RUTGERS ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

Stock Price Prediction With LSTM - RNN

Hua Liu
September 19, 2017

abstract In this paper, I implemented a long short term memory based recurrent neural network, trained and tested with historical stock data of S&P500. The accuracy measured by Root Mean Square Error (RMSE) is around 0.99. In addition, experiments aiming at fine-tuning the LSTM hyper-parameters and developing insights from stock prices are implemented.

1 INTRODUCTION

Stock market prediction is the act of predicting the future value of a company stock prices or other financial instruments traded on an exchange. A successful prediction of stock's future price produces considerable profits. However, stock markets are affected by many uncertainties and interrelated economic and political factors at both local and global levels. According to the efficient-market hypothesis (EMH), stock's prices are a function of all currently available information and rational expectations. Which implies any price changes that are not based on newly revealed information are inherently unpredictable. Burton Malkiel, in his influential 1973 work "A Random Walk Down Wall Street", claimed that stock prices could therefore not be accurately predicted by looking at price history.

While the efficient-market hypothesis is popular among financial academics, actual market experience shows differently. Actually, a large industry has grown up around the implication proposition that some analysts can predict stocks better than others. And there are a lot of big companies spending tremendous amount of money on hiring experts to predict stock prices using statistical models. So, in this article, under the assumption that stock prices are predictable to some degree, I explored to what extend, my prediction method performs.

2 Previous Work

Generally speaking, stock prediction methodologies are consist of three main categories: fundamental analysis, technical analysis (charting) and technological methods. On the one hand, fundamental analysis believes that the stock trends are highly related to the company that underlies. They evaluate a company's the credibility of its accounts as well as its past performance using performance ratios, such as P/E ratio, to aid the analysis process. Technical analysis, on the other hand, predicts the future price of a stock based solely on the trends of the past price. During the analysis, numerous patterns such as the "head and shoulder" and "cup and saucer", as well as statistical techniques such as the exponential moving average (EMA) are adopted by the technical analysts. Last but not least, with the advent of the digital computer, stock market prediction has moved in to the technological realm.

In the technological methods category, the most promising technique involves the use of artificial neural networks (ANNs). For stock prediction, the mostly used type of ANNs is the feed forward network with backward propagation algorithm in the weight training phase. In addition, another form of ANN that is especially suitable for stock prediction is the recurrent neural network (RNN) or time delay neural network (TDNN). In a feed forward ANN the output of each layer in the network depends solely on the current input. While in a RNN, the output is defined by both the current input and a hidden layer of states which are usually represented as a matrix. What makes RNN suitable for time serial prediction is that it combines two properties: (1) Distributed hidden state that allows them to store a lot of information about the past efficiently. (2) Nonlinear dynamics that allows them to update their hidden state in complicated ways. Examples of RNN are the Elman, Jordan, and Elman-Jordan networks. They are also known as "simple recurrent networks" (SRN).

The current state-of-the-art result in stock prediction using RNN is from Armando B.'s *Financial Market Time Series Prediction with Recurrent Neural* in 2012. They used a Echo State Networks (ESNs) which is also a branch of RNN to try to capture the inherent nature of stock data with feature vector included the current and 5-day history of the stock price, the 5, 10, 15, and 20 day moving averages, volume. As for data source, the S&P500 index is use. And the test error, measured by Root Mean Square Error (RMSE), for the ESN is 0.0027.

In this article, I'm going to explore RNN's, especially Long Short Term Memory Recurrent Neural Networks (LSTM-RNN), for its ability in predicting stock price, using no other possibly related factors but only historical prices.

In the technological methods category, there are other popular trends. Two most significant ones are the use of Text Mining together with Machine Learning algorithms and the use of new statistical analysis tools of complexity theory (researchers at the New England Complex Systems Institute (NECSI) performed research on predicting stock market crashes.) These trends will not be discussed in these article.

3 Implementation of LSTM-RNN

In this section, a Long Short Term Memory Recurrent Neural Network (LSTM-RNN) for historical S&P stock prices prediction is built, trained and tested with TensorFlow as back-end

platform using python.

3.1 THE VANISHING GRADIENT PROBLEM AND LSTM CELLS

In the phase of ANNs training using back propagation, there may raise a difficulty called the vanishing gradient problem. The cause of the problem lays in the nonlinearity of activation functions such as *sigmoid and tanh* . These activate functions have very small gradients for very large region of input. So, during the network training phase, this makes it hard for the error in the output layer to be back propagated to the earlier layers without getting too small. And this problem gets worse as the layer number increases. A common solution to this problem is to use ReLU instead of tanh or sigmoid as activation functions. Since the ReLU derivate is a constant of 0 or 1.

