Automated Stock Market Trading System using Reinforcement Learning, Neural Networks, and Naive Bayes

Xinwen Zhang, Ben Struhl, Jian Cui
Northeastern University, Boston, MA

Abstract

Since the birth of the stock market, investors have been constantly looking for ways to improve their trading strategies to maximize profits with minimal labor cost. With new advancements in computational capability, and decreased hardware cost, investors can now utilize a variety of Artificial Intelligence techniques to automate and improve trading systems. However, choosing the proper techniques could be challenging.

In this paper, we are interested in evaluating deep Q learning, Bayes networks, and Long-Short Term Memory networks by using each as the basis of a trading policy. We then evaluate each policy by comparing it to a generic buy and hold strategy in terms of profitability.

Introduction

In recent years, new innovations in machine learning techniques, and the increased availability of large stock datasets has made automated trading systems more prolific in the financial world. The ability to execute trading strategies automatically with minimal human intervention tremendously reduces labor cost and human errors.

In real world trading environment, human traders tend to be influenced by their emotions and internal biases, causing them to make subpar trades. In an automated system, emotions are eliminated, and biases are minimized due to these systems inherently trading on patterns within objective data.

To better understand how an automated trading system works, it is critical to know what trading strategies are and how machines could use them to make trading decisions. Stock trading is the act of buying and selling stocks based on predicted short-term or long-term stock price movements. The mentality associated with the most basic strategy is to buy low and sell high. The overall goal is to maximize profits by initiating trades based on the traders generated policy.

In this paper, we create trading policies for our dataset using three different AI algorithms: DQN, Naive Bayes, and LSTM.

For our problem, the trading policy must make a decision based on time series stock data. One way you could tackle it is by using an LSTM network. An LSTM network is like a recurrent neural network except it is designed so that it can remember important pieces of data from previous inputs to use as features for the new inputs. This ability to hold onto relevant context information is important for stock price predictions that rely on previous stock price trends.

Another approach is to use DQN. Traditional Q-learning evaluates the next possible states and greedily choose the state with highest approximation. The algorithm keeps repeating and adjusting the Q values until the policy converges. Deep Q network uses the same idea of evaluating states with the enhancement that it utilizes neural network to process the trading environment with huge number amount of states.

The final approach is Naive Bayes. Given some features, the Naive Bayes can classify the training data and create a distribution chart. If we are only interested in seeing if a stock will go up or down in some period, we can use the distribution chart to infer probabilities and making predictions. In the Bayesian approach, we get a complete posterior distribution for each feature, which quantifies how likely an event is. The approximation becomes more accurate as we provide more features for it to utilize.

In the real-world trading environment, many factors could cause the stock prices to behave unpredictably, such as news, events, and company financial reports. In order to reduce the factors in the environment, we use a pure technical analysis approach which only considers the historical closed price changes as a reference. We model the domain of the input as 2-dimension charts with x being time and y being price traded.

DQN Background [Xinwen Zhang]

Reinforcement Learning refers to a kind of Machine Learning method which the agent interacts with a well-defined environment and receives feedback in the form of rewards. The agent learns the environment on the fly by trying out different actions and generates a policy to remember the learned experience. The goal of reinforcement learning is to generate a policy such that it can choose the best action for each state leading to global maximum reward. Reinforcement Learning is mostly used in games with performance exceeding human agents.

One type of reinforcement learning agent is Q-learning. Q-learning learns the utility value of an action from a given state. The utility value is based on the rewards received by the agent performing valid actions given a state. Assuming there exists an optimal policy, the agent should be able to approximate or find it.

Q-learning typically maintains a tabular table that stores the current Q-values for every possible state-action pair. This is necessary because Q-values are updated after each action based on reward and estimate of optimal future reward. The limitation of Q-learning is that the total number of possible states in the environment could be huge and it might be impossible to hold all states in memory.

One way to solve this problem is to use deep Q-Network algorithm. Deep Q-Network utilizes a neural network to approximate *Q (s, a)*. With this approach, we generalize the approximation of Q-value function rather than explicitly remembering the Q values.

$$Q(s,a) = r(s,a) + \gamma Q(s', argmax_a Q(s',a))$$

TD target

DQN Network choose action for next state

Target network calculates the Q value of taking that action at that state

[2]

In the formula, *r (s, a)* represents reward function given state s and action a. γ is the discount factor that determines the impact of future return. *Argmax (s', a)* is the approximated best action chosen by the neural network.

Neural networks are a set of algorithms designed to recognize patterns and accumulate experience over time. It consists of an input layer, many hidden layers and an output layer. Each layer is connected by weights and biases to represent how important a pattern is in the network. It learns from the sample data and predicts the output based on an unseen input. When initializing a neural network, we have to provide it with a learning rate, size and number of layers. Learning rate α is how quickly a network abandons old beliefs for new ones.

In the training process, the neural network updates its internal weights by reducing the error. The error is calculated by taking the difference between our *Q_target* (maximum possible value from the next state) and *Q_value* (our current prediction).

$$\Delta w = \alpha[(R + \gamma\, max_a\, \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)]\, \nabla_w \hat{Q}(s, a, w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

Gradient of our current predicted Q-value

[2]

One of the challenges is that neural network tends to forget the previous experiences as it overwrites them with new experiences. So, we need a list of previous experiences and observations to re-train the model with the previous experiences.

When the agent starts acting in the environment, there is one more variable that needs to be introduced: exploration rate or epsilon. This variable defines how frequently an agent takes a random action. This helps the agent choose between exploration and exploitation.

Related Work [Xinwen Zhang]

Instead of Deep Q-learning, many people have attempted using different strategies or algorithms to make trading decisions.

Most commonly used strategy (policy) among many individual investors is buy-and-hold strategy. This strategy is a passive investment strategy for which an investor buys stocks and holds them for a long period regardless of fluctuations in the market. The buy-and-hold strategy employs a mentality that suggests price movements over the long term will outweigh the price movements in the short term, as such, short-term movements should be ignored [3]. It is worth mentioning because this strategy worked really well from 60s to 90s when stock market was so predictable in the long term. In this paper, we are going to use this strategy as our baseline because we know that Google's stock price will most likely go up in the long run.

Another approach would be using Bayes net. Bayes networks are a type of probabilistic graphical model that uses Bayesian inference for probability computations. The most difficult part about Bayes is to identify variables and determine edges between variable in order to build a graph. Once the graph is built, it cannot be changed. The performance of the Bayesian net is totally dependent on the nodes and edges. If the graph is built with wrong edges or variables, it would be impossible to get an accurate inference since the variables or edges may have nothing to do with each other. Another

downfall would be the nature of Bayes net. It is intended for making predictions, not generating policies. Even if the prediction has 100% accuracy, it does not guarantee an optimal solution. Generating a policy from predictions leads to another dynamic programming problem. Deep Q-learning, on the other hand, does not have this limitation. Deep Q learning generates a policy directly from training data and the algorithm only deals with states, rewards and actions which makes it more flexible and requires less configurations.

