Recurrent Neural Networks for Stock Price Prediction

Mustafa Mohammadi
The University of Adelaide
a1838795@adelaide.edu.au

Abstract

Standard Recurrent Neural Networks is the go algorithm for modeling sequential data with small time gap. Since, standard RNN cannot handle longer temporal sequences because of limitations of memory, vanishing and exploding of gradients during the backpropagation, we experiment with RNN architectures which have overcame those limitations, such as LSTM and GRU. The experiments performed in this paper evaluate the performance of LSTM and GRU on Google Stock Price prediction, with their strength and limitations noted.

1. Introduction

This paper presents Recurrent Neural Network (RNN). RNN is the state of art modelling algorithm for reasons stated below. When we refer to RNN, we imply standard RNN. However, there are many variations of RNN which correspond to various needs of the different types, such as One-to-One RNN is suited for tasks such as image processing, where every input pixel corresponds to an output pixel. Similarly, the other RNNs will be introduced in Section 2.2. In spite of the flexibilities of RNNs, it does not perform well with data assuming i.i.d (independent and identically distributed), which renders standard RNNs handicapped when dealing with temporal and spatial data such as time series, video/audio sequences, speech and machine translation modelling. The handicap-ness of RNN comes to play when there exists a long time gap between the first and last input, which means standard RNN does not have the necessary memory to remember dependencies over time. Furthermore, the longer temporal dependencies are sensitive to backpropagation due to exploding/vanishing gradients. Consequently, from various RNN architectures, in this paper we suggest two architectures, Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) to remedy the memory issues common in RNN by remembering historical time steps and the errors to propagate across time through a carousel without diminishing or exploding, both in forward and backward passing.

2. Background

2.1. The need for RNN

Deep Neural Networks, particularly those inspired by Deep Belief Networks (DNN) built by stacked, restricted Boltzmann Machines, and Convolutional Neural Networks (CNN) exploit local dependency of visual information have recently made major contributions to the application of prediction vision systems. The strength of their application is limited because of their assumptions of independence. The assumptions of independence or i.i.d is common in regression methods such as linear, logistic regression, and Support Vector Machine and Artificial Neural Network (ANN) and further with CNN.

Such assumption is not a weakness if each example is generated independently. In methods such as Vanilla Neural Network, after each data point is computed, the entire state of the network is lost and the model does not retain the memory of previous step. This implies that the previous step will not affect its future state, which is not a problem if the data points were not related temporally or spatially. However, if the data points were related in time and space, there is much lost information assuming independence, therefore the aforementioned methods with such assumption is invalid for temporal and spatial data. Examples of these types of data include individual frames of video or frames, a specific window of audio, words extracted from sentence; and daily, monthly or seasonal stock value, is where assumptions of independence fail, because individual bit in a sequence makes little sense without the prior or after the bit. For instance, today's Google stock value is not independent of yesterday, because internal operations of Google is based on current, past, and future projects, which means what Google does now, determines its future tomorrow and the years ahead, therefore the temporal nature of data requires temporal or sequential method of evaluation.

2.2. What is RNN?

RNN are feedforward Neural Networks augmented with an edge to itself called recurrent edges. At time t, a node with a recurrent edge, receives input from current point in time x_t, and also from hidden state (h_{t-1}) from the networks previous state. The edge to itself, is inclusion of other time steps in the sequence, which represent the notion of time and order to the model. The output $\hat{y}$ at each time t, is computed using the newly computed h_t.

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b),$$

$$y_t = h_t,$$

Given that Standard NN is a one-to-one network architecture of fixed length vector, it is not flexible as not every data pattern is one-to-one. For example, speech recognition where the input is audio signals, output is a sequence of words. Music generation where the input is a genre, output is a music of certain length. Sentiment classification where the input can a sequence of words, output is a single star ranging between 1 to 5, Machine Translation where the input is a sequence of words in French, but output is a different count of sequence in English. RNN support these types of data pattern with unique architectures. For example,

1. one-to-one network of fixed length to networks
2. one-to-many for patterns such as music modelling,
3. many-to-many (not limited to equal length sequence) for machine translation or name entity recognition,
4. many-to-one for sentiment analysis.

2.3. Why long-range dependencies in a sequence is a problem for RNN?

Computational Complexity. As indicated above, many patterns in the world are temporal. Learning structure in time-extended sequence is computationally cumbersome because an input pattern may not contain all the information relevant to a task. It may be contained in a different example or in a different sequence in time. Since it is not available at this instant at time t, then a layer or cell must hold certain contextual information from history until a later point in time to be used [2].

Models with independent assumptions precludes long term dependencies such as DNN. However, Markovian model offers a window of hope, standard computation, where each state is immediately dependent on previous step in Hidden Markovian Models (HMM) can be extended to a step where at any time step, a step can contain information of arbitrarily long window; however, it is computationally impractical because the procedure grows the state space exponentially O(2^N) with size of the window.