For RNNs, during the training phase, we adopt Back propagation Through Time (BPTT) algorithm to update weights. BPTT can be viewed as standard back-propagation on an RNN which is unrolled in time. Thus the vanishing gradient problem gets worse in RNNs structure. Which, in turns, results in the long term dependencies embedded in the input signal can not be properly learned. To solve this problem, instead of using the common ReLU activation function for ANNs, a even more popular and specifically suitable solution for RNNs is to use Long Short Term Memory (LSTM) architectures.

Long Short Term Memory networks, usually just called "LSTMs", are a special kind of RNN, capable of learning long-term dependencies in input signals. They were introduced by Hochreiter and Schmidhuber (1997). An illustration of a LSTM cell unrolled in time is shown in Figure 3.1.

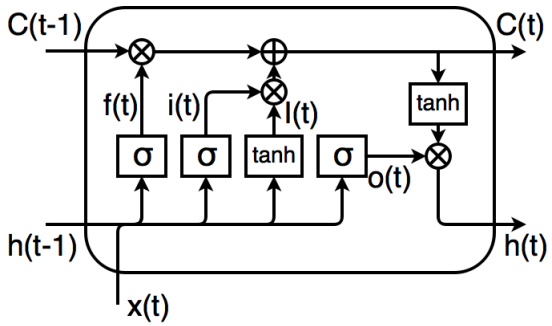

Figure 3.1: Illustration of a LSTM cell unrolled in time

In Figure 3.1, $x(t)$ is the cell's input at time t. And $h(t)$ is the cell's output at time t. $C(t-1)$ and $C(t)$ are the LSTM **Cell States** at time $(t-1)$ and t respectively, which are the key reason for LSTM to learn long term dependency without gradient vanishing risk. Cell states are controlled by three specifically designed structure called **Gates**. There are three gates: forget gate f(t) which conditionally decides what information to throw away from the block; input gate $i(t)$ which conditionally choses values from the input to update the memory state; output gate $o(t)$ which conditionally decides what to output based on input and the memory, i.e. cell state, of the block. Mathematically, the computation to update the LSTM cell states at time t(i.e.

$C(t)$) is shown in equation Equation 3.1; the computation to get the LSTM cell output at time t(i.e. $h(t)$) is shown in equation Equation 3.2.

$$\begin{aligned} f(t) &= \sigma(W_f * x(t) + U_f * h(t-1) + b_f) \\ i(t) &= \sigma(W_i * x(t) + U_i * h(t-1) + b_i) \\ I(t) &= \tanh(W_I * x(t) + U_I * h(t-1) + b_I) \\ C(t) &= f(t) * C(t-1) + i(t) * I(t) \end{aligned} \tag{3.1}$$

$$\begin{aligned} o(t) &= \sigma(W_o * x(t) + U_o * h(t-1) + V_o * C(t) + b_o) \\ h(t) &= o(t) * \tanh(C(t)) \end{aligned} \tag{3.2}$$

As for how the gradients-vanishing-exploding problem be solved and thus long term dependency be learned, there are two factors that affect the magnitude of gradients - the derivatives of weights and the activation functions are the two factors affects the magnitude of gradients. For example, the $tanh$ derivative is «1 for all inputs except 0; sigmoid is even worse and is always âL'd'0.25. On the one hand, LSTM's activation function is the identity function with a derivative of 1.0. So, the back-propagated gradient neither vanishes or explodes when passing through, but remains constant. One the other hand, its effective weight is equal to the forget gate activation. So, if the forget gate is on (activation close to 1.0), then the gradient does not vanish. Since the forget gate activation is never >1.0, the gradient can't explode either. So that's why LSTM is so good at learning long range dependencies.

3.2 BACK PROPAGATION THROUGH TIME

Our LSTM network is trained using **Back Propagation Through Time (BPTT)**. Although, by design, the output of a LSTM cell depends on arbitrarily distant inputs. Thus, the back propagation in time should be as long as possible, ideally infinity. Unfortunately, this makes back propagation computation difficult. In order to make the learning process tractable, it is common practice to use a **Truncated BPTT** which is a finite steps approximation of back propagation in time. So, truncated BPTT inherently has a parameter **Time Steps** which defines the truncated length. An example of truncated BPTT with three time steps on a input series of length six is shown in Figure 3.2. In this figure, four errors get back propagated a full three time steps; one error gets back propagated two time steps; and the first error only gets one time step.