Project Description [Xinwen Zhang]

As mentioned in the previous section, the goal of the system is to maximize overall profit. Building the policy using DQN involves training the agent and evaluating the agent's performance.

Google's daily closed price from 2014-03-27 to 2015-12-31 is used as training data and the data from 2016-01-04 to 2017-11-10 is used as the testing data.

A proper environment is needed in order for the agent to learn. One of the interests in this research focuses on trying out different configurations in the environment to see if the agent can gain better performance. The environment should have a reward function, and a sequence of states over time. We define the states in such way that an agent can only see the historical records and make one trading action at the current state. As time moves forward, the agent sees more historical data. The reward function is another tricky part in the system. I initially let realized profit be the reward because it is human intuition. Realized profit is profit that comes from a completed trade which consists of a stock being bought and sold.

Finally, the DQN agent is used to interact with the trading environment. In the real world, traders have many ways to trade, such as short selling, placing stop loss orders and etc. To simpfy our model and make training faster, we only allow three actions: buy, hold or sell. Short selling is also prohibited. If the agent initates sell action without having any shares, nothing will happen. It will be treated as holding. We also define the rule that when the agent buys, he can only buy 1 share and when the agent sells, it will sell all share at the current price. Additionally, we assume there is no total cap on the trader's assets, because having this variable adds complexity to the nural network and make training slower to converge. The pseudocode for the training is provided in Figure 1. The agent starts off playing randomly and remembers the states, actions and rewards. The neural network receives the randomized replay samples and then updates itself to reduce error caculated by gradient descent forumula.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

[4]

Figure 1

Experiment [Xinwen Zhang]

All the configurations mentioned in the project description section are implemented in python 2.7 with Keras, a high-level neural networks API. All experiments are performed on a 64-bit Ubuntu 18.04. The implementation consists of training and evaluation programs. The training program takes some parameters and outputs a trained model. The evaluation program outputs the trading actions using the model and generates a statistically report.

Before I run any experiment, I first calculate the base line using buy-and-hold strategy. On 2016-01-04, Google has a closed price of $741.84 and $1028.07 on 2017-11-10. The profit is 1028.07-741.84 = $286.23.

My first set of experiments varies the reward function. I use a gamma value of 0.1 to diminish the future reward and make the training greedier. I choose realized profit as the reward function for my first experiment because it is human intuition that the agent gets rewarded more if it makes more profit. The agent only gets rewards when it has shares on hand and decides to sell. The reward is calculated as:

```
Total-Profit = Total-Profit + ((current-share-price * len(list-of-bought-share-prices) - sum(list-of-bought-share-prices))
```

I run my tests with the episode size of 100 and 150 as a comparison. See Figure 2 for reward changes. Series 1 is trained

with 150 episodes and series 2 is trained with 100 episodes. Series 1 has steeper ups and downs because the agent explores more. Both charts fluctuate a lot because reward function is not normalized. Because the reward is not normalized, it is very easy to get overfitted policies. The overfitted model would generate all buy or all sell actions. With numerous attempts, I am able to complete the training with a converged result. The result is shown in Figure 2. Both models converge at around $800. After running it against the testing data set, the model with 100 episodes generates $711.71 profit and the model with 150 episodes generates $808.68 profit. I have also tried training the agent with a greater number of episodes, but the reward changes were very unstable and couldn't converge. My assumption is that because reward is not normalized the agent gets lost by the overfitted values in the exploration process and could not learn anything.

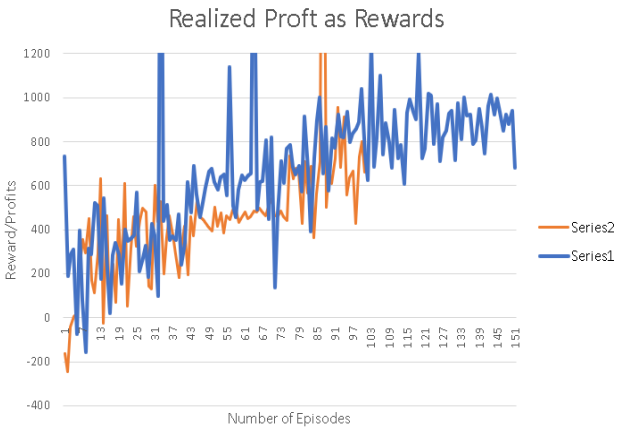


Figure 2

After looking at the trades generated by this policy, I notice that the agent makes many trades and each trade makes a small profit. Based on this observation, I can assume that it might be possible to improve the profits by maximizing the number of winning trades and minimizing the number of losing trades. So, I start my second experiment and let reward function be +1 if a trade is profitable and -1 when a trade is a losing trade. The intuition is to maximize the number of winning trades and minimize the number of losing trades.

I run my tests with the episode size of 100 and 150 and the reward converges at around 70. See Figure 3 for the training result. I then run my evaluation function against the testing data. The evaluation shows that the model with 100 episodes makes a profit of $565.10 and total reward of 78. The total reward is the total number of winning trades minus the total number of losing trades. The model with 150 episodes makes a profit of $636.55 and a total reward of 82.

In comparison to the first reward function, using net number of winning trades as reward does not improve the overall profit. The table in Figure 4 shows the statistics between the two reward functions. The second reward function can indeed increase the number of winning trades and decrease the number of losing trades. However, it does not improve the realized profit.

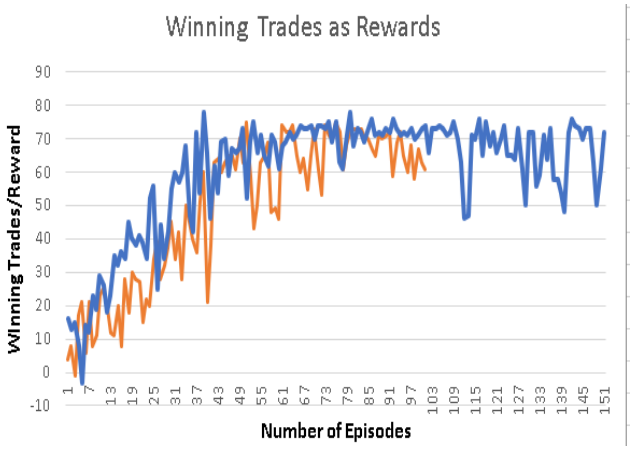


Figure 3

RP: Realized profit
W/L: Winning/Losing

	Total Profits	Net # of Winning Trades	# of Winning Trades	# of Losing Trades
RP as Reward 100	$711.71	76	97	21
RP as Reward 150	$808.68	76	96	20
W/L Count as Reward 100	$565.10	78	100	22
W/L Count as Reward 150	$636.55	82	103	21

Figure 4

The next set of experiments varies the gamma value, the discount factor. When testing gamma, I let the reward be the realized profit since it has a better performance. The goal is to test if gamma value could improve the realized profit.