If a given data is related temporally or spatially, the variation in time and space maybe over short term or long term. Given the limitation of Markov Models, RNN is justified for the following reasons. Just like Markov Models, any state in vanilla RNN depends only the current input and on the state of the network at the previous time step. However, the hidden state in RNN can contain number of distinct states from an arbitrarily long context window. While the RNN network grows exponentially with the number of nodes, the computational complexity grows quadratically O(N^2).

Vanishing/Exploding Gradient: a problem of local error backflow. Why gradients vanish or explode? Consider an English sentence being translated to French. Assume the sentence has a length of twenty words, there is a gap of 18 words between the first and last word in the sentence. This is long temporal gap for the network to remember if it wants to predict the 21st word. As there is long gap, there is a high chance of forgetting, so let's say, the network makes a mistake in predicting the 21st word. To adjust the weights with the respect to the first input, we must backpropagate along the path from the 21st word, all the way to back to the first word, this means that the error signal has to travel back in time in very long time steps.

If weight = 1.5 and time = 20,

$$\frac{\partial h^{(T)}}{\partial h^{(1)}} = w^{T-1} = 1.5^{20} = 3320,$$

On the other hand, if weight = 0.9 and time = 50.

$$\frac{\partial h^{(T)}}{\partial h^{(1)}} = w^{T-1} = 0.9^{50} = 0.0051537752.$$

The backflow gradient matrix is called the Jacobian. It can explode if the largest eigenvalue of the weight matrix is larger than 1, and diminish if less than 1. The exploding gradient is not issue in the case of forward propagation as the non-linear activation functions squashes the value in certain range, preventing blow up. However, since backpropagation is an entirely linear computing task, it can blow up easily, just as easy to diminish, depending on weight vectors values.

Solution to Exploding Gradients include 1) Gradient Clipping and 2) Truncated Backpropagation.

Gradient Clipping is an algorithm that prevents gradients from blowing up by a simple trick which is to scale the gradients to be a particular value n [12]. If $\| g\| > n$, g is the gradient, we reset g to be:

$$g = \frac{\eta g}{\|g\|}$$

Truncated Back Propagation (TBTT) is backpropagation through time but instead it is partially back propagating through the sequence. Truncation saves computation time as it relieves the need to backpropagate through the entire

sequence at each step. The truncated estimates are biased and therefore may not converge. Further, truncation favours short term dependencies, which is not the best solution in the context of RNN because of the data with possibly long temporal relations [12].

Solutions to Vanishing Gradient. Backpropagation fails due to decaying error in extended time sequences. Given that many real-world phenomena are temporal, speech, music and unfolding events in the context of stock price prediction, thus learning in extended temporal structure is impossible if the context is not reached as a result of decaying error. There are algorithms to remedy the decaying error such as back propagation through time (BBTT) or real time recurrent learning, but with both algorithms error signals flowing back in time, either blow up or vanish. In the first case, it may lead to oscillating behaviour, and in the second, it may stop to learn all together.

In studying asymptotic behaviour of error gradient as a function of time lags, the case of RNN is considered and the following remedies are suggested due to the limitations of gradient descent as a search procedure for finding optimal weights in RNN [12].

1. Time Constant
2. Long Short-Term Memory

2.4. Time Constant

We will consider Time Constant as a stepping stone algorithm in understanding how remembering diminishes as a function gap/lag.

Current input, maybe connected in time with supposedly historical input; here Mozer presents a situation where back propagation fails. It involves remembering an event across time with intervals.

The input of the sequence is A, B,C,D,…,X,Y. The task is to predict the next symbol in the sequence. Each sequence starts with either X or Y, called trigger symbol. A sequence starts with one of the trigger symbols, and is followed by a fixed sequence ABCDE and the sequence is ended by the other trigger symbol. Examples are… XABCDEX, YABCDEY. Let's say in our first sequence XABCDEY is presented. X is the first trigger symbol. To predict the next symbol, which can by ABCDEY, it must at least remember X, so that it can recall it in the future which is five-time steps later.

There are two trigger symbols, X and Y, but between the trigger symbols are ABCDE called the gap. The goal here is to train different networks on different gaps, and to observe the difficulty of learning with different gaps. A: All input sequence has same length, consisting of X or Y, followed by gap(XABCDXEFGHIJK), in this case, gap of 4. B: All input sequence has same length, consisting of X or Y, followed by gap(YABCDXEFGHYIJK), in this case, gap of 8.

