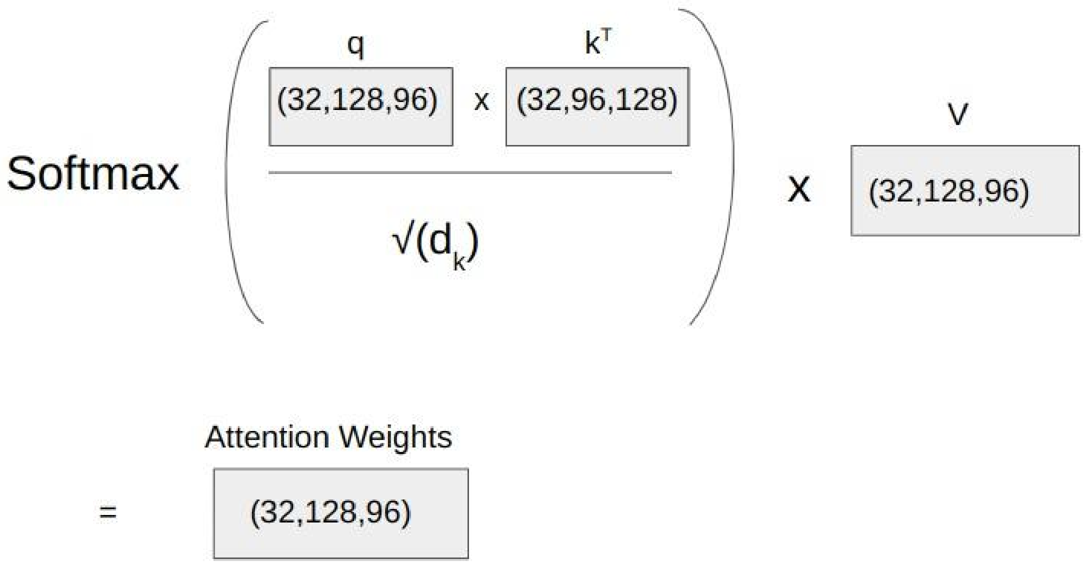
ATTENTION MECHANISM:

The self-attention mechanism consists of a Single-Head Attention and Multi-Head Attention layer. The self-attention mechanism is able to connect all time-series steps with each other at once, leading to a creation of long-term dependency understandings.

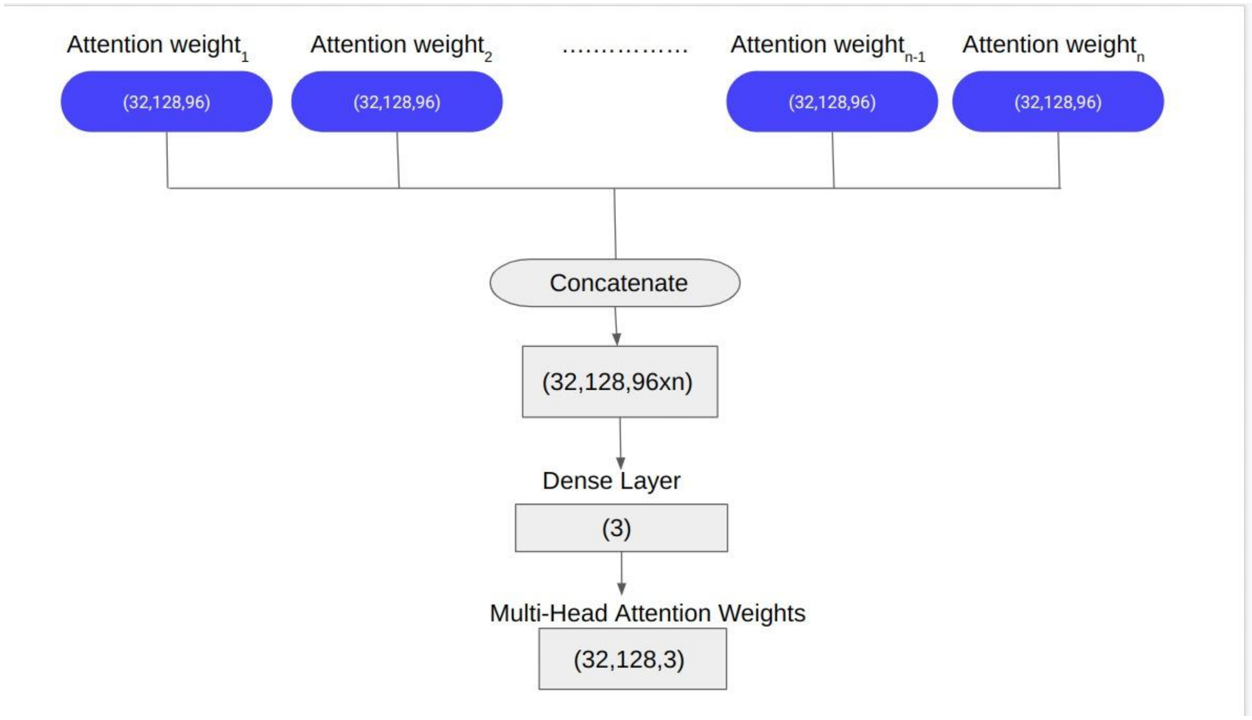
Finally, all these processes are parallelized within the transformer architecture, allowing an acceleration of the learning process.