I test gamma with three values 0.5, 0.8 and 1. I run 150 episodes for each. In the training process, the training with gamma value of 0.5 and 0.8 are able to converge. However, the training fails every time when gamma is set to 1. My guess is that since the agent's actions do not affect trading states it is impossible or difficult for the agent to estimate long term rewards. So, the agent must act at the current state to get the most rewards. Also, the training with 0.8 gamma value produces a more stable training. It is because the agent chooses a less greedy action at the current state hoping to get more rewards in the future. Figure 5, 6 shows the rewards vs episodes charts for gamma value of 0.5 and 0.8 and Figure 7 shows the evaluation report. Both gamma values, however, does not improve the overall profit.

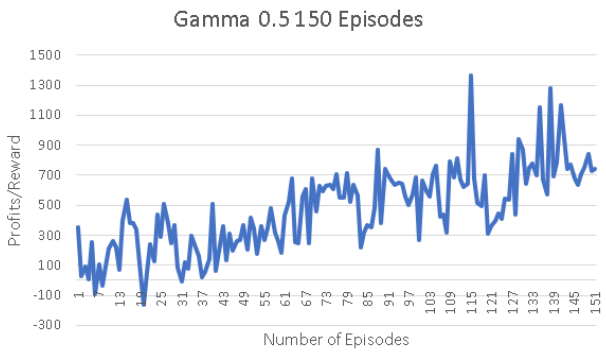


Figure 5

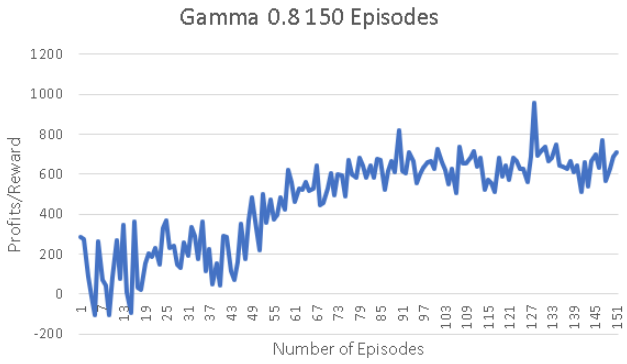


Figure 6

	Total Profits	Net # of Winning Trades	# of Winning Trades	# of Losing Trades
Gamma 0.5	$640.32	72	92	20
Gamma 0.8	$733.97	73	90	17

Figure 7

I have also tried using other reward functions, neural network configurations and different combinations of those. In the end, I am able to generate a model that can make $1500 profits using the following configuration:

- Neural network: 3 hidden layers and a learning rate of 0.005
- Reward function: realized profit
- Gamma: 0.5
- Number of episodes: 200
- Epsilon decay: 0.995

See Figure 8 the profit chart to see the comparisons of all training policies.

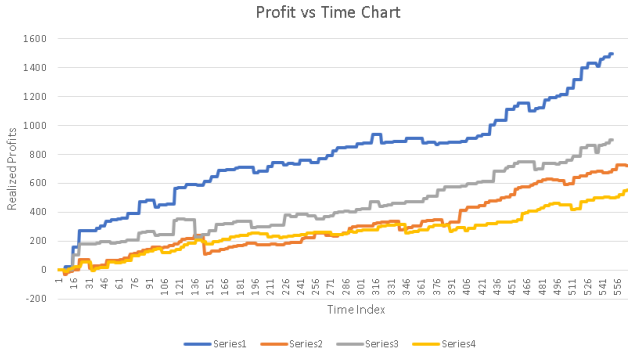


Figure 8

Conclusion [Xinwen Zhang]

The results presented above shows that it is possible to use machine learning approaches to generate positive profits. With Deep Q-Network, the agent is able to make $1500+ profits with certain configurations. However, this result might not be realistic in the real world.

In this research, the trading environment and agent actions are over simplified to reduce hidden variables, such as total asset, number of shares on a single trade and transection fees etc. The real-world trading environment has more hidden variables and it may take longer or impossible to train. On the other hand, without concerning these variables, the agent is able to learn quicker and show promising results.

Using this simplified environment also allows us to analyze how training configuration affect the profits. The result of using different reward function shows that maximizing the number of winning trades and minimizing the number of losing trades does not improve the performance. It also indicates that it is easier to get overfitted models using realized profit as a reward function since it is not normalized. Additionally, using different gamma values does not improve the profits, but it could make the training process more stable. Gamma value of 1 always leads to the agent not able to learning anything.

Bayes Background [Jian Cui]

The naive Bayesian classification algorithm is based on Bayes' theorem, and it is a statistical classification method used to calculate the probability of a input belonging to a category. A comparative study of classification algorithms shows that the classification performance of the naive Bayesian classification algorithm is comparable to that of decision trees and artificial neural network classification algorithms.

The basic principle of the naive Bayesian classification algorithm is as follows

(1) Each data sample uses a Van dimension vector = {x_1, x_2 x_3, …, x_n}, they are respectively for attributes of the sample {A_1, A_2, ⋯, An}

(2) Assume that there is *m* category C_1, C_2, …, C_m. Given a data sample X of an unknown category and known attributes. The classifier predict X belongs to the category with the largest a posteriori probability. In other word, classify the sample into category C_i, if only if : $P(C_i \mid X) > P(C_j \mid X),\ 1 \le j \le m,\ i \ne j$. According to Bayes' theorem:

$$P(C_i \mid X) = \frac{P(X \mid C_i)P(C_i)}{P(X)} \quad [1]$$

(3) Base on Bayes' theorem

$$P(X) = \sum_{i=1}^{m} P(X \mid C_i)P(C_i) \quad [1]$$

P(X) is constant for all categories, so we only need to find the largest $P(X|C_i)P(C_i)$. $P(C_i)$ is called the prior probability of the class, calculated by the formula $P(C_i)=S_i/S$, where S_i is the number of samples in the training sample with category Ci, and S is the total amount of samples in the training sample set

(4) Given a data set with multiple attributes, the cost of directly computing $P(X|C_i)$ can be very large. To reduce the computational cost, the Bayesian classification algorithm usually assumes that all attributes happens independent, so for a category:

$$P(X \mid C_i) = \prod_{k=1}^{n} P(x_k \mid C_i) \quad [1]$$

a) If attribute A_k is a non-contiguous quantity, there is $P(x_k \mid C_i) = S_{ik} / S_i$, where S_{ik} is the number of samples in the training sample set with category C_i and attribute A_k, and S_i is the sample in the training sample set with category C_i.

b) If attribute A_k is a continuous quantity, then we usually assume that the attribute has a Gaussian distribution:

$$P(x_k \mid C_i) = \frac{1}{\sqrt{2\pi}\sigma_{C_i}} e^{\frac{(x-\mu_{C_i})^2}{2\sigma_{C_i}^2}} \quad [1]$$

(5) We can predict the sample X belongs to category C_i, if only if : $P(C_i \mid X) > P(C_j \mid X),\ 1 \le j \le m,\ i \ne j$

An n-gram model is a type of probabilistic language model for predicting the next item in such a sequence in the form of a (n − 1)–order Markov model. N-gram models are now widely used in probability.