Each training consisted of two sequences, one with X, the other with Y. Different network of same length were trained on different gaps. For one input trigger symbol, there is one output trigger symbol and ten context units. Twenty five replications of each network were run with different random initial weights. The result Mozer realized was that a sequence with gaps of 4 or more did not learn accurately.

Table 1: Learning contingencies across gaps

gap	*% failures*	*mean # epochs to learn*
2	0	468
4	36	7406
6	92	9830
8	100	10000
10	100	10000

Mozer's " general impression is that back propagation is powerful enough to learn only structure that is fairly local in time" [3]. He further states that neural Network could master the rules of composition for notes within a musical phrase, but not found rules of relationship between a local to global notes. It is suggested that reduced description of sequence that makes global contours of the dataset more explicit and detectable. Which can be achieved by using hidden units that operate with different time constants. Mathematically he called it "Smoothed and Compressed", because with large time constant, for example pi = 0.8, Ci(t) holds onto much of the previous information and thereby averaging the response to the net input over time.

With small time constants, the memory its holding, decays and takes current inputs [5]. And if the time constant is large, it is sluggish and takes little current input. If time constant is 1, it holds the memory for potentially infinite time lags [4].

A task requires an Artificial Agent(AA) to think and act accordingly at various time scales. Suppose AA pours pours apple juice from a bottle into a glass. In short time scale, as described by Mozer, speeding up the AA by two fold or multiple times the AA normal speed, the AA has to change its motor commands correctly and quickly given a pouring situation or context. Depends on how many glasses, where the glass is, how is the glass shaped and in what position is the glass's original orientation. While, in longer time scale, AA has to continuously change its mode to a different state such as taking glasses, pouring the juice and handing it over to someone who drinks the apple juice.

Performing multiple time step, with different time scale in RNN with static neurons is difficult because static neurons' time scale is equivalent to time step size, and if all time step size or neuron have same time scale, it is sacrificing a representation of individual element for overall contour of the sequence which is a speedy version

of pour the juice and give to someone. There is much loss of information locally. For RNN, adaptive method of time constant is essential so that it can have different time constant for various time scales. Direct gradient descent updates the time constant leads to vanishing/exploding gradient depending on the time constant.

In the naive approaches described above, to avoid vanishing gradient we achieve constant error backflow through a single unit J back to itself is to make sure the time constant equals 1.

$$f_j'(net_j(t))\, w_{jj} = 1.0.$$

The weaknesses of the naive approach as described above are:

1. Input weight conflict: Let additional single input weight to equal wji. Assume total error can be reduced by switching on unit J in response to certain input, keeping unit J open for the time being until its useful. Wji is used simultaneously for storing the current signals and protect it from the incoming signals. This makes learning difficult, more like unit j becomes a storage with a short-lived memory, as its state is being changed with incoming signals constantly.

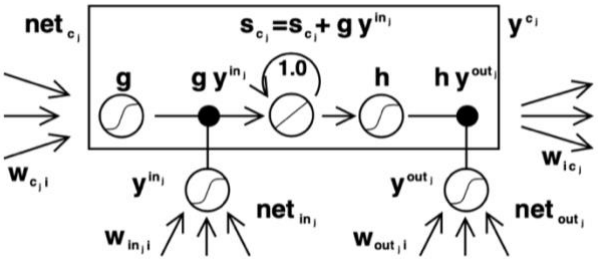
2. Output weight conflict: Assume unit j is switched on, have some previous memory which means its non-zero. Since its non-zero, and a signal k is outgoing from j. Since unit j is switched on, its receiving conflicting value and is constantly disturbed, therefore protecting outputting value k from j is a conflicting task.

Input and output conflict are local but their effect compounds with time. Therefore, the naive methods does not fare well unless it is involving local input-output and nonrepeating input patterns. To ensure constant error flow and protect stored information in unit j, we introduce Long Short-Term Memory. Which does not protect us from exploding gradient but instead from vanishing gradient.

3. Methodology

3.1. LSTM

Long Short-Term Memory is a recurrent network architecture with gradient based method that overcome error backflow problem. LSTM's computational complexity per time step is O(1). Constant Error Carousel (CEC) is the central feature of LSTM. The CECs solve the vanishing gradient error as it remains constant.