In addition, TensorFlow uses a even more distorted implementation of truncated BPTT as is shown in Figure 3.3. Where, two errors get back propagated a full three time steps; two error gets back propagated two time steps; and two errors get only one time step. This may lead to a further inaccuracy to the results because the errors used to update the weights is significantly different from the ideally situation. The impact will be explored in the experiments later.

As is shown in Figure 3.3, the red thick arrow means, if the LSTM model is **stateful**, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch. Note that, our LSTM model is **stateless**. Which means the last state for each sample at index i in a batch will **NOT** be used as initial state for the sample of index i in the following batch.

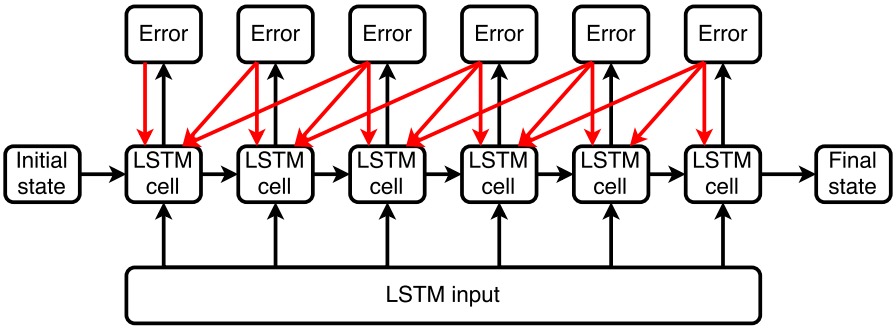

Figure 3.2: Illustration of three time steps truncated BPTT on a LSTM cell unrolled in time

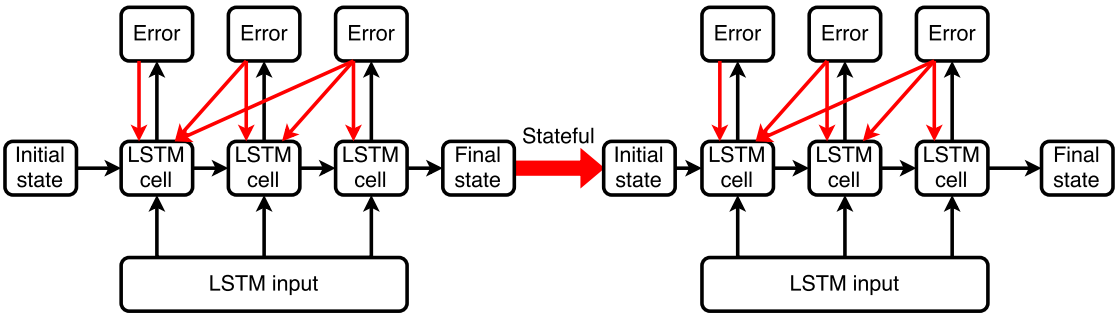

Figure 3.3: Illustration of TensorFlow's implementation of three time steps truncated BPTT on a LSTM cell unrolled in time

3.3 Data Preprocessing

The goal of data preprocessing step is to transform the original one dimension input data of time serial stock prices into an input matrix X that can be feed into LSTM network for batch processing. X should be of shape **Number Of Samples * Time Steps * Sample Dimension**. Here, the **Sample Dimension** is one since our input data is the one dimensional historical daily closing stock price of S&P 500. And **Time Steps** is the Truncated back propagation length introduced in section 3.2.

The data preprocess to get input matrix X is implemented as illustrated in Figure 3.4, where:

- Original input data $x(t)$ of length N and sample dimension 1.
- Sliding window is of length $1 + TimeSteps$. Where the last element in each window will be the $label$.
- Normalize window normalize each element d_i in a window using the first element in the window, specifically, $d_i = (d_i / d_0) - 1$.
- reshaping is on the purpose of forming the tensor accepted by tensor flow. Specifically, this step transform the input tensor of shape [n,m] to shape [n,m,1] where n is the train or test sample number and m is the Time steps.