Single-Head Attention: The input we fed to the transformer is the initial input to the first single-head attention layer. The single-head attention layer takes 3 inputs (Query, Key, Value) in total.





Multi-Head Attention: It concatenates the attention weights of n single-head attention layers and then applies a non-linear transformation with a Dense layer.



ENCODER LAYER:

Each encoder layer incorporates a self-attention sublayer and a feedforward sublayer. The feedforward sublayer consists of two 1-dimensional convolutional layers with ReLU activation in between. Each sublayer is followed by a dropout layer, after the dropout, a residual connection is formed by adding the initial Query input to both sublayer outputs.

TRAINING, VALIDATING AND TESTING THE MODEL:

The normalized data is split into train (80%), validation (10%), and test (10%) data in the proportions mentioned in brackets. The model is trained on the training data for 20 epochs. After each epoch, the validation loss is calculated and the model which gives the best validation loss is saved.

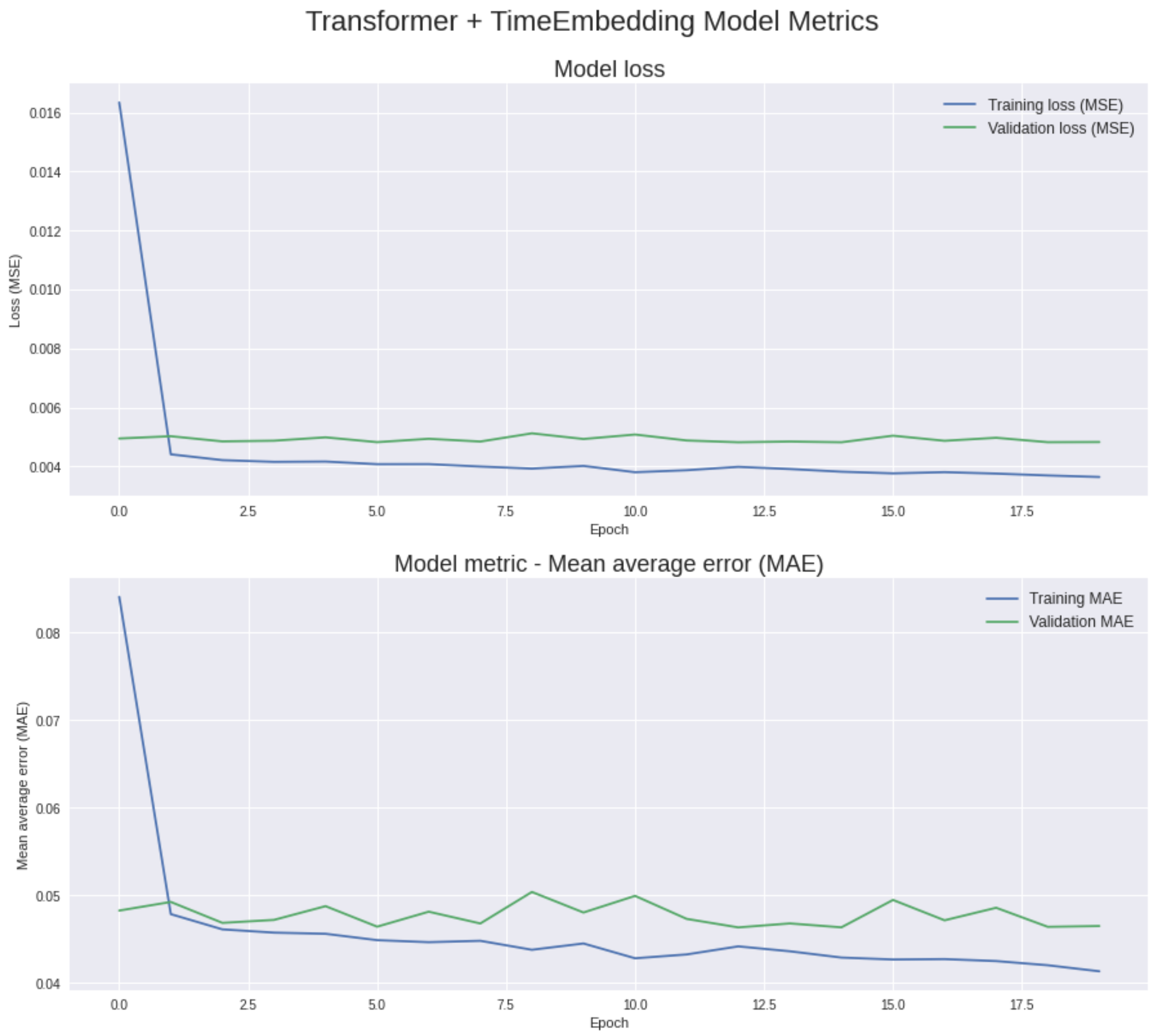
HYPERPARAMETERS:

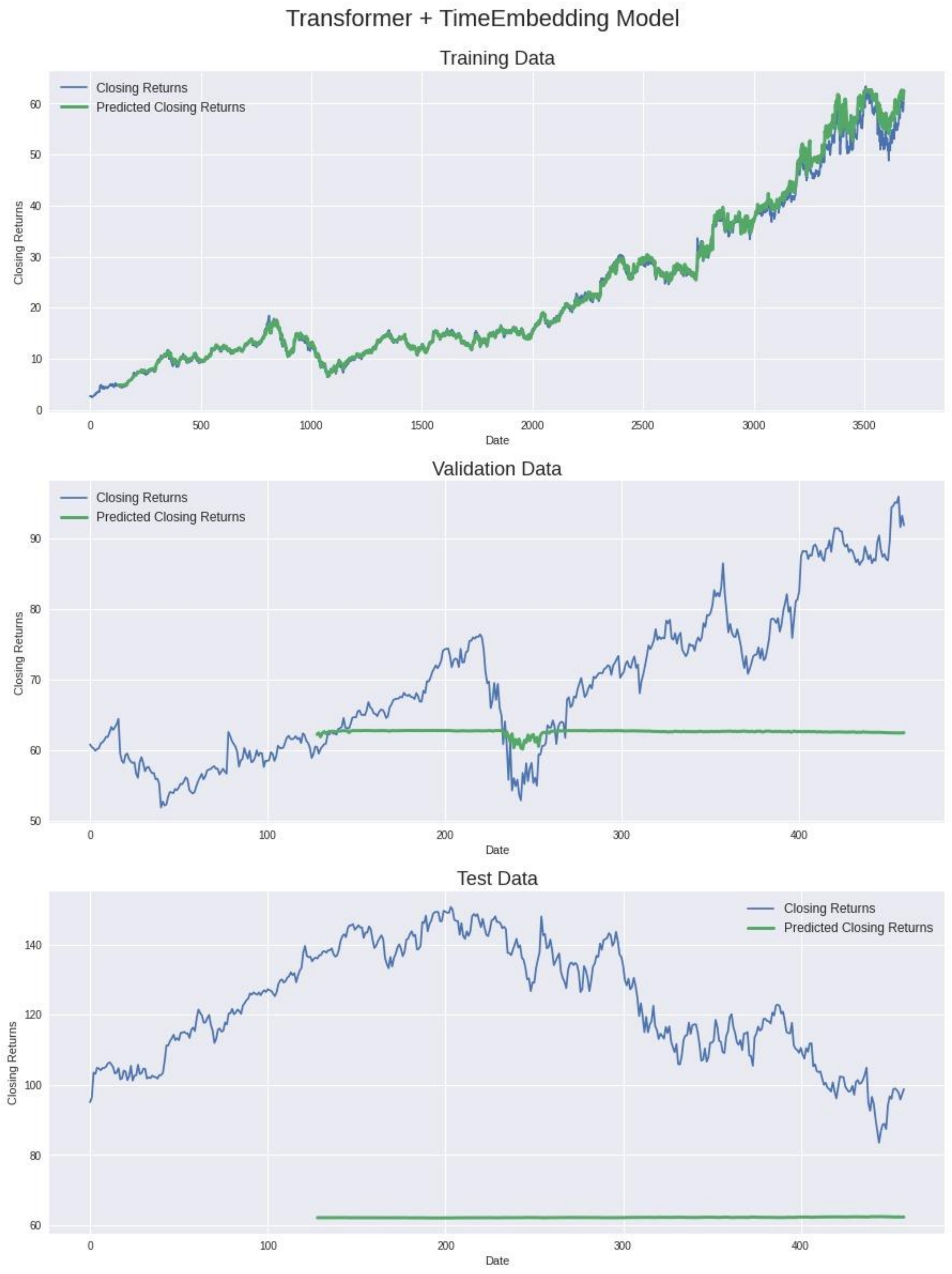
There are three major hyperparameters, viz., batch size, length of sequence and the number of heads for the multi-head attention layer. After experimentation with these values, we chose the following values for these hyperparameters.