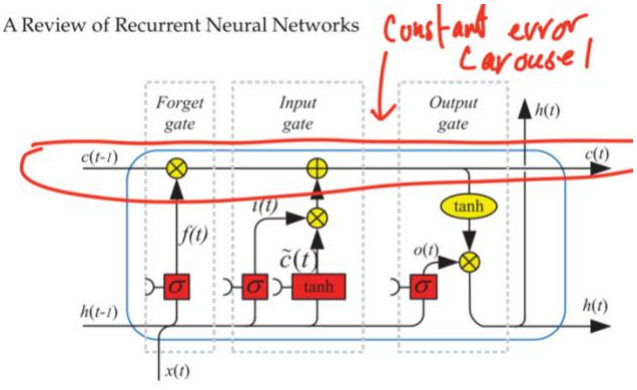


Original LSTM before the Forget Gate [15]

$$i_t = \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i),$$
$$\tilde{c}_t = \tanh(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}),$$
$$c_t = c_{t-1} + i_t \cdot \tilde{c}_t,$$
$$o_t = \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o),$$
$$h_t = o_t \cdot \tanh(c_t),$$

Original LSTM algorithms [16]

How do LSTM works? The basic units of LSTM are memory cells. Each memory cell at its core has a self-recurrent linear unit the CEC, whose activation is called the cell state. The CEC as mentioned above eliminates the potential of decaying error signals when propagated through time, back or forth. The sigmas are multiplicative gates that decide if content is important or not. Ct is the linear history carried by constant error carousel. The information that is carried by Ct is only affected by the gates. The gates either add information to Ct or remove. Cell Memory is the historical data from previous time steps but is only considered as the hidden state of Ct once it goes through the gates. CEC carries error flow in time. While the purpose of the Gates are to learn nonlinearity of the sequential data. Gates are sigmoidal units with inputs ranging between 0-1, controls how much of the information to let through.



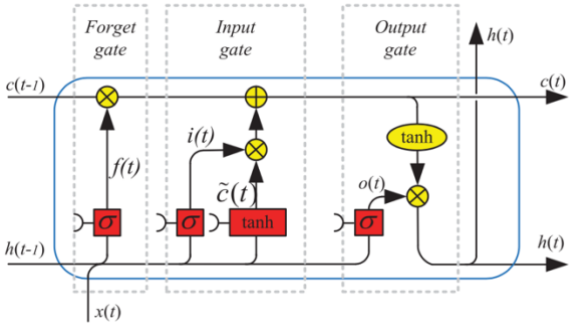
Constant Error Carousel in Red [17]

LSTM with Forget Gate 1

Input Gate: Input gate introduces new pattern or values. Input gate has sigmoid activation and decides on the new detected pattern. It could be that there is a closing bracket maybe, given the hidden state, if found that there is no opening bracket at all, then. Make sense to ignore the new input which is closing bracket.

Output Gate: The output get decides how much of the cell state is allowed to the hidden state which will be used as one of the inputs in the next time step. Which means output gate decide if the content memory is relevant or not to be passed on.

Learning to Forget: Forget Gate. LSTM can learn to bridge time legs in excess of 1000 discrete time steps by enforcing constant error flow through constant error carousels (CECs) within cells. However, as mentioned, LSTM works well with discrete time steps. However, it fails to learn to process very long, continuous time series correctly that are not already segmented into fixed chunks with clearly defining beginning and endings. Since the cells process input, and output, the cell memories carried into the future cells, the CECs memory grow without bounds. Therefore, adding the forget get, the CEC is protected from both forward flowing and backward flowing error by input and output gate. Which means the CEC neither grow or decay, when there is no inflow of input to the cell. Forget Gate multiplies with Ct-1, decide what to forget or keep. Forget get is concatenation of input and hidden state together with their weights [7].

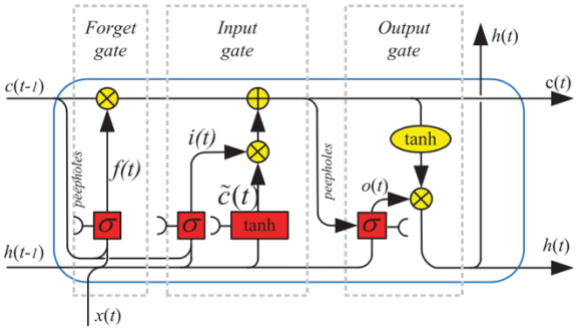


LSTM with forget gate [18]

$$
\begin{aligned}
f_t &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f),\\
i_t &= \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i),\\
\tilde{c}_t &= \tanh(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}),\\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t,\\
o_t &= \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o),\\
h_t &= o_t \cdot \tanh(c_t).
\end{aligned}
$$

Algorithms for LSTM with forget gate [19]

Peephole Connection. Peephole is built on LSTM with forget gates. In LSTM network , there is a direct connection between each gate and the input unit, however, there is no direct connection between the cell state (ct-1) and the gates. The remedy suggested is add weighted peephole connections from the CEC to the gates of each memory block[7]. To ensure CEC content is not interfered with unwanted signals, during learning no error signals are propagated back from gates via peephole connections to CEC. It is like one way road. CEC content flows to the gates, but nothing flow back to CEC from gates.