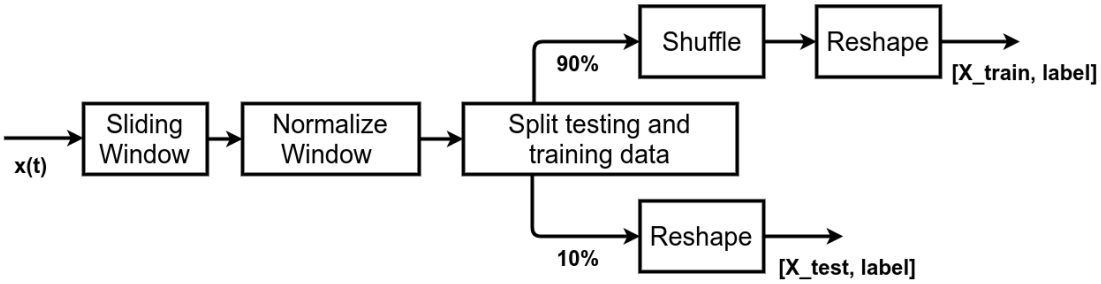

Figure 3.4: Illustration of the data preprocess before feeding them to the LSTM network

3.4 Structure of LSTM stock prediction network

As explained in section 3.2, the input matrix X of our LSTM network is of shape **Number Of Samples * Time Steps * Sample Dimension**. Then X gets slided into batches along its first dimension **Number Of Samples** and feed into the neural network. During the BPPT, the updating of parameters are based on batches. That is to say, for the samples in a batch, compute each output, **average the gradients and back propagate it trough time for each time step** to update the parameters matrix.

As is shown in Figure 3.5, the basic structure of our network is a concatenation of two LSTM cells followed by a regular Densely-Connected Neural Network layer to translate each multidimensional output of LSTM into a single prediction value.

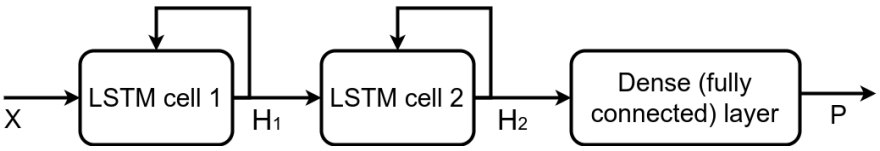

Figure 3.5: Illustration of a LSTM network structure

From Equation 3.2, we can easily get the matrix version of the LSTM cell's functionality during batch processing. Note that, in order to make full use of parallel computation. We remove the $V_o * C(t)$ part in Equation 3.2, which means the output gate $o(t)$ does not depend on cell state $C(t)$. Thus, take LSTM cell 1 for example, the cell functionality becomes Equation 3.3:

$$H_1 = \sigma(W_o * X + U_o * H_1') \tanh(C) \tag{3.3}$$

where, cell input X is of dimension **Batch Size * Time Steps * Input Dimension**. Cell output of current batch H_1 and of previous batch H_1' are both of dimension **Batch Size*Hidden State Size**. Cell state C is of dimension **Batch Size*Hidden State Size**. **Hidden State Size** is a configurable parameter of the LSTM cell that indicates its capacity of memorizing. **Hidden State Size** also decides the output dimension of the cell. In the basic implementation, it is set to be 50 for LSTM cell 1, and 100 for LSTM cell 2.

4 Experiments and Results

In this section, I conducted three sets of experiments to explore the model's ability of predicting stock prices, with each experiment focuses on one category of hyper parameters:

- Training parameters, i.e. epoch number and batch size.
- LSTM cell hyper parameters, i.e. time steps and cell state size.
- Network hyper parameters, i.e. layer numbers of stacked LSTM cells.

To measure how well our model performs in terms of predicting stock prices, we used **root mean square error (RMSE)**. However, a low RMSE is not enough, sometimes even misleading, on measuring performance. For example, the point by point prediction of a one layer LSTM cell (time step 50, cell hidden state size 60) after training of 70 epochs with batch size 512 is shown in Figure 4.1. Visually, it is pretty precise and the RMSE is 22.0185708455, which is pretty low considering the range of stock prices runs from 1850 to 2150. How ever, if we look closely, we will find that: this trained network is **echoing** instead of predicting. That is to say, the best strategy it found to predict the price at discrete time n is simply echoing the output at time $i = n - 1$, which means it didn't capture any hidden regular patterns embedded in the stock prices.

