Stock Prediction using Generative Transformers

Alan Osys
ao539@kent.ac.uk

School of Computing
University of Kent

Word Count: 5,031

March 28, 2024

Abstract

The most prominent way of stock prediction has been Long Short Term Memory models and Recurrent Neural Networks due to the extensive knowledge we have on them and the proven reliability of both architectures[8]. The main problem of these architectures is the fact they lack the ability to perform parallelised training[8] which slows down training time tremendously and prevents scalability. I instead propose the use of transformers as they can easily be adapted to any time-series data with results such as a 0.699 correlation coefficient while predicting the MasterCard stock value. With their ability for parallelisation during training and getting a successful results within 30 minutes, the longest taking 50 minutes, excluding the very large one, training on a Nvidia 3060 GPU. The questions I tried to answer is whether the Transformer architecture is capable of replicating various time-series data, how well it can do it, how long it takes to train a model to perform the task and what time-series data does the architecture struggle with.

1 Introduction

Transformers have been outperforming other architectures used for time-series data for a little over 2 years now and have been the talk of most AI breakthroughs with LLMs that can translate, teach and even write code like shown in [1]. That being said the utilisation of transformers in this task hasn't been researched thoroughly, the only literature I could find about it was [6], where they successfully implemented a transformer for stock prediction however they didn't show how the architecture compares in training time to others like it and different sizes of the networks and [8] which was mostly theoretical which this paper is trying to build upon and bring the theory into reality.

Alongside the transformer I also recreated the RetNet architecture recently developed by Microsoft[5], that was said to have the same training time and performance as the transformer, with much faster and more computationally efficient inference.

I also implemented an RNN and an LSTM to compare how the Transformer architecture compares to the current state-of-the-art in stock prediction.

I propose using the Transformer architecture to circumvent problems with the current state-of-the-art architectures such as their inability to be trained in parallel and very expensive scalability.

2 Background

In this section I go into the background of the various architectures I use and what stocks are.

2.1 Stocks

Stocks are shares of a company that have a value dependent on how well the company is performing. While its true that stocks are related to the growth of a given company and current real world events, the reality is that stocks are inherently random in nature and therefore very hard to predict without having a vast knowledge of the stock market and being up to date on current events[6]. The value doesn't fluctuate much with only minor changes each day meaning predicting where the stock is going to go long-term is much more important than knowing it's direction in the short-term.

2.2 Why use Transformers

Stocks are an example of time-series data meaning the structure is a one dimensional array of numbers[6]. Until now RNNs and LSTMs were used most commonly for time series data due to both of these architectures being able to predict data using data preceding it[7].

2.3 RNNs

RNNs are inherently slow due to the fact that they must process a data point before being able to move on and process the next data point since the data from that computation is used in the computation that follows[7]. This is an issue, because the network is unable to be trained in parallel meaning increased training time which scales rapidly with size of the network.

2.4 LSTMs

LSTMs, much like RNNs, process one data point after another, however they are able to use memory cells to discard useless data which prevents vanishing gradients which is an issue in RNNs.[7] The issue of RNNs inability to perform parellelised training still stands.[2]

2.5 Transformers

Most of the issues with LSTMs and RNNs simply disappear with Transformers due to their ability to parallelise training(see Section 4.3)[1].

The Transformer is a fairly new architecture developed by Google which uses a multi-headed attention mechanism to learn relations between tokens and their positioning (see Section 4) which allows it to learn time-series data in parallel and understand the relations between different stock values at different times.

The attention mechanism is the main aspect of the Transformer architecture that allows it to outperform the current state-of-the-art architectures like RNNs and LSTMs, because of its ability to be trained in parallel and being less expensive computationally. Until the transformer used attention the best way of

learning time-series data were LSTMs which use similar principles to attention like discarding less useful data and allowing the network to focus on learning the more important data points, however it was still held back by the need to be trained recurrently.

2.6 RetNet

RetNet is a very new and not widely explored architecture that was developed off the back of the transformer[5], it uses a parallel representation for training which allows much faster and more computationally efficient training, but during the inference step it uses a recurrent representation which allows it to have an inference step that is as fast as an RNN but faster than a transformer. This is all in theory however and in reality, at least in my experience, the architecture performs quite poorly(see Section 5.5).

3 Methods

In this section I describe how I created each dataset, how I tested each network and how I evaluated the performance of each network.

