Buying and Selling Stock with Q-Learning

Sarah Hensley