For example: we have a string: ⋯to_be_or_not_to_be…

1-gram sequence: ⋯, t, o, _, b, e, _, o, r, _, n, o, t, _, t, o, _, b, e, …

2-gram sequence: ⋯, to, o_, _b, be, e_, _o, or, r_, _n, no, ot, t_, _t, to, o_, _b, be, …

3-gram sequence: ⋯, to_, o_b, _be, be_, e_o, _or, or_, r_n, _no, not, ot_, t_t, _to, to_, o_b, _be, …

Related Work [Jian Cui]

Instead of Naïve Bayes, many people have attempted using different methods to predict stock trend and make trading decisions. The most popular algorithm is Deep-Q learning. In the Q-Learning Algorithm, there is a function called Q Function, which is used to approximate the reward based on a state. We call it R = Q(s,a) + discount * Q'(s,a), where Q is a function which calculates the expected future value from the state s and action a, Q' calculates the future reward after doing action a and the discount is a constant between 0 and 1. The discount tells how much weight you give the future reward. Choosing a proper gamma is difficult due to the fact that it is subjectively set based on the particular task environment. Thus due to the difficulty of choosing a particular discount value, I chose to use Naïve Bayes over Deep-Q learning.

Project description [Jian Cui]

For Naive Bayes

For the Bayesian model, the most important part is the selection of attributes. First, I choose six attributes, which are the opening price, the closing price, the highest price, the lowest price, the difference between the opening price and the closing price, and the slope of the difference. I choose these attributes due to my assumption that most traders would look at these indicators before making a judgement about a stock's future price. I also use the N-gram algorithm to filter the sample. Below I demonstrate how I calculate all the above-mentioned attributes for my model.

Figure 1, the example of data

Date	Open	High	Low	Close
2014/4/1	555.6547	565.3414	555.6547	564.0585
2014/4/2	596.709	601.5225	559.1157	563.8994
2014/4/3	566.7338	584.0685	561.045	566.6244
2014/4/4	571.5076	574.6105	540.0306	540.1699
2014/4/7	537.783	545.4807	524.2673	535.2072
2014/4/8	539.6328	551.965	538.6482	551.8655
2014/4/9	556.5598	562.2783	549.9262	561.055

(1) Get the stock data for N days:

For the example above: we classify them by data. Therefore, we can get data set D ={d_1, d_2, …, d_7} where d_1 = {555.6547, 565.3414, 555.6547, 564.0585, -8.4038, -0.01512}, d_2 = {555.6547, 565.3414,

555.6547, 564.0585, -8.4038, -0.01512}, …, d_7= {556.5598, 562.2783, 549.9262, 561.055, -4.4952,0.008077}

(2) Use N-gram to rebuild the data set
For the example above, assume n = 3, we can fix the data set to $D' = \{ d_1', d_2', ..., d_7' \}$ where $d_1' = [d_1, d_2, d_3]$, $d_2' = [d_2, d_3, d_4]$, ..., $d_5' = [d_5, d_6, d_7]$

(3) Then generate the sample with the attribute S={s_1, s_2, s_3, s_4, s_5}. Cause we want to predict the label for the third day, so we need assume we don't have the information of the third day, then the s_i is:
a) s_i*_opening_price =* d_{i+1}*_opening_price -* d_i*_opening_price*
b) s_i*_closing_price =* d_{i+1}*_opening_price -* d_i*_opening_price*
c) s_i*_highest_price =* d_{i+1}*_highest _price -* d_i*_ highest _price*
d) s_i*_lowest_price =* d_{i+1}*_ lowest _price -* d_i*_ lowest _price*
e) s_i*_diff_price =* d_{i+1}*_ closing_price -* d_i*_ opening_price*
f) s_i*_diff_slope =* d_{i+1}*_diff_slope -* d_i*_ diff_slope*.

(4) Get the label set L = {l_1, l_2, l_3, l_4, l_5}, if the closing price of the third is higher than the opening price of the first day , So: l_i = d_{i+2}_ closing_price - d_i_ opening_price

(5) Because the data is a continuous quantity, we need use Gaussian distribution then train the model. Both formula we talked above.

(6) Make the trade policy base on the predict result.
Naïve Bayes policy(L'(predicted Label):
a) profit = 0, sumOfCost = 0, amountOfStock = 0
b) For l_i *in L:*
if the l_i *is 1:*
sumOfCost = sumOfCost + d_i*_open_price*
amountOfStock = amountOfStock +1
if the l_i *is 0 and amountOfStock > 0:*
*profit = profit + amountOfStock ** d_{i-1}*_close_price – sumOfCost*
sumOfCost = 0, amountOfStock = 0
c) output profit

Experiment [Jian Cui]

My experiments are based on a few different features. First, I choose 6 features from the sample, like I mentioned above. So, I formulate the experiment according to the following:
1） only the opening price
2） only the highest price
3） only the lowest price
4） only the closing price
5） only the price difference
6） only the difference of slope
7） opening price and the highest price
8） opening price, highest price and lowest price
9） opening price, highest price, lowest price and closing price
10） opening price, highest price, lowest price, closing price, and the price difference
11） opening price, highest price, lowest price, closing price, price difference and the difference of slope.

The result of prediction accuracy is shown in figure 2, which is the calculating using the variables L_c/L_s. L_c is the number of predictions which are correct and L_s is the size of the real label set. From figure 2, for the models that only use a single feature, and the model that uses the closing price get 0.787234043 accuracy, which is the highest accuracy of the single feature. The lowest accuracy is 0.670212766 whose corresponding feature is the price difference. We also can find the accuracy is increasing as the number of features increase. The accuracy for 4 features, 5 features, 6 features is 0.840425532, 0.840425532, 0.842553191. So, when we evaluate the sample with the price difference feature, the accuracy does not decrease and when we add the difference of slope feature, the accuracy increases. Which means the more features we have, the better the accuracy we will get.