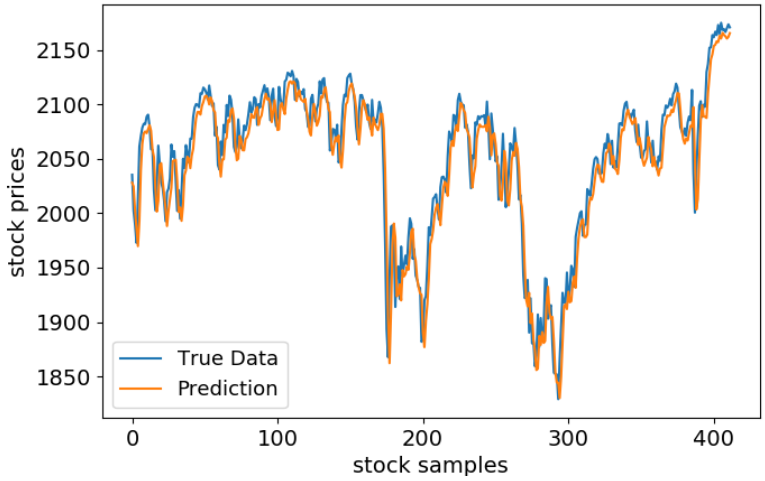

Figure 4.1: Point by point prediction of a one layer LSTM cell

So we need another indicator that can identify "echo" from "real prediction". For that purpose, I calculated the **Turning Point Prediction Accuracy (TPPA)** and **Turning Point prediction Confusion Matrix (TPPCM)**. Given the original stock prices series $x(n)$ and the prediction series $y(n)$, to get the TPPA and TPPCM, firstly, we need to classify each sample $a(i)$ in both $x(n)$ and $y(n)$ as one of the three class:

- **Summit** : $a(i-1) <= a(i)$ and $a(i) > a(i+1)$.

	predicted not a turning point	predicted summit	predicted valley
not a turning point	110	35	43
summit	74	0	37
valley	65	45	1

Table 4.1: Turning point prediction confusion matrix for a one layer LSTM cell

- **Valley** : $a(i-1) > a(i)$ and $a(i) <= a(i+1)$.
- **Not A Turning Point** : $a(i-1) < a(i)$ and $a(i) < a(i+1)$; or $a(i-1) > a(i)$ and $a(i) > a(i+1)$; or $a(i-1) = a(i)$ and $a(i) = a(i+1)$

Then, compare each sample's class in $x(n)$ and $y(n)$ accordingly. And get the TPPCM. To get the TPPA, simply add up alone diagonal in TPPCM and divided by the total sum of each element in TPPCM. And the TPPA for the prediction in Figure 4.1 is 0.27 which is pretty low. And the TPPCM is shown in Table 4.1. Where, none of the summit in $x(n)$ are successfully predicted and only one of the valley in $x(n)$ is correctly predicted. **So a low TPPA, around 0.3, indicates the network is rather echoing instead of doing meaningful predictions.**

The data source for all the experiments following in this section is from the S$P 500 stock's daily closing price, which are 4121 samples in total. I set the first 90 percent as training data and the rest are for testing, resulting in 3709 training samples and 412 testing samples.

4.1 Experiment 1: batch size and number of epoch in training phase

Since, during BPPT, the updating of network parameters are based on the average error of a batch of data. So the batch size may have an influence on the result. In this experiment, we first explored whether batch size is relevant. If not, it would be very beneficial to reduce the amount of experiments needed for other network hyper-parameters.

Especially, I chose two models: a one layer LSTM cell of state size 50, followed by a dense layer; two layers stacked LSTM cells which are both of state size 50, followed by a dense layer. The time steps during truncated BPTT are set as 50 and training process are run 50 epochs. Then I tested the RMSE and TPPA for different batch sizes. And the result is shown in Table 4.2.

From Table 4.2, it is obvious that for both the tested models, **the increase of batch size doesn't decrease the TPPA. And the increase of batch size doesn't increase RMSE**. So, considering that the increase of batch size linearly decreases the execution time, we will choose the largest possible batch size my computer hardware can handle, which is 512 in all the following experiments.

Besides batch size, there is another hyper-parameter during training phase - number of epochs, which means the iteration times of training. Not enough epochs may results in the model not sufficiently trained. While too much epochs will waste time without any result improvement. What's worse, it may lead to over-fitting problem. Roughly speaking, over-fitting typically occurs when the ratio of "complexity of the model " over "training set size" is too high. So in this experiment, we measured the perform of four models with different amount of epochs, where the four models are:

batch size	RMSE	TPPA	batch size	RMSE	TPPA
32	19.67	0.21	32	19.67	0.21
64	20.19	0.22	64	20.25	0.22
128	19.75	0.21	128	19.75	0.22
256	19.80	0.22	256	20.35	0.23
512	20.66	0.26	512	22.15	0.25