3.1 The correlation coefficient

The correlation coefficient(ρ) is a value that expresses the similarity between two correlations and is the main way I evaluate the performance of each model. ρ is calculated by dividing the covariance by the product of the two variables' standard deviations. Where Standard Deviation is a measure of the dispersion of data from its average. Covariance is a measure of how two variables change together. The higher the correlation coefficient the better since it means the generated sequence is very similar to the real stock data that the ρ is measured against.

$$\rho = \frac{\sigma_{xy}}{\sigma_x \sigma_y} \tag{1}$$

Equation 1 is used to calculate the correlation coefficient where σ_x,σ_x is the population standard deviation and σ_x is the population covariance.

3.2 Sin function

I took the first 1000 values of a sin(x) function at intervals of 0.1 and used Equation 2 to generate my data.

$$A = \{x_0, x_1, x_2, ..., x_i\} \rightarrow x_i \cdot 100 + 100 \tag{2}$$

Equation 2 A is a list of positive integers used for training the transformer.

I tested the networks ability to replicate the sin(x) wave by giving it a random chunk of the sin(x) sequence and predicting the next 400 values. I then found the correlation coefficient between the generated correlation and the real one, with the results presented in Section 4. I was looking for a correlation coefficient of 1.0 since it is a deterministic, repeating, sequence that does not need to be generalised.

3.3 Fibonacci sequence

I took the first 100 numbers in the Fibonacci sequence as the dataset for this experiment. No additional work was needed since all of the Fibonacci numbers are positive integers that are quite easy to learn for the transformer.

I tested the networks ability to replicate the Fibonacci sequence by giving a Fibonacci number and having it recreate the sequence then finding the correlation coefficient between the generated sequence and the real one. I was looking for a correlation coefficient of 1.0 for the training data since it is a deterministic sequence.

3.4 Artificially Generated Stock Data

I created Artificially Generated Stock Data by creating a Markov chain of real stock data, allowing me to create a substantially large dataset of artificially generated stocks. The Markov chains states were chosen by iterating over every single number in the sequence of all the real stocks. The initial state is a random number taken from the stocks and then 200 numbers are generated in a sequence to be used as the Artificially Generated Stock Data.

The Network was tested by checking its ability to generate stocks from a different dataset I created the same way and once again found the correlation coefficient between the generated sequence and the real one.

I didn't have high hopes for the networks ability to replicate this data, due to the nature of Markov chains being random, however it would've been very useful to generate infinite amounts of data if the data was proven to be predictable.

3.5 Real Stock Data

For the first dataset I found a big repository of various stock data from around 2011 up to 2017 that I scraped from the respective text files, rounded down and turned into integers. I appended it into a large text file and trained the model on it.

The second dataset was the stock data rounded down to 3 significant figures and not turned into integers to see if the network was able to learn sequences

of more complex numbers.

The final dataset was the raw stock data without any processing done to it which was the final test to see if the transformer architecture can perform as well as RetNet, RNNs and LSTMs.

I tested the performance of each of the networks trained on their respective dataset by getting the context from an unseen sequence of numbers taken from a stock and comparing the correlation between the generated sequence and the true data found in the file and finding the correlation coefficient between them.

My implementation of RetNet performed significantly worse than my transformer. I go into the accuracy in the results section, but the training time was significantly increased and got exponentially worse the bigger the network got. The long train time was largely due to my inability to adapt it to training on a GPU, however even a simple Transformer trained on a CPU took less time to train.

4 Network Structure

In this section I describe the network architecture and parameters for each of the different problems I tried to solve.

4.1 Tokenisation

Each individual symbol (numbers, commas, dots and spaces) is encoded into a token. The alphabet used for encoding takes every symbol/character in the dataset and attaches a value to it which allows the transformer to output any language and any symbol.

Usually however, you would encode each number as a different token rather than each symbol which would increase the accuracy, but would also greatly increase the size of the vocabulary which would increase both the size of the network as well as its training time. I chose to tokanise symbols to save time and computing power.

4.2 Embedding

Embedding is a fundamental technique in natural language processing and machine learning that enables tokens to be represented as positional vectors within a latent space. This encoding methodology allows the preservation of semantic relationships between tokens, allowing values close together to be mapped to proximate positions within the embedding space. In the context of stock prediction this gives Transformers the advantage over other architectures that

can't use positional embeddings. By finding the inherent relationships and patterns within between the stocks, transformers possess the capability to discern the distances between different stock values within the latent space. Thanks to that, this enhanced understanding of the underlying structure enables more effective prediction of future stock market trends with greater precision and efficiency.

4.3 Attention

The attention mechanism is described as ' mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.' in [1].

The attention mechanism is a fairly new algorithm that selectively focuses on important elements of input data which improves accuracy and computational efficiency by emphasising points of data that it deems important, much like memory cells in LSTM networks that incerases computation speed by ignoring less important data.