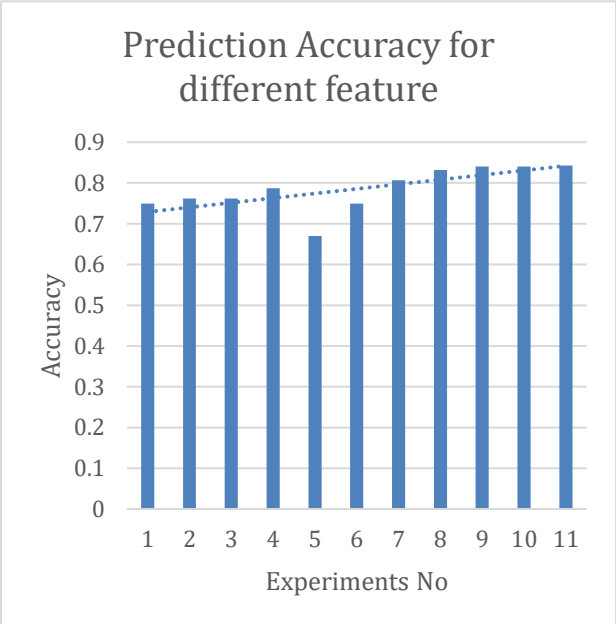
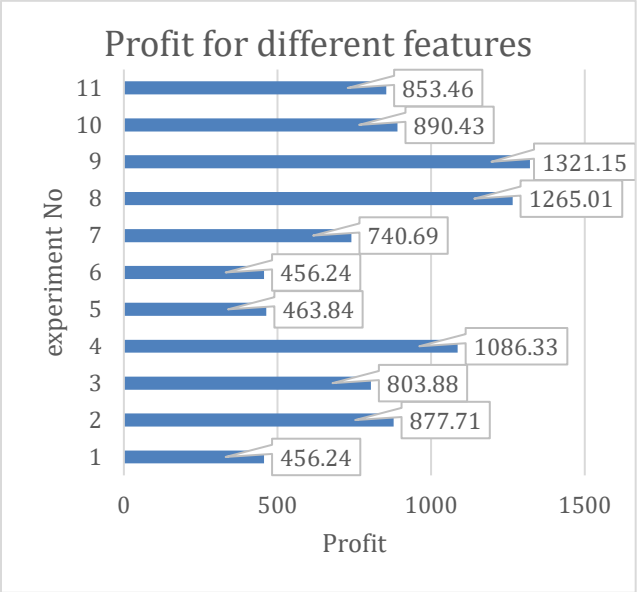


Figure 2, prediction for different feature

The next step is generating a trading policy with the prediction result. When the label is 1, we take the action to buy one share stock and sell all stock when the label is 0. The buy price is the open price for that day. And the selling price

is the close price of the day before that day, which means we sell all the stock before the price goes down.

Figure 3 Profit of different features



The profit comparison is shown in figure 3. We can find that if we only consider the closing-price feature, the profit is better than some feature combinations. The highest profit we make is experiment No.9, which is $1321.15. Then for the experiment No.10 and No.11, although the accuracy is increased, the profit is less than experiment No.9. Both experiments generate around $800. Their profit is also less than the experiment No.4. This means the accuracy may influence the profit but is not linear.

In Figure 4, we can see the profit curve. The profits of Experiments No.6, No.5, No.10, and No.11 fell sharply at the beginning. This has a big impact on their overall profit.

After manually looking at the policy generated by these experiments, I found that they made a wrong prediction at several key points. For experiment 10:

1) action 8 buy 731.530029
2) action 9 buy 750.460022
3) action 10 buy 784.5
4) action 11 buy 722.81
5) action 12 buy 703.87
6) action 13 buy 667.85
7) action 14 sell 682.74

The prediction labels for action 11 -13 are wrong. The stock price goes down starting at action 11. It keeps going down from action 11 to action 13. However, the prediction label for these actions is 1, which make the system buy more stock until action 14. At action 14, the stock price raises, but the prediction label is 0. Therefore, the trade system sells all stock, which causes a huge loss (about -$332.67). The experiments No.6, No.5, No.10 also have similar situations. Therefore, although the accuracy is increased, the profit may decrease by some action base on a wrong prediction. For experiments No.10, it has in total performed 232 actions and its owns $890.43 in google stock. So, about 1.71% action makes the whole system lose about 37.36% profit.

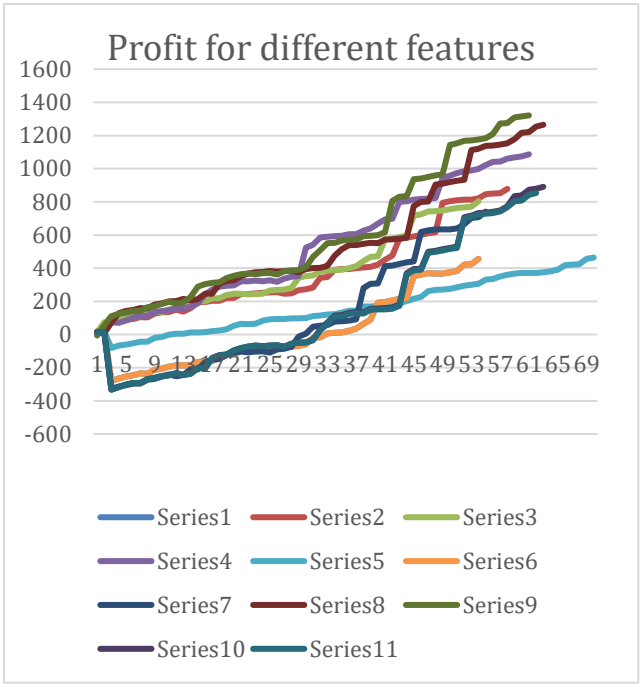


Figure 4 profit changes for different features 2

Conclusion [Jian Cui]

From the experiments on Naïve Bayes model, we can find the most efficient combination of features is opening price, highest price, lowest price and closing price. These features are classic attribute considered by people in stock market trading. The Naïve Bayes model gets 84% accuracy on the prediction, which means it can make a somewhat accurate trend prediction for stocks. However, as shown accuracy does not guarantee profit. The Naïve Bayes model treats every prediction equally. You may be able to get the right results at some worthless point and get the wrong result at some key point. In this case, you will lose money on your trading. Another flaw is the naive Bayesian model is not suitable for more detailed predictions. In our experiments, the selling price is the closing price, but you may get more profit if you sell at the highest price. It is very difficult to predict the highest price by using Naïve Bayes. Because the feature will be too complicated to decide. Choosing features is the most important part of this model. As the experiments above show, the price difference and the difference of slope are bad features. All the test who involved them lose a lot of money in the early game. Thus, without a good feature, the Naïve Bayes model will fail.

Background [Ben Struhl]

A Long short-term memory (LSTM) network is a type of recurrent neural network (RNN) that is specifically designed to solve classification and regression tasks on sequences of data which require the given network to memorize state information over a long period of time to perform optimally. An LSTM network consists of 6 key parts that allow the network to memorize information across multiple given data points, in a training data sequence. [6] [1]

- a cell state (C_t) which acts as the internal memory of the network from input instance to input instance
- a hidden state (H_t) which holds the output of a input data instance in a series of input data instances
- an input gate layer (I_t) which controls which values in the cell state to update, given the last hidden state, and input data instance
- a forget gate layer (F_t) which decides how much information we should forget in the cell state given the last hidden state, and input instance.
- an output gate layer (O_t) which decides what information should be flow into the next hidden state, given the last hidden state, current input instance, and cell state which has been filtered by a full connected tanh layer
- a tanh layer which outputs ($\hat{C}_{t-1}$), which contains a vector of new values the network might want to update cell state with based on the output of the input gate layer

for the activation functions within a standard LSTM network we use the hyperbolic tangent and the sigmoid function, however for the LSTM model in our network we will use the hard-sigmoid function, which approximates the sigmoid function, due to it being faster to compute. We define all three functions below (where σ= sigmoid and hσ = hard sigmoid)

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} \quad \sigma(x) = \frac{1}{1+e^{-x}} \quad h\sigma(x) = max(0, min(1, (x*0.2)+5))$$