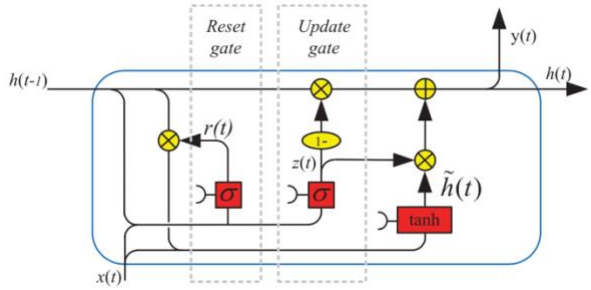


LSTM with Peephole [20]

$$
\begin{aligned}
f_t &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + P_f \cdot c_{t-1} + b_f),\\
i_t &= \sigma(W_{ih}h_{t-1} + W_{ix}x_t + P_i \cdot c_{t-1} + b_i),\\
\tilde{c}_t &= \tanh(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}),\\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t,\\
o_t &= \sigma(W_{oh}h_{t-1} + W_{ox}x_t + P_o \cdot c_t + b_o),\\
h_t &= o_t \cdot \tanh(c_t),
\end{aligned}
$$

LSTM with Peephole Algorithm [21]

3.2. Gated Recurrent Unit (GRU)



Gated Recurrent Unit 1

$$r_t = \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r),$$
$$z_t = \sigma(W_{zh}h_{t-1} + W_{zx}x_t + b_z),$$
$$\tilde{h}_t = \tanh(W_{\tilde{h}h}(r_t \cdot h_{t-1}) + W_{\tilde{h}x}x_t + b_z),$$
$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t.$$

Gate Recurrent Unit Algorithms [22]

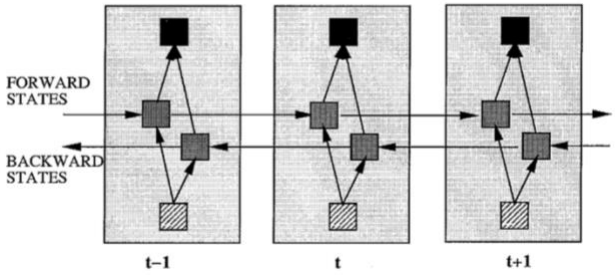
GRU is a variant of LSTM. It integrates the forget gate and input gate of LSTM into one, calling it update gate. The GRU has two gates. Update Gate and Reset Gate, saving it from the computation of retrieval of signal and its associated parameters. Overall LSTM performance is higher to that of Gated Recurrent Unit, since GRU cannot be taught to count, or to solve context free language and does not work for machine translation [8]. In an empirical experiment of LSTM and GRU and traditional Tanh method, evaluating performances on music and speech modelling, it was found that in the case of music modelling, GRU outperformed LSTM. The additional parameters added by LSTM adds computational burden that maybe unnecessary in certain sequential modelling. As in the Music Modelling example above, GRU outperformed LSTM in faster number of updates and actual CPU time. The empirical evaluation's results were inconclusive and noted that the choice of the type of gated recurrent unit is dependent heavily on the dataset at hand [9].

3.3. Bidirectional RNN

Standard RNN's are only to make use of the previous time steps. The limitation of unidirectional RNN is vast such as name entity recognition. For example,

a. Teddy Bears are on sale!
b. Teddy Roosevelt was a president

To predict whether Teddy is part of a person's name or not, knowing half the information is can lead us to false answer as indicated above. In a, Teddy is not part of a name, while in b teddy is part of a name. Since standard RNN does not provide us information about future or time step (t+1), which means at time t, we cannot know whether the next step is Bears or Roosevelt. Therefore, having just having Teddy can lead to less success in prediction. This is where bidirectional RNN comes to rescue. BRNN can be trained both forward into the future (traditional RNN) and in reverse order, which is from the future back to historical time steps. BRNNs are the staple method of modern learning technique, such as BRNN can be combine with LSTM, and GRU and other variations of LSTM and GRUs [11]. Below are the inner workings of simple BRNN.



Standard Bidirectional Diagram [23]

4. Experimental Analysis

4.1. Data Exploration & Preparation

The train data provided is a time series dataset from 01/01/2012 to 31/12/2016 where each timestep is a daily observation of the Google Stock Price. Each timestep contains the open, high, low, and close stock price (unit: US Dollar) and the stock Volume which is the number of shares traded on that day (unit: number of shares). The test dataset consists of data from 01/01/2017 to 31/01/2017. Both were retrieved from Kaggle. Figure 1 shows the full dataset plotted over time. It can be seen that Google's stock price has an upward facing trend over the available 5 year data without any clear seasonality. The sudden in the "Close" graph is a result of Google's stock split in March 27th 2014 where each stock was split into two [source]. Despite the sudden drop, we see that the stock price data (Open, Close, High, Low) are highly-correlated and of similar scale while the Volume data is of a much larger scale (1e-7) and much noisier.