I believe using attention mechanisms in more architectures such as simple convolutional neural networks could improve training speed and accuracy in the long term.

An example of an attention mechanism is the dot-product attention which I use in this implementation of a transformer. It is much more space-efficient and faster than additive attention due to high speed matrix multiplication.[1]

4.3.1 Self-Attention

Self-Attention is used to weigh the importance of different elements within a sequence against each other in this case it weighs the importance of different stock values against one another.[3]

By applying self-attention across all elements in the sequence, the model can effectively capture dependencies between distant elements and learn complex relationships within the data. This allows the Transformer to learn relations between stock values that are distant to each other and allows it to keep the correlation over time more effectively.

Self-attention is calculated in this paper by first taking a sequence of input vectors X=$\{x_1, x_2, ..., x_n\}$ where x_i represents a stock value at a given point in time which has been turned into a token encoded using a vocabulary.

For each token three vectors are calculated: Query(q_i),Key(k_i),Value(v_i). Each

one is obtained by applying learned linear transformations.
$q_i = (W_q) \times (x_i)$
$k_i = (W_k) \times (x_i)$
$v_i = (W_v) \times (x_i)$

(W_q),(W_k),(W_v) each represent a learned sets of weight matrices corresponding to queries, keys and values.

A dot product between the query(q_i) and the key(k_i) is calculated to evaluate how much attention needs to be paid to other tokens in relation to the current token.

Attention score(q_i,k_j) $= q_i \cdot k_j$

We use this equation to calculate the attention weights:

Attention(Q,K,V) = softmax $(\frac{QK^T}{\sqrt{d_k}})$V

Where K represents a matrix of all key vectors $\{k_1, k_2, ..., k_n\}$, V represents the matrix of all value vectors $\{v_1, v_2, ..., v_n\}$ and d_k represents the dimensionality of the key vectors. To prevent vanishing gradients from the growing magnitude of the dot product it is scaled by $\frac{1}{\sqrt{d_k}}$.
The softmax turns the attention matrix into a probability that sums to 1 which represents the importance of each token in the sequence relative to the current token.

Before multiplying the attention with the values we apply a mask that masks out all values in the input of the softmax which correspond to illegal connections.[1]

Lastly, the output that is passed forward is a matrix multiplication of the softmax and the values, calculated by this equation:
$Output = V \times Attention$

4.3.2 Multi-Head-Attention

'Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.'[1] The transformer uses this to have multiple representations of attention and allows the transformer to discard pointless data much more efficiently, for example if one attention head focuses too much on noisy data, the other heads can overcompensate for this by learning the more relevant data and outweigh the one incorrect head.

Multiple heads also allow increased robustness by adding more heads I can increase the accuracy of the model.

One big benefit of multi-headed attention is that while increasing the size of the network would usually affect training time significantly, thanks to them being able to be trained in parallel, the amount of heads doesn't affect training much as much as adding layers to a recurrent neural network.

4.4 Structure

The Transformer architecture consists of a decoder and an encoder alongside the attention mechanism that is necessary to learn the relation between different tokens.

Each head of the multi-head attention mechanism consists of 3 E, H Linear layers where E is the number of embedding neurons and H representing the head size, these represent Q,K and V respectively. The output of these layers is passed through a buffer of the lower triangular of a matrix of ones of the size B^2, followed by a dropout layer to prevent overfitting.

The FeedForward layer is a sequential layer consisting of a Linear layer of size $E, (4{\cdot}E)$ with a ReLU activated function, followed by another Linear layer of size $(4{\cdot}E), E$, followed by a dropout layer that takes E as an input.

The transformer Block consists of a Multihead attention module of size $X{\cdot}H$ where X represents the number of heads, followed by a feedforward layer of size E, then two layers of layer normalisation of size E.

The transformers overall structure starts with an embedding layer for the tokens of size $V{\cdot}E$ where V represents the vocabulary size which is the number of different characters in your dataset which in my case was 13 representing every number, commas, full stops and spaces. This is followed by a positional embedding of size $B{\cdot}E$ where B represents the block size, followed by a Sequential layer of L Blocks, where L represents the number of layers, of size $E{\cdot}X$, followed by a Layer normalisation layer of size E, finishing off with a Linear layer of size $E{\cdot}V$.

4.5 Parameters

This is the equation to get H:

$$H = \frac{E}{X}$$

Parameter	Small Transformer	Big Transformer
Batch size	64	64
Blocksize(B)	200	300
Max iterations	5000	5000
Evaluation interval	50	50
Learning rate	3e-4	3e-4
Embedding neurons(E)	320	512
Number of Heads(X)	6	7
Number of Layers(L)	6	12
Dropout	0.2	0.2
Optimizer	AdamW	AdamW

Table 1: This table shows every parameter used for each of the transformers.