batch_size	:	32
seq_len	:	128
n_heads	:	12

RESULTS:

Training Data -	Loss : 0.0034, MAE : 0.0389
Validation Data -	Loss : 0.0048, MAE : 0.0463
Test Data -	Loss : 0.0048, MAE : 0.0511




Transformer + TimeEmbedding Model
Training Data
Closing Returns
Predicted Closing Returns
Closing Returns
Date
Validation Data
Closing Returns
Predicted Closing Returns
Closing Returns
Date
Test Data
Closing Returns
Predicted Closing Returns
Closing Returns
Date

CONCLUSION:

The transformer model is just predicting a flat line that is centered in between the daily stock price changes. Upon clearly observing the graphs for validation and test data, we can infer that the problem is with the choosing of training, validation and test datasets. The test and validation datasets have values higher than the maximum value in training data. Hence, for these, the maximum value of training data is always predicted.

REFERENCES

1. **Time2Vec: Learning a Vector Representation of Time :** https://arxiv.org/pdf/1907.05321.pdf
2. **Attention is All You Need :** https://arxiv.org/abs/1706.03762
3. For Transformers: https://github.com/JanSchm/CapMarket/blob/master/bot_experiments/IBM_Transformer%2BTimeEmbedding.ipynb

FURTHER ATTEMPTS AT IMPROVEMENT

Using Gaussian Process Regression

There are GP methods using which we can do the stock prediction.The main idea is to choose the Kernel function to Meet the task.

$K(u,v) = \mu^2 \exp(- ||u - v||_2^2 / 2\sigma^2)$, $\theta = [\sigma, \mu]$ are the set of parameters.

Now given test data X, we need to predict y. So we need to maximize the Marginal LIkelihood $P(y|X, \theta)$ wrt θ

So $\theta = \mathrm{argmax}_\theta P(y|X, \theta)$

So we can minimize the Negative-log-likelihood instead.

So $\theta = \mathrm{argmin}_\theta -\ln(P(y|X, \theta))$
$\theta = \mathrm{argmin}_\theta(y^T(K(X) + \sigma_n^2 I)^{-1} y + \ln(\det(K(X) + \sigma_n^2 I))$

We tried implementing this , but got stuck on a few NaN errors. Should be attempted further.

We tried to implement this paper
https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9754114&tag=1

RNNs references:

https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

https://www.kaggle.com/code/ozkanozturk/stock-price-prediction-by-simple-rnn-and-lstm

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

https://www.kaggle.com/code/rodsaldanha/stock-prediction-pytorch

https://towardsdatascience.com/pytorch-basics-how-to-train-your-neural-net-intro-to-rnn-cb6ebc594677

https://towardsdatascience.com/stock-prediction-using-recurrent-neural-networks-c03637437578