The provided train dataset was further spitted into training set (2012-2015, 1006 samples) and validation set (2016, 252 samples) which equated to exactly 80/20 splitting. We then scaled the dataset by normalisation using a Min-Max scaler fitted on the training set to the range (0,1). While standardisation is also often used for data preparation, it is expected that the input data comes from a normal distribution which does not apply for our case. Finally, training will be done the scaled datasets while final

performance will be evaluated on the predictions inverted to the normal scale.

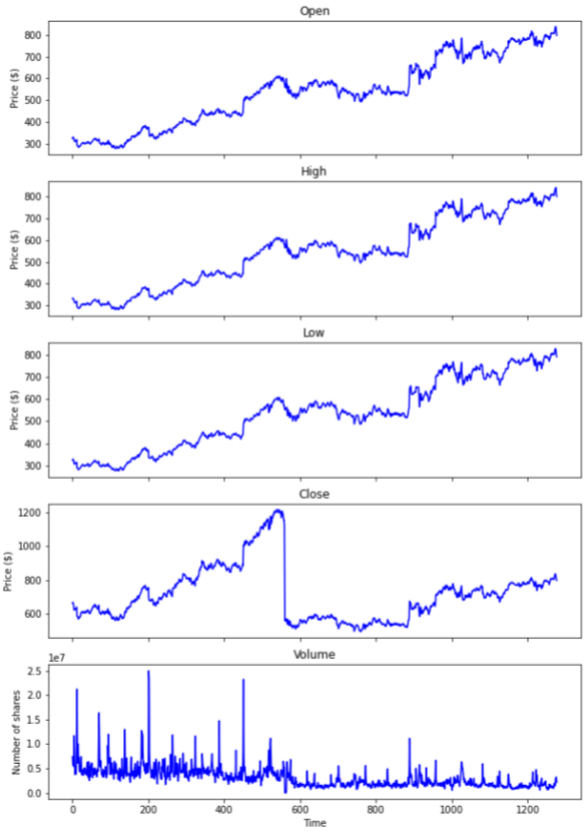


Figure 1. Google's Stock Price data from Jan 2012 to Jan 2017

To turn out dataset into a supervised learning problem, we set our model design choice to be many-to-one. That is, with a given window size or timesteps N, we use the past N days data to predict the next day data. For instance, if the original time series consists of 5 timesteps and we set the window size to 2, then our transformed training set will be:

Training sample 1: data $[x^{<1>}, x^{<2>}]$ label $[x^{<3>}]$
Training sample 2: data $[x^{<2>}, x^{<3>}]$ label $[x^{<4>}]$
Training sample 3: data $[x^{<3>}, x^{<4>}]$ label $[x^{<5>}]$

4.2. Experiment 1: Comparison of Models

The goal of our first experiment is to carry out a comprehensive comparison between the following 6 models: Simple RNN, GRU, and LSTM (all with and without bidirectional connections) by using the same architecture and hyperparameters on the proposed models. The architecture can be illustrated as follows:

Input → Layer → Layer → Layer → Dense

where each 'Layer' can be Simple RNN or Bidirectional Simple RNN depends on the model currently being tested. We did not see the need to have Batch Normalisation in the current architecture due to the following three reasons: 1) we do not apply standardisation for our time series inputs, 2) the many sigmoidal and tanh activations keep the information flow within a certain range, and 3) it is suggested that z-score is normalization is inefficient in handling non stationary time series since the statistics used for the normalization are fixed both during the training and inference[24].

The window size chosen was 60, which represents 3 financial months. The number of hidden units is set to be 32. The number of layers was set to be 3 as stock price prediction is a fairly complicated process, and higher number of sequential layers is said to handle complicated data better. Every sequential layer is followed by a Dropout layer with a dropout rate equals to 0.2 for regularisation. Due to the limitation of computing power, the number of epochs is set to be 100, and Early Stopping is used to halt training when the RMSE on the validation set can no longer be improved. The metric used for evaluating performance is MSE as it is convex and penalises large error. Finally, the models are trained using the Adam optimization algorithm with the learning rate of 0.01. Given the stochastic nature of sequential learning, each model was ran 5 times and the averaged results were reported.

Table 1. Evaluation results

Model	**RMSE on test data**
Simple RNN	2,154,309.24
Simple RNN with Bidirectional	357,576.13
GRU	293,575.24
GRU with Bidirectional	273,680.60
LSTM	326,775.58
LSTM with Bidirectional	296,537.98

Table 1 shows the averaged RMSE of the proposed models on the test data. It appears that Simple RNN with no bidirectional connections did not perform well given its high RMSE. This result is expected as Simple RNN does not retain long-term memories [more explanation source]. On the other hand, it can be seen that GRU with and without bidirectional connections performed best. This result is surprising as we expected LSTM to perform best.