4.6 Training

Each network was trained over 5000 steps. The large models' steps took around 5 minutes while the small networks steps took around 20 seconds.

By using both positional embeddings and token embeddings, the transformer can learn where in the sequence the specific token should be and what its value should be in relation to all the other tokens that precede it.

In these experiments, a single character is encoded as a token rather than having a number being encoded as one. This allows the network to be far smaller while still learning the necessary data. The moment I use numbers as tokens the size of the network increases exponentially both in size and training time.

The training process begins by taking a block-sized chunk of the stock values and commas(the window), which have been encoded into tokens and turned into tensors so they can be trained on a GPU. The data is passed through the various multi-headed attention modules(see Section 4.3.2 for explanation of the process inside those) where the network learns the positional embeddings and tries to predict what the next token in the sequence is, the error is then propagated back through the network. The loss is calculated by using cross entropy which takes the token generated and compares it to the target token that was suppose to be generated.

$$CrossEntropyLoss = -\sum t_i \log(p_i) \tag{3}$$

Equation 3 This is the equation for cross entropy loss where t_i=target token and p_i = softmax probability for the predicted token.

The network then moves the theoretical window by one token and by taking the block-sized chunk without the first token from the previous windows first value but instead with the value it tried to predict as its last value it tries to

predict the token that follows. By repeating this process the network learns how various stocks look and can then replicate the correlations.

5 Results

In this section I describe the results of training each of the networks described in Section 2.

5.1 Sin function

The networks trained on the sin(x) function all managed to achieve a correlation coefficient of 1 after enough training time as seen by Table 2. This shows that the transformer architecture can easily learn infinitely repeatable deterministic sequences of numbers.

5.2 Fibonacci sequence

The networks trained on the Fibonacci sequence all achieved a correlation coefficient of 0.998 and show Transformers can learn deterministic non-repeating sequences of numbers as seen in Table 2. However the network overfits and cannot generate a sequence from a fibonacci number it has not seen, meaning it learned the sequence it has seen but cannot replicate the correlation of the fibonacci sequence further.

5.3 Artificially Generated Stock Data

The small network trained on the artificially generated stock data managed to achieve result of 0.72 on the training data and 0.593 on test data showing the network overfit, but not significantly. This shows the potential of this architecture to learn sequenced data.

The large version of this network achieved a correlation coefficient of 0.053 on training data and 0.026 on the test data, showing the network is guessing mostly at random when it comes to artificial data.

This is most likely due to the fact that the artificially generated data has little to no actual correlation, rather it is a stochastically generated string of numbers. This was anticipated, however still unfortunate since it would've allowed the network to have infinitely more data to be trained on.

The correlation coefficient dropoff can be seen in Figure 5. It starts off with a drop by 0.23 which is a significant drop, especially compared to the transformer dropoff seen in Figure 3. This clearly shows that the artificially generated stock data has no continued correlation which means the transformer simply learned to replicate the sequences and not true correlations.

5.4 Real Stock data

The first small transformer trained on the stock data that was heavily processed by rounding and being turned into integers only managed to get around 0.058 on average in the correlation coefficient matrix on the training data and a 0.41 on the test data.

The large version of the first network performed worse. This is most likely just luck and in reality neither of the networks could predict the Real Stocks 1 dataset in any way.

The second small transformer that was trained on stock data that was rounded down to 2 decimal places and achieved a correlation coefficient of 0.173. Tested on training data which was an improvement from the first network showing that more detailed data is harder to learn, taking nearly 5 times as long, but was much more accurate for predictions.

The large version of the second network performed worse much like the first one, showing the size of the network might be more of a hindrance than help.

The last small transformer that was trained on full data that wasn't rounded down or compressed in any way, achieved a correlation coefficient of 0.12 on training data, showing that it struggles to learn more intricate data if the network was too small, while having a surprisingly good test correlation coefficient of 0.699 which nearly achieved the desired result.

The large version of the last network performed much worse on testing data, but followed the trend of getting better at predicting the training data alongside achieving a higher correlation coefficient on the test data than the second Large Transformer.

5.5 RetNet Comparison

My implementation of RetNet struggled to achieve any desirable results, with the highest correlation coefficient at a -0.012, the network could barely replicate an output that resembled stocks, and not a string of numbers and full stops.

While the performance was disappointing and training time was significantly higher, the inference step was a lot faster than that of the transformer, showing the main benefit of the architecture. However for this task, high speed inference is not needed, while bigger models and faster training time is much more important.