Table 4.2: Prediction performance of a one layer LSTM cell of state size 50 (left) and two layers stacked LSTM cells both of state size 50 (right) with different batch size

- **Model 1:** a one layer LSTM cell of cell state size 100, followed by a dense layer; and the time step during BPTT is 10;
- **Model 2:** a one layer LSTM cell of cell state size 100, followed by a dense layer; and the time step during BPTT is 50;
- **Model 3:** two-layer-stacked LSTM cells of cell state size 50 and 100, followed by a dense layer; and the time step during BPTT is 10;
- **Model 4:** two-layer-stacked LSTM cells of cell state size 50 and 100, followed by a dense layer; and the time step during BPTT is 50;

And the result is shown in Figure 4.2.

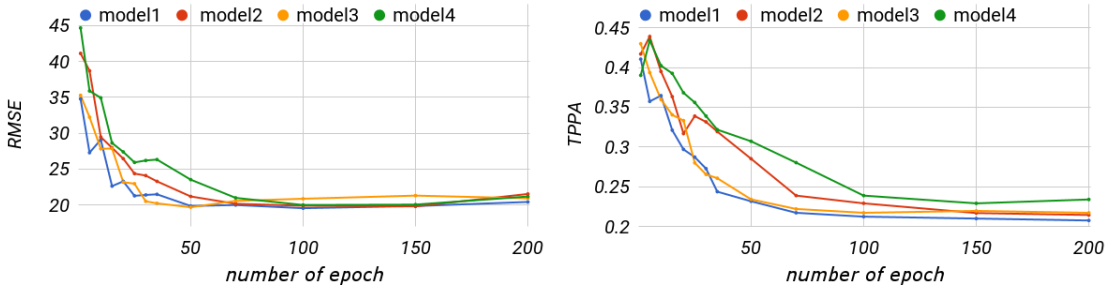

Figure 4.2: Performance of four models with different training epochs

From Figure 4.2, there are two observations:

- **Observation 1:** as the epoch number increases, TPPA decreases all the way down to around 0.2. In addition, the final RMSE for each of the four models are all 20. This means all the tested four models doesn't pick up any hidden rule about the input data. Instead, **they are trying to echo back the input of the current time as the prediction of the price of next time step.**
- **Observation 2:** in terms of RMSE, compared with the two models of time step 50, the two models with time steps of 10 get stable faster. This means, a more complex model needs more epochs to get trained. But from this result, we find out that **around 70 epochs are enough**.

4.2 Experiment 2: time steps and hidden state size of LSTM cell

From the previous experiment we inferred that batch size doesn't infect the result and 70 epochs are enough. So, in this experiment, we will further experiment on the prediction ability of a single LSTM cell in relation with its two hyper-parameters - hidden state size (i.e. cell state) and Time Steps which is introduced in section 3.2. And the result is shown in Figure 4.3 and Figure 4.4.

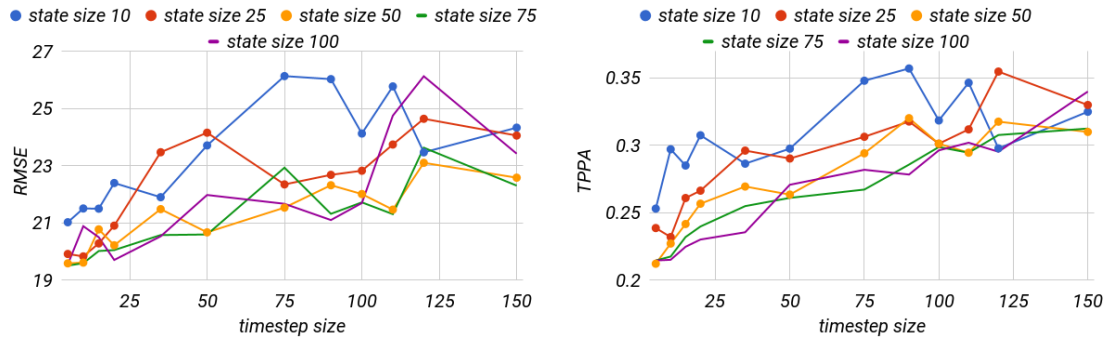

Figure 4.3: Performance of single layer LSTM cell with different time step size

- **Observation 1:** From Figure 4.3, we can infer that for all the tested state size, as time step increases the RMSE increases and TPPA increases. Which means **with larger time step, the model tends to not simply echo the previous input as the prediction of next time step**. Even though, with one layer of LSTM cell, their ability of prediction is still limited, since all the TPPA is under 0.35.
- **Observation 2:** Models with state size as 10 and 25 both had a "breakdown" in terms of RMSE when time step is around 75 and 50 separately. Which shows **for small state size model, relatively large time step will do harm to its results. But when time step size goes even larger, bigger than 100, this effect will somehow be counteracted**.