As we inspect the results of GRU model with bidirectional, it can be seen that our predictions captured the correct general trend and local troughs and peaks (albeit slightly delayed due to the large window size). However, it did not capture the precise numerical values of the stock prices or volumes. This is expected since we trained on the data of 2012-2015 and tested on the first month of 2017. That is, we have lost one year of information. Hence, with the current data partition, we do not expect our model to give accurate stock price prediction, but more of the

movement of it.

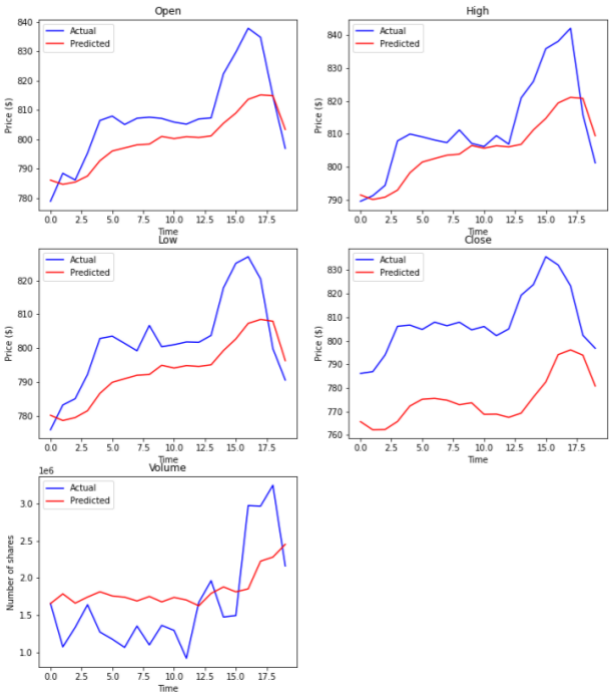


Figure 2. Prediction results by BRNN GRU

4.3. Experiment 2: Hyperparameter tuning

Using the best model found in Section 4.2, we perform hyperparameter tuning on it to find the most effective architecture on training. The hyperparameters tuned include window size, batch size, learning rate, and optimizer. Table 2 shows the levels that were tuned on. It is preferable to also tune the number of layers as well as the number of hidden units in each GRU cell. However, due to our limitations in computing power, we keep the model design from Experiment 1, i.e. 3 GRU layers, each followed by a Dropout layer with a dropout rate equals to 0.2, and each GRU cell has 32 hidden units.

Table 2. Hyperparameters tuned and their levels.

Window Size	[20, 60]
Batch Size	[32, 128]
Learning Rate	[0.0001, 0.001, 0.01]
Optimizer	[SGD, RMSprop, Adam]

Following this, we retrain the model on the train set 5 times and achieved an averaged RMSE of 248,125, lower than that in Experiment 1 but not considerably. However, as we retrain on both the train and validation set, the mean RMSE achieved is further lowered to 234,718, highlights the importance of having up-to-date data in forecasting stock prices. It can be seen in Figure 3 that the results after training on both the train and validation set is numerically a lot more aligned with the ground truth values.

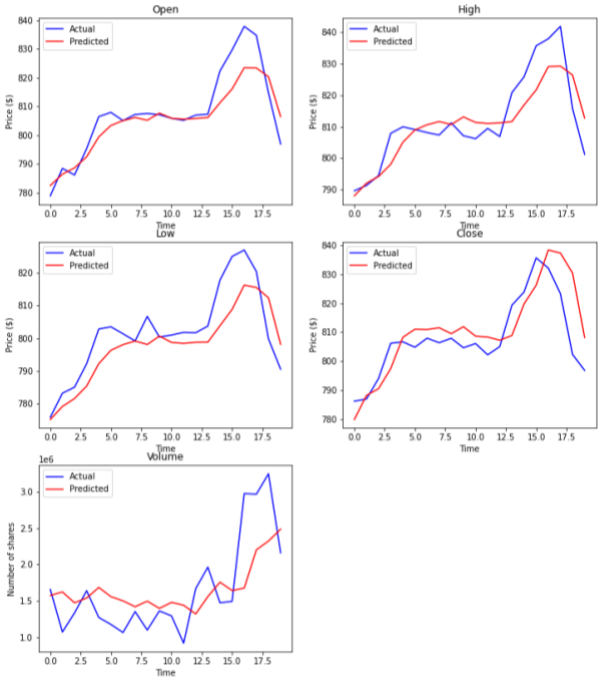


Figure 3. Prediction results by BRNN GRU retrain on both training and validation set

4.4. Limitations

It is important to note that all of our trained models above had trouble learning the Volume feature. As mentioned in the Data Exploration section, the first four features are highly correlated while the Volume feature is not. Hence, it is possible that the trained models have overfit on the noise of the first four features. A possible remedy to this problem is to have two different models, one learns and predicts Stock Prices while another learns and predicts Volume.