Model	Data	ρ(Tr)	ρ(Te)	Training Time	Overfit
Small Transformer	Sin Wave	1.0	1.0	5 minutes	no
Small Transformer	Fibonacci	0.998	0.051	7 minutes	yes
Small Transformer	AG Stocks	0.72	0.593	40 minutes	yes
Large Transformer	AG Stocks	0.053	0.026	407 minutes	yes
Small Transformer	Real Stocks 1	0.058	0.41	7 minutes	no
Large Transformer	Real Stocks 1	-0.044	0.029	83 minutes	no
Small Transformer	Real Stocks 2	0.173	0.599	32 minutes	no
Large Transformer	Real Stocks 2	0.010	-0.085	323 minutes	no
Small Transformer	Real Stocks 3	**0.12**	**0.699**	40 minutes	no
Large Transformer	Real Stocks 3	**0.236**	**0.136**	425 minutes	no
Small RetNet	Real Stocks 3	-0.012	0.0	340 minutes	no
LSTM	Real Stocks 3	0.677	0.724	2 minutes	no
RNN	Real Stocks 3	0.233	-0.341	3 minutes	no

Table 2: Performance of each network. ρ = correlation coefficient average over 10 trials. AG = Artificially Generated. Tr = Training dataset, Te = Testing dataset

5.6 RNN Comparison

My RNN model was unable to replicate a sequence that resembled a stock, it created a huge fluctuation in the beginning before levelling off completely. I don't believe RNNs can be compared to Transformers anymore as they simply cannot perform as well as other, more recent, architectures.

5.7 LSTM Comparison

The LSTM architecture clearly is able to replicate sequences as good if not better than transformers, however much like the RNN architecture the sequence didn't resemble the stock I tried to predict therefore I cannot recommend it being used over transformers. While LSTMs can give a reasonable estimation of the direction a stock will take, using a lag plot shows that after even 5 iterations the correlation diverges by 0.27 compared to the real stock value.

6 Conclusion

The nature of stocks is that they are random, therefore very hard to predict. No machine can truly predict a stock by simply learning sequences of correlated numbers. This model however proved that it can learn the trajectories of where the stock will go with increasing accuracy. As I scaled the model, it began performing better and I believe that a large enough model with enough data could semi-consistently make the correct assumption on when to buy and when to sell. This should not be used as a basis for a portfolio however I believe it can be used as additional information to guide a decision on whether to buy or

sell at a given point.

The training time of the Transformer can be greatly decreased by tokenising the symbols such as numbers or letters rather than using entire words or values as tokens since the size of the network scales with the size of the vocabulary, this however decreases accuracy due to the networks ability to produces incoherent sequences of numbers which can be avoided using the more computationally expensive tokenisation method.

Compared to other state-of-the-art architectures, the Transformer generated sequences that most resembled a real stock and diverged the least when measured using a lag plot. With the networks ability to replicate less random data increasingly well and reaching 1.0 accuracy on the sin(x) wave, I believe it demonstrated that the Transformer architecture could be used for various examples of time-series data.

7 Reflection

Looking back on the project there were some things that could've been done better and I talk about those things in this section.

7.1 Struggles with the project

Firstly, finding stock data was much harder than I was expecting. There are very few open-source stock datasets that are large and useful. A lot of them were mainly to do with gold, but luckily I found a very large dataset, however the dataset is very outdated meaning it couldn't be used for prediction right now since having current data would allow me to actually test the network by giving it some money which I go into further in section 7.4.

Besides the data being scarce I also encountered very long training time for the larger Transformers which deterred me from training it to the fullest extent. I believe if given a better machine I would be able to get actually impressive results.

Lastly I decided, very last minute, to implement an LSTM and an RNN which could've been explored a lot more and comparing them on many different tasks could've yielded a lot more interesting results.

7.2 What could've been done better

I believe tokenising each symbol rather than numbers played a role in the networks ability to produce coherent sequences. When training the network some-

times produced unusable numbers and therefore those earlier iterations couldn't have been tested. By tokenising the numbers, the network would always produce a sequence that resembled a stock, but unfortunately tokenising numbers would exponentially increases training time which was already limited. I would love to try this approach with more time and maybe a better machine.

Using the artificially generated data was quite interesting and added complexity to the project, however it turned out to be pointless. Stock data is already quite random and putting inherently random data through a markov chain to re-create that data the sequences had no correlation between each other which is shown in Figure 5. I believe trying to generate the stock data using a transformer that then uses the data it generates to train could be interesting a possible extension of this. Unfortunately it turned out that the networks simply learned the data it was trained on rather than the correlations and if trained for long enough, it would overfit.