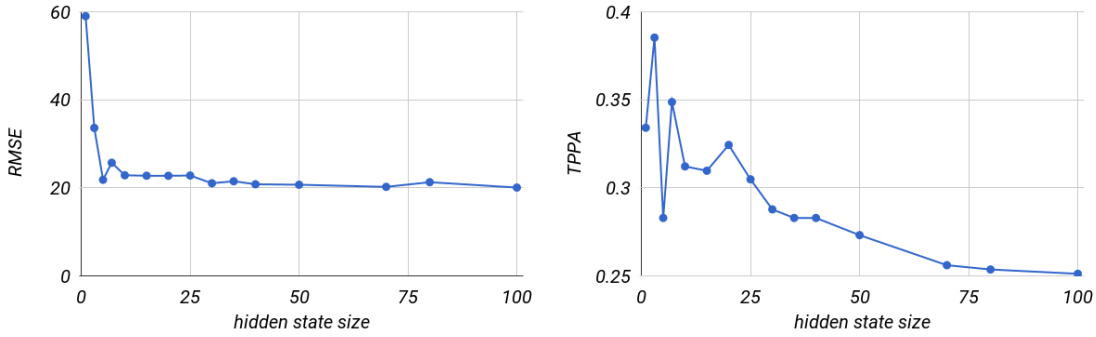

Figure 4.4: Performance of single layer LSTM cell with different hidden cell state size

- **Observation 3:** From Figure 4.4, it is obvious that **the prediction error RMSE goes down dramatically as state number of the LSTM cell increases from 1 to 5.** This is because when state number increases, the ability of LSTM to remember long term memory increases. Thus more hidden nature of the input data is captured.

- **Observation 4:** From Figure 4.4, after state size 10, the RMSE gets stable and the TPPA goes down as state number of the LSTM cell increases. This means, what the model picked up is "echoing" instead of hidden patterns embedded in the input.

4.3 Experiment 3: depth of the LSTM network

From previous experiment, we found out some information and underlying properties about LSTM cell's behaviors. But all the one or two layers of LSTM models tested can not really start "predicting" instead of "echoing". So in this experiments, we add more layers to the model and test whether depth can help it get out the "echoing". In addition, for each model, we tested the result of: (1) point by point prediction, which means only predict the price for the next day (2)multiple prediction, which means predict the price for the next n days. Where I chose n as 20. But constrained by the computer hardware, I can not go too deep. The final candidates LSTM networks are:

- **Model 1:** a one layer LSTM cell of cell state size 60, followed by a dense layer; and the time step during BPTT is 50;

- **Model 2:** two-layer-stacked LSTM cells both of cell state size 30, followed by a dense layer; and the time step during BPTT is 50;

- **Model 3:** three-layer-stacked LSTM cells all of cell state size 20, followed by a dense layer; and the time step during BPTT is 50;

And the result is shown in Figure 4.5.

- **Observation 1:** From Figure 4.5, it is shown that as the layers increase (from top left to the bottom left), the point by point prediction result doesn't get better. But **the result for tendency prediction does improved, comparing the multiple prediction result of the one layer model(top right) and the two layers model(medium right)**. Specifically, for the drastic drop in stock price around stock sample 170, the one layer model predicts the future trend as continuing dropping, which is very wrong. While the two layer model correctly predicts the future "bouncing back" pattern. In addition, the three layer model even predicts the trend better.

5 Future Work

This article is a beginning of an interesting journey. With the current result, there are even more questions left open:

- Is a more complex LSTM RNN with more layers can achieve better result?