Furthermore, given that the stock price data has an upward trend globally (i.e. stock price does not have a maximum value that it can attain), our Min-Max scaler fitted on the train data does not translate well onto the validation and test set. That is, the training set will be scaled within 0 and 1, yet the scaled validation and test set will take on values above 1. To overcome this limitation, a novel deep adaptive input normalization(DAIN) algorithm is introduced, which can adaptively be changed during inference according the distribution of the current time series [25].

5. Code

[https://github.com/parzival2108/LSTM]

6. Conclusion

We demonstrated that bidirectional GRU outperformed LSTM with and without bidirectional connections. This is

a surprising result and proves that we need to extensively test different methods for the given data. As our dataset is very dynamic and noisy, it may be beneficial to use differencing and moving average to smooth out the data as an extension for data preparation. With more time and resources, we should also tune for the number of layers and the number of hidden units in GRU cells. More importantly, we have found that it is not trivial to predict stock price data precisely. More in-depth data analysis specific to said stock needs to be done such as when new products are released and overall users experience in order to make accurate predictions and should not rely on the machine learning models alone.

7. BONUS

LSTM CECs remembers previous time steps. Instead, what if at each time step, instead of remembering it, we try to intentionally forget what is not possible. It saves memory by not remembering and reduces error.

References

[1] Z. Lipton, J. Berkowitz and C. Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1506.00019. [Accessed: 12- Oct- 2021].

[2] M. C. Moser, "Learning structure in temporally-extended sequences", *Proceedings.neurips.cc*, 2021. [Online]. Available: https://proceedings.neurips.cc/paper/1991/file/53fde96fcc4b4ce72d7739202324cd49-Paper.pdf. [Accessed: 13- Oct- 2021].

[3] M. C. Moser, "Learning structure in temporally-extended sequences", *Proceedings.neurips.cc*, 2021. [Online]. Available: https://proceedings.neurips.cc/paper/1991/file/53fde96fcc4b4ce72d7739202324cd49-Paper.pdf. [Accessed: 13- Oct- 2021].Actual Author Name. The frobnicatable foo filter, 2014. Face and Gesture (to appear ID 324).

[4] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997. Available: https://direct-mit-edu.virtual.anu.edu.au/neco/article/9/8/1735/6109/Long-Short-Term-Memory.

[5] T. Matsuki,Shibata, *Learning Time Constant of Continuous-Time Neurons with Gradient Descent*. SpringerLink, 2021, pp. 1-2.

[6] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997. Available: https://direct-mit-edu.virtual.anu.edu.au/neco/article/9/8/1735/6109/Long-Short-Term-Memory.

[7] F. A. Gers and J. Schmidhuber, "Recurrent nets that time and count," Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, 2000, pp. 189-194 vol.3, doi: 10.1109/IJCNN.2000.861302.

[8] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[9] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[10] N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj and A. Iosifidis, "Deep Adaptive Input Normalization for Time Series Forecasting", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1902.07892. [Accessed: 14- Oct- 2021].

[11] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," in IEEE Transactions on Signal Processing, vol. 45, no. 11, pp. 2673-2681, Nov. 1997, doi: 10.1109/78.650093.

[12] John F. Kolen; Stefan C. Kremer, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies," in A Field Guide to Dynamical Recurrent Networks , IEEE, 2001, pp.237-243, doi: 10.1109/9780470544037.ch14.

[13] R. Grosse, *Lecture 15: Exploding and Vanishing Gradients*. Toronto: University of Toronto, 2021.

[14] C. Tallec and Y. Ollivier, "Unbiasing Truncated Backpropagation Through Time", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1705.08209. [Accessed: 15- Oct- 2021].

[15] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[16] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[17] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[18] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[19] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[20] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[21] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[22] Y. Yu, X. Si, C. Hu and J. Zhang, "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures", *Neural Computation*, vol. 31, no. 7, pp. 1235-1270, 2019. Available: 10.1162/neco_a_01199.

[23] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," in IEEE Transactions on Signal

Processing, vol. 45, no. 11, pp. 2673-2681, Nov. 1997, doi: 10.1109/78.650093.

[24] N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj and A. Iosifidis, "Deep Adaptive Input Normalization for Time Series Forecasting", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1902.07892. [Accessed: 15- Oct- 2021].

[25] N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj and A. Iosifidis, "Deep Adaptive Input Normalization for Time Series Forecasting", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1902.07892. [Accessed: 15- Oct- 2021].