7.3 If I knew what I know now

Firstly I would create a lot more networks with varied parameters to test each networks ability with much smaller size increases. I would perhaps use a genetic algorithm to automatically create networks that could be fit for the task.

I would definitely test RNNs and LSTMs more to compare the previous state-of-the-art at every task and not just the Real Stocks 3. I would also give RetNet more testing in hopes of finding a version of it that generates coherent stocks.

Every day new architectures are being developed for time-series data tasks. I would like to test the Transformer against every possible architecture in hopes of finding something that can compete with it at this task. This would also force the transformer to be pushed further and hopefully improve its ability to predict stocks.

7.4 Future work

If I could continue this project I would like to try using the transformer for more varied time-series data like trying to predict weather data or maybe even temperature. This type of data is quite easy to acquire and would be very useful if the transformer was able to predict it somewhat consistently.

I could also try using the attention mechanism in more architectures to improve the ability to parallelise in RNNs or maybe use them instead of convolutions in basic Convolutional Neural Networks.

Using the Transformer with more up to date data and trying to predict current stock prices and hook it to automatically buy and sell at advantageous times and seeing if the network is able to make money is the easiest way to extend this project further and could be quite interesting.

8 References

[1]A. Vaswani ıet al., 0220Attention Is All You Need.0221 arXiv, Aug. 01, 2023. Accessed: Mar. 10, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

[2]Md. A. Istiake Sunny, M. M. S. Maswood, and A. G. Alharbi, 0220Deep Learning-Based Stock Price Prediction Using LSTM and Bi-Directional LSTM Model,0221 in ı2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Oct. 2020, pp. 870821192. doi: 10.1109/NILES50944.2020.9257950.

[3]H. Zhao, J. Jia, and V. Koltun, 0220Exploring Self-Attention for Image Recognition,0221 in ı2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA: IEEE, Jun. 2020, pp. 10073821110082. doi: 10.1109/CVPR42600.2020.01009.

[4]M. Roondiwala, H. Patel, and S. Varma, 0220Predicting Stock Prices Using LSTM,0221 ıInternational Journal of Science and Research (IJSR), vol. 6, Apr. 2017, doi: 10.21275/ART20172755.

[5]Y. Sun ıet al., 0220Retentive Network: A Successor to Transformer for Large Language Models.0221 arXiv, Aug. 09, 2023. Accessed: Mar. 10, 2024. [Online]. Available: http://arxiv.org/abs/2307.08621

[6]C. Wang, Y. Chen, S. Zhang, and Q. Zhang, 0220Stock market index prediction using deep Transformer model,0221 ıExpert Systems with Applications, vol. 208, p. 118128, Dec. 2022, doi: 10.1016/j.eswa.2022.118128.

[7]S. Selvin, R. Vinayakumar, E. A. Gopalakrishnan, V. K. Menon, and K. P. Soman, 0220Stock price prediction using LSTM, RNN and CNN-sliding window model,0221 in ı2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Sep. 2017, pp. 16430821116 47. doi: 10.1109/ICACCI.2017.8126078.

[8]T. Muhammad ıet al., 0220Transformer-Based Deep Learning Model for Stock Price Prediction: A Case Study on Bangladesh Stock Market,0221 ıInt. J. Comp. Intel. Appl., vol. 22, no. 03, p. 2350013, Sep. 2023, doi: 10.1142/S146902682350013X.

9 Graphics

In this section I show off some results using graphs. All of these are taken from a sequence the small (Real Stocks 1) transformer generated.