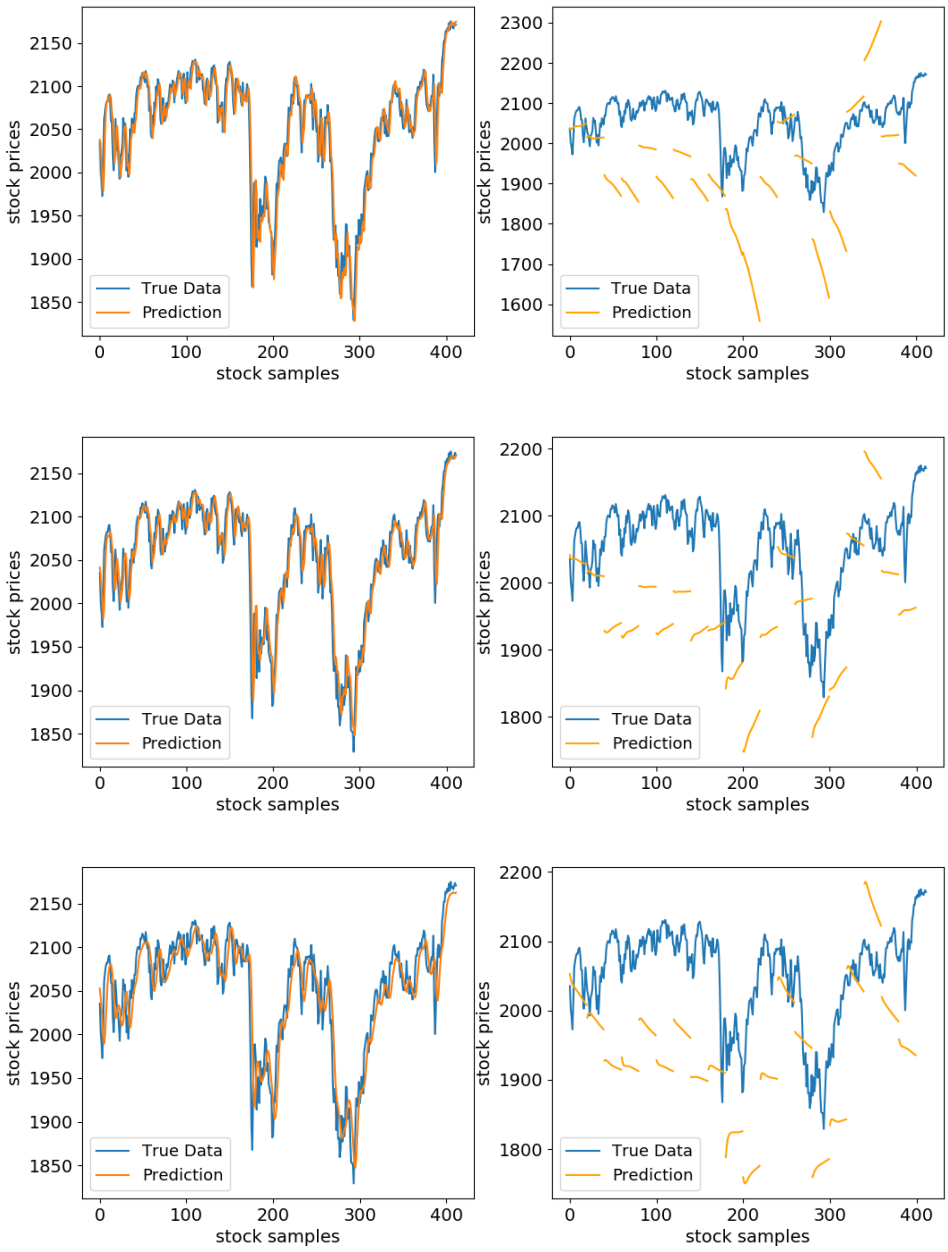

Figure 4.5: Point-by-point prediction(left) and multiple prediction(right) of 3 models of stacked LSTM network (model 1 : top; model 2 : middle; model 3 : bottom).

- How about the generosity of the trained LSTM model? Is the accuracy going to drop down dramatically when train the net with stock TSLA and tested it with stock AAPL? Further more, is training with multiple Stock Symbols will produce an unified LSTM that has good result on all kinds of data?

- What is the relationship between training length and prediction accuracy? This will determine how quick this RNN can adapt to new data, which is a critical feature in real time environment.

REFERENCES

[1] Hochreiter, S., Schmidhuber, J. *Long short-term memory. Neural computation.* 1997. `http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf`

[2] Torkil, A. *Predicting Stock Markets with Neural Networks A Comparative Study* Spring 2015.

[3] Xiao D., Yue Z., Ting L., Junwen D. *Deep Learning for Event-Driven Stock Prediction* 2015.

[4] Armando B., Sam F., Rohit P. *Financial Market Time Series Prediction with Recurrent Neural Networks* December 14, 2012

[5] J. G. Agrawal1 , Dr. V. S. Chourasia2 ,Dr. A. K. Mittra3 *State-of-the-Art in Stock Prediction Techniques* 2013