To train the weight vectors and biases within the network we first run the network on our training data. We split up the data into sequences of N consecutive days, and set the label of each of these training instances equal to the N+1 day. The goal of this is to have the network accurately predict the label value given the N previous days. This variable N which controls the length of sequences we feed into the network is called the timestep hyperparameter. Then using Back propagation through time [4], with mean squared loss as the cost function [5], we calculate the adjustments we need to modify the weights, and biases by. We then keep doing this until we've surpassed the number of given iterations and then return the weights and biases our network will use. Also for the LSTM network we implement dropout layers to avoid overfitting [3]. This training algorithm is detailed below

```
train-lstm: list-of-input-vector-seqeuences, input-vector-labels, iterations{
        intialize weights and biases with random values
        while i < iteraions :
                Output = []
                For input-vector-sequence in list-of-input-vector-sequences:
                        Output_i = feed-forward-lstm(weights, biases, input-vector-sequence)
                Backprop-error_i = back-prop-through-time(weights, biases, Output,hσ,tanh)
                weights, biases = update-weights(weights, Backprop-error_i)
        return biases and weights
}
```

To calculate the output of the network we run the corresponding feed forward algorithm that runs the LSTM network, which is defined below. For $H_t, C_t, H_{t-1}, C_{t-1}, X_t$, their dimensionality is equal to that of the input vectors in the sequence of input vectors given.

```
feed-forward-lstm: weights, biases, input-vector-sequence {
        W_i, W_o, W_c, W_f = weights
        b_i, b_o, b_c, b_f = biases
        H_t, C_t, H_{t-1}, C_{t-1}, X_t = empty vectors
        for each input-vector in list-of-input-vector-sequences:
                X_t = input-vector
                f_t = hσ(W_f · [H_{t-1}, x_t] + b_f)
                i_t = hσ(W_i · [H_{t-1}, x_t] + b_i)
                Ĉ_t = tanh(W_c · [H_{t-1}, x_t] + b_c)
                C_t = f_t*C_{t-1} + i_t*Ĉ_t
                o_t = hσ(W_o · [H_{t-1}, x_t] + b_o)
                H_t = o_t*tanh(C_t)
                H_{t-1} = H_t
                C_{t-1} = C_t
        return H_t
}
```

Related work [Ben Struhl]

While we could of have used a traditional Artificial neural network (ANN), or RNN to predict stock prices, both approaches have problems that an LSTM network addresses. An ANN has no ability to encapsulate state. Therefore, if we wanted an ANN to make a prediction based on a series of previous prices, it would be unable to store any contextual information at the end each instance that might aid in computing an optimal result, such as price trends and current momentum. An RNN fixes this by making the network save its output from a given input vector and use it as part of calculating the output of a new input vector. This saved vector is called the hidden state, and it allows an RNN to accumulate state information it might find useful, over a sequence of input vectors. This ability would aid our algorithm in keeping track of price trends. However, an RNN has its own pitfalls. An RNN has issues keeping track of long-term dependencies [2], and also tends to exhibit the vanishing gradient problem [6] which matters if we want our model to pick up on long term price patterns. The way these two different models are related to an LSTM network

is that an LSTM network is a type of RNN, which itself is a specialized version of a ANN. However, an LSTM model is different than a normal RNN due to it having a cell state, in addition to a hidden state. This cell state solves the above stated issues by allowing the network to store memory for long term and short-term periods of time over multiple data instances.

Project Description [Ben Struhl]

To start, the goal of this experiment is to test how much profit an LSTM stock trading agent could make compared to a traditional "buy and hold" strategy. To calculate the results, we will use the following methodology.

The trading agent starting from the earliest day sequentially will parse the closing share price of Google from 2016/01/04 to 2017/11/10 and use a policy based on an LSTM network to output a trade corresponding to an action, while also keeping track of the current total profit, and list of share prices bought. The available actions the agent will have the potential to perform on each day are Buy, Sell or Hold. If the agent chooses Buy, it will add the current closing stock price to its list of closing share prices. Next if the agent chooses Sell, it will adjust its current total profit by using the following algorithm.

```
Total-Profit = Total-Profit + ((current-share-price * len(list-of-bought-share-prices) - sum(list-of-bought-share-prices))
```

Finally, if the agent chooses to Hold it will do nothing. At the end of the experiment we will use the current total profit as the main attribute to compare against. In addition, the agent will also keep track of winning trades, losing trades, and net winning trades. a winning trade is when the agent chooses the Sell action and increases its total profit. This is in contrast to a losing trade which is when the agent chooses the Sell action and decreases its total profit. Finally, net winning trades would be the total amount of winning trades minus the total amount of losing trades over the course of the test. We use these three values as a way to see how the total profit at the end is generated. It shows if the agent was consistently successful, or made a few very profitable trades. Additionally, the agent can also store information after each parsed trading day to aid its decision making. For training, the agent can also only use Google's closing stock price from 2014/03/27 to 2015/12/31. Finally, the agent will use "Buy and Hold" as the baseline policy to compare against. In "Buy and Hold" the investor buys one share of a company and waits to sell it until a certain date, hoping the stock price goes up. For this experiment this consists of buying 1 share of Google stock at beginning of the test and selling it at the end. The total profits generated from this strategy is $286.23, which is the number the will test against. I will also compare the performance of the agent trained on different timestep values

To generate a policy, I will have to first train my LSTM network with a given timestep value called n. Then when the network is training, I will select n subsequent prices at a time as a training sample from the given training set, and have the label be the n+1 price, with the intent on training the network to use the past n days to predict the next day's price. Then using that trained network I output the following policy.

```
n = some value specified
predict-next-price-func = trained-lstm-network-with-time-step-n(n)
profit = 0
shares = []
total-stocks-witnessed = 0
n-days = previous n days of closing stock prices
// current-price is the current day closing stock price
lstm-trade-agent: current-price{
        if total-stock-witnessed + 1 is less then n: increment total-stocks-witnessed and return Hold
        if predict-next-price-fun(n-days, current-price) > current price:
                share.append(current-price)
        else if predict-next-price-fun(n-days, current-price) < current price and shares.length() > 0:
                profit = profit + ((current-price * len(shares) - sum(shares))
        else:
                continue
for each closing-stock-price in test-set:
        lstm-trade-agent(closing-stock-price)
return profit
```

In essence if the network predicts the next day's price will be higher than the predicted price, given the last n days plus todays current price, it will buy. Else it will sell if it can, and in the case where the next predicted price match's todays price, it will hold. To build and train the LSTM network I used the following architecture and hyper parameters. The network itself is 4 connected LSTM layers with 50 hidden units as output, with a densely connected layer added to the end, and has dropout layers behind every LSTM layer with a dropout of 20%. Then the initial LSTM network will take an input vector equal to the length of the time step, and the output dense layer will be just one unit. The activation functions used within the network are the hard sigmoid and hyperbolic tan function, as mentioned in the background. Finally, I will use mean squared error as the loss function, and use the Adam optimizer to help improve the performance of the network.