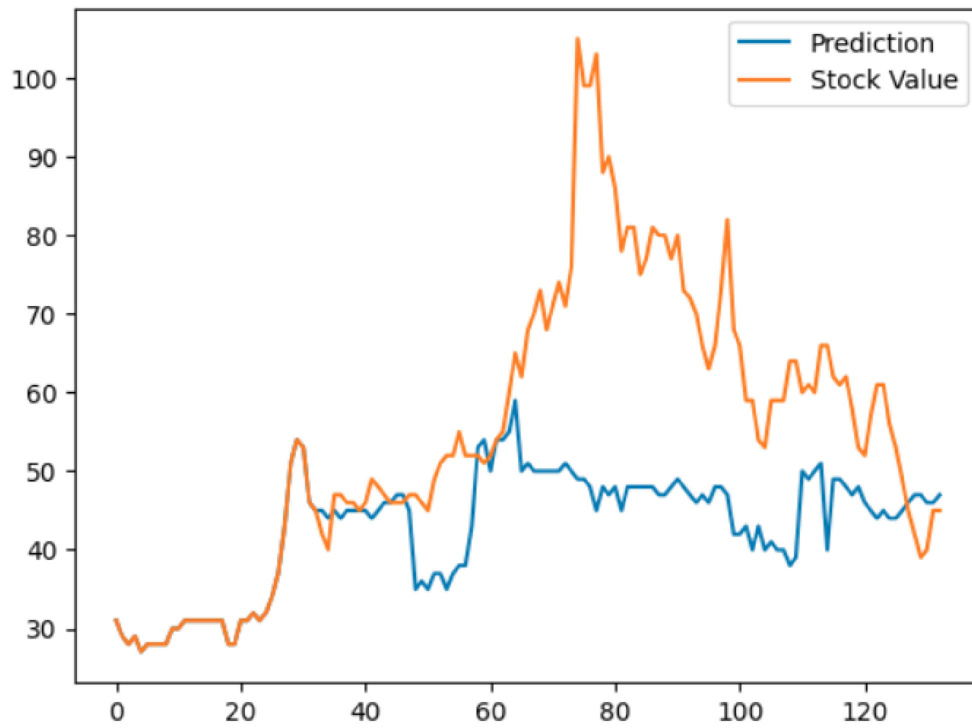

Figure 1: This graph shows the real value of the stock against the predicted value, while the prediction isn't perfect, it shows the network can tell where the stock is going to go in the long term, but struggles with the short term.

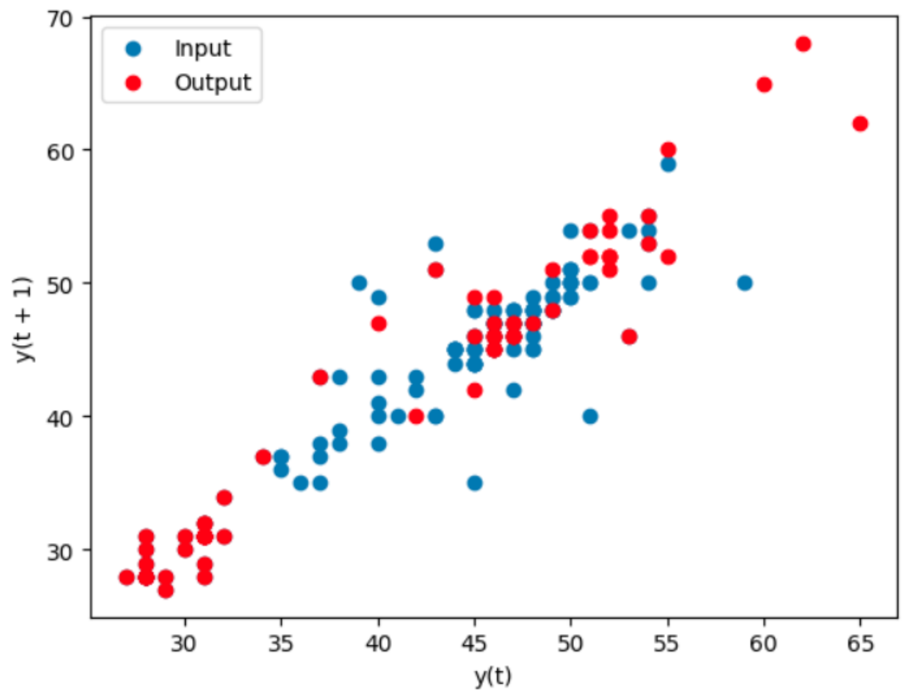

Figure 2: This is a lag plot generated by taking the generated values and the real values over time. The plot shows that while the generated data (output) is dispersed more heavily it still stayed relatively close to where the real stock value(input) is.

	t	t+1	t+2	t+3	t+4	t+5
t	1.000000	0.970818	0.925200	0.876778	0.837725	0.818215
t+1	0.970818	1.000000	0.970656	0.924202	0.881181	0.841600
t+2	0.925200	0.970656	1.000000	0.971614	0.928740	0.888517
t+3	0.876778	0.924202	0.971614	1.000000	0.972405	0.934920
t+4	0.837725	0.881181	0.928740	0.972405	1.000000	0.972895
t+5	0.818215	0.841600	0.888517	0.934920	0.972895	1.000000

	t	t+1	t+2	t+3	t+4	t+5
t	1.000000	0.930968	0.867831	0.801944	0.732877	0.687040
t+1	0.930968	1.000000	0.931130	0.867979	0.802188	0.733461
t+2	0.867831	0.931130	1.000000	0.931179	0.868124	0.802625
t+3	0.801944	0.867979	0.931179	1.000000	0.931269	0.868310
t+4	0.732877	0.802188	0.868124	0.931269	1.000000	0.931337
t+5	0.687040	0.733461	0.802625	0.868310	0.931337	1.000000

Figure 3: This table(dropoff matrix) represents how close the variable at a certain time t is to the first sampled variable. From the table we can tell that the drop of correlation between each time step is significantly higher in the generated sequence(bottom), while the real stock value drops by only 0.4 on average, the generated data has an average drop of 0.6.

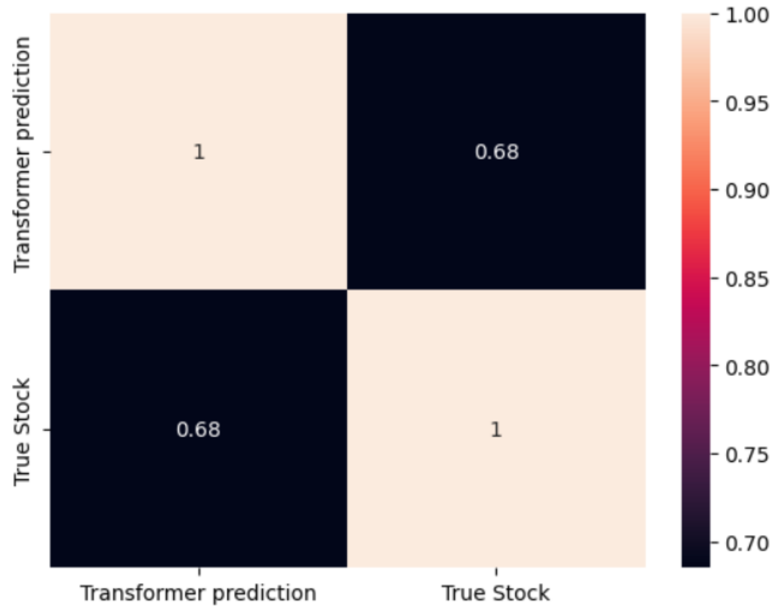

Figure 4: This is a heatmap which I used to visualise ρ

	t	t+1	t+2	t+3	t+4	t+5
t	1.000000	0.778415	0.706110	0.559320	0.375210	0.219926
t+1	0.778415	1.000000	0.780441	0.710564	0.565479	0.382450
t+2	0.706110	0.780441	1.000000	0.782625	0.713349	0.568976
t+3	0.559320	0.710564	0.782625	1.000000	0.784855	0.715510
t+4	0.375210	0.565479	0.713349	0.784855	1.000000	0.786124
t+5	0.219926	0.382450	0.568976	0.715510	0.786124	1.000000

Figure 5: This is a dropoff matrix for the artificially generated stock data showing the correlation coefficient dropping with each time step, a clear sign that stocks generated using a markov chain are inherently random and shouldn't be used for training

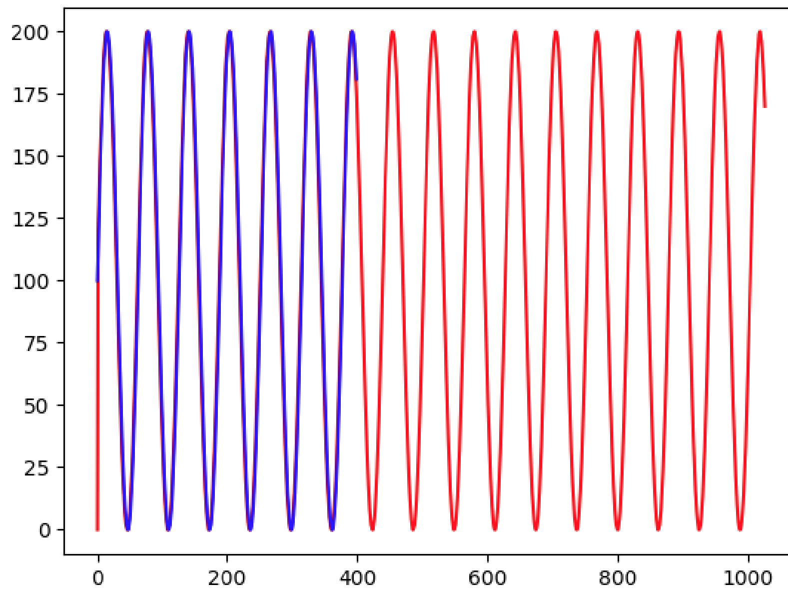

Figure 6: This is a graph depicting my Sin Wave Transformer generating a sin(x) wave from a context. The context ends where the first red line ends and the prediction is depicted using the blue line