Experiments [Ben Struhl]

For my hyper parameters, I will choose to experiment with different time step values to see if my network will perform better if given more previous day closing prices. My hypothesis is that if the LSTM network has more previous prices to consider, it could see better price trends in the data, and thus make a more accurate prediction on what the next subsequent price would be. Therefore, I will choose to train the network with a time step set at 1, 7, and 28. These numbers are specifically chosen due to the values corresponding to one day, week, and month of information for the network, which is a common time frame for humans to look at to predict price patterns. Other than that, I choose a batch size of 32 and will train the network for

150 episodes. After running my experiments, I gathered the following results

Time step value	total profits	net good trades	total good trades	total bad trades	standard deviation of error of predicted price to real
1	$792	139	186	47	13
7	$2752	183	210	27	27
28	$3411	135	51	16	44

Figure 1: Results of LSTM trading agent

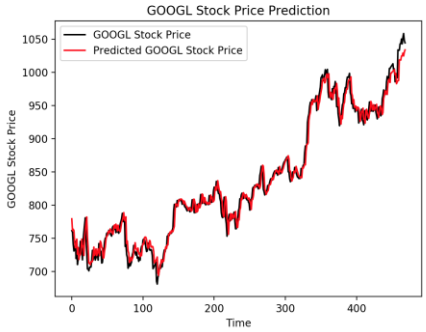


Figure 2: Google Stock Price Prediction time step = 1

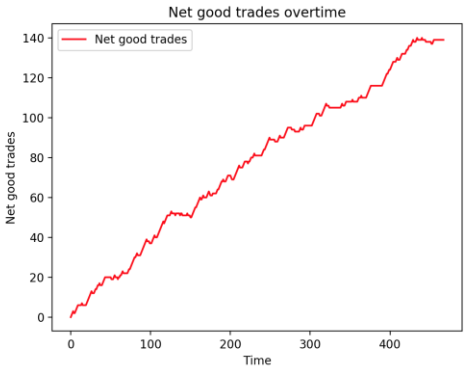


Figure 3a: Net good trades overtime time step = 1

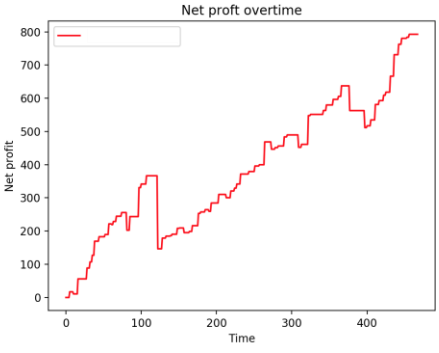


Figure 3b: Net profits overtime time step = 1

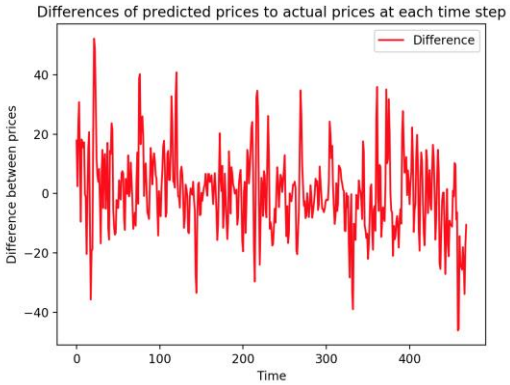


Figure 4: differences of predicted prices to actual time step = 1

As we can see in Figure 1, 3b, 6b, and 9b the LSTM trading agent beat the profits of the Buy and Hold strategy across all chosen time step values. However, one can also observe that for each time step, the way profits were generate drastically differed. First let's look at when the network did at time step value of one. This makes the network behave similarly to a standard Artificial Neural Network (ANN). a standard ANN is similar in function to a LSTM network, except it doesn't have a way to store output from training example to training example. This makes network treat every training example as a singular black box with no relationship to other examples in the training set. However, when looking at what the network predicted, a seemingly perfect chart is shown in Figure 2. However, this graph is misleading. Historically, day to day stock prices do not move drastically, and for all variations of the LSTM network used for this experiment we are only having the network predict price from day to day. Therefore, while at each time step the network might not be off by that much from the predicted target, the margin between the agent choosing between buying and selling is thin, which makes small errors affect performance extremely. In Figure 3a and Figure 2 you can see that the one step agent generally makes good moves and follows the trend line well. However, when you look at the standard deviation of error for time step one in figure 1 and figure 4, the network seems to be randomly guessing prices with a near constant error rate. This also backed up Figure 3b which shows the net profit wildly fluctuating overtime.

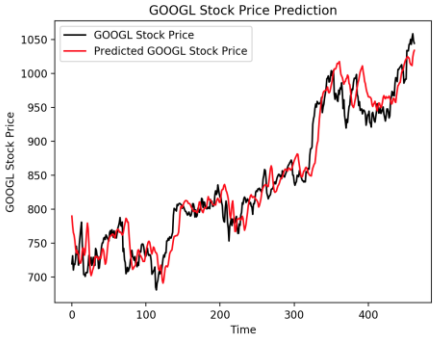


Figure 5: Google Stock Price Prediction time step = 7

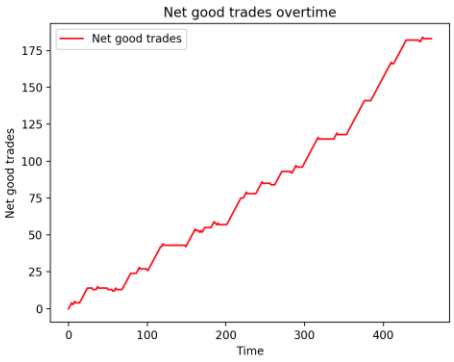


Figure 6a: Net good trades overtime time step = 7

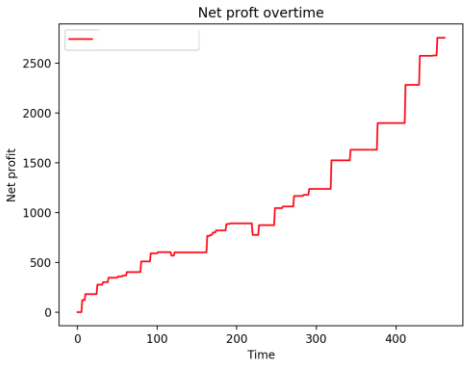


Figure 6b: Net profits overtime time step = 7

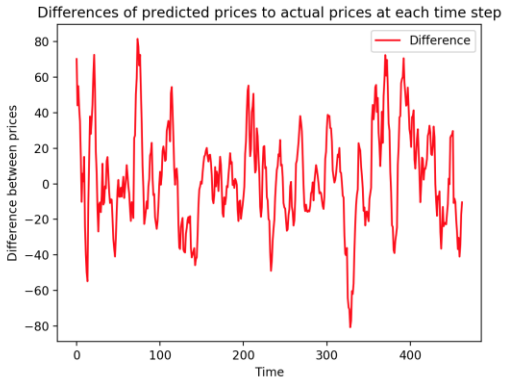


Figure 7: differences of predicted prices to actual time step = 7

When time step = 7 we can see that we get much better results than the time step value = 1 network from Figure 1. The total profit is more than double that of time step = 1, with a proportionally smaller number of bad trades. This could be attributed to the fact that with a time step of 7, the network now has the ability to take advantage of its ability to hold state. When given a training instance, it can now store price data as it goes through each closing stock price in the sequence. This allows the agent to have the ability to look for price trends and momentum within the stock, which in turn would make the agent think more long term. This is supported by figures 6a and 6b which shows a pattern of long plateau's followed by exponential gains in net good trades and net profits. This would suggest that the agent is correctly predicting upward trends in the closing stock price, and choosing to buy and not sell until the trend reverses to which the agent then exploits to for a greater profit. Also supporting this fact is Figure 5 which shows the network predicting prices which aren't as tightly packed to actual price as Figure 2. In addition, the networks higher standard deviation of error (Figure1) and less chaotic graph in Figure 7 when compared to that of Figure 4 show that the network has a more consistent learned methodology for output.

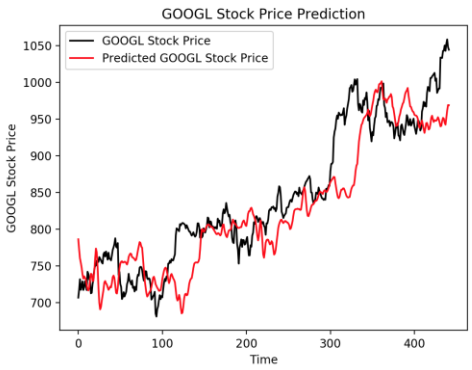


Figure 8: Google Stock Price Prediction time step = 28

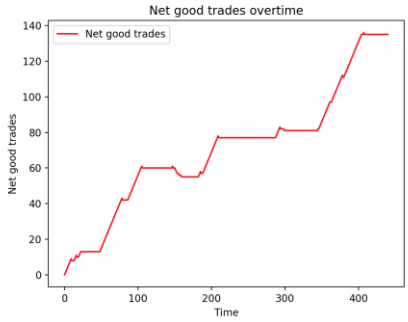


Figure 9a: Net good trades overtime time step = 28

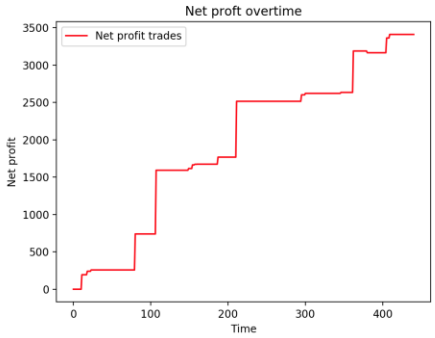


Figure 9b: Net profits overtime time step = 28

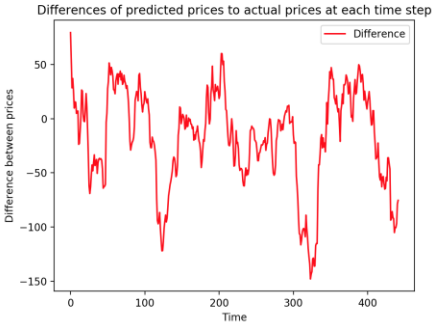


Figure 10: differences of predicted prices to actual time step = 28

for time step = 28, we also see the same pattern amplified from time step = 7. From Figure 1 the total profit has almost doubled from time step = 7, our standard deviation of error has almost doubled, and from Figure 10 the network error fluctuates wildly over time. This might be attributed to extra noise in the data generated by the large amount of previous day prices the network now uses. One phenomenon we can see now is that the network's prediction seems to be the previous true price of the network a number of days before, as seen in Figure 8. This might also be attributed to large amount days we give the network to consider before predicting a price. Finally, in Figure 9a and 9b we can see that the that net good trades overtime follows a similar pattern to that of Figure 6a and 6b, just with more extreme plateaus and gains.

Conclusion [Ben Struhl]

As we can see from the above results, the LSTM network can use closing stock price of previous days to successfully predict the closing stock price of the next day. While it's accuracy might be aided by the fact that historically stock prices do not move by much from day to day, we can safely assume it can learn more powerful features given more days to work with, which is backed up by the figure 1's general trend of increased profits, given an increased timestep. When the LSTM network is given a time-step value of 1 we can correlate its noisy error graphs and thrashing profit graph in figure 4, and 3b with the its limited perspective. With a time-step value of 1 the network can only look at one price before giving a prediction. While it can seemingly learn that the prices from day to day change with limited magnitude, being given one day prevents it from learning of any price movement via previous day prices. This is analogous to training a pong agent with only one frame. Without more than one frame it has no way of telling which way a projectile or enemy player is moving, making it have to guess which direction it should go in. However, in the end we can conclude that day traders can do a lot better than buy and hold by using an LSTM trading agent.

References

DQN References

[1] Xin Du, Jinjian Zhai, Koupin Lv. *Algorithm Trading using Q-Learning and Recurrent Reinforcement Learning: 5*

[2] Simonini, Thomas. *"Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and Fixed..." FreeCodeCamp.org, FreeCodeCamp.org, 6 July 2018, medium.freecodecamp.org/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682.*

[3] Zucchi, Kristina. *"4 Common Active Trading Strategies." Investopedia, Investopedia, 4 Dec. 2018, www.investopedia.com/articles/active-trading/11/four-types-of-active-traders.asp.*

[4]Sanjeevi, Madhu, and Mady. *"Ch:13: Deep Reinforcement Learning - Deep Q-Learning and Policy Gradients (towards AGI)." Medium.com, Medium, 10 Sept. 2018, medium.com/deep-math-machine-learning-ai/ch-13-deep-reinforcement-learning-deep-q-learning-and-policy-gradients-towards-agi-a2a0b611617e.*

Alvin P, Umberto E., and Eleni V. *Modelling Stock-market Investors as Erinforcement Learning Agents*: 3-5

Yang Wang, Dong Wang, Shiyue Zhang, Yang Feng, Shiyao Li, and Qiang Zhou. *Deep Q-trading*: 5–8.

Bayes References

[1]Margaret H Dunham. *eds. 2006. Data Mining: Introductory And Advanced Topics.*

LSTM References

1. Olah, C. (2018). *Understanding LSTM Networks -- colah's blog*. [online] Colah.github.io. Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ [Accessed 12 Dec. 2018].
2. Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jurgen Schmidhuber (2001). Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. In: J Kolenand S Kremer (eds.), *A Field Guide to Dynamic Recurrent Networks* (pp. 237–243). *Keywords:Recurrent neural networks and Sequence learning.*
3. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *J. Machine Learning Res.* 15, 1929–1958 (2014).
4. P. Werbos. Backpropagation through time: what it does and how to do it. Proceedings of IEEE, 1990.
5. Z. Wang and A. C. Bovik, "Mean squared error: Love it or leave it? A new look at signal fidelity measures," IEEE Signal Process. Mag., vol. 26, no. 1, pp. 98–117, Jan. 2009.
6. S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. Neural Computation, 9(8):1735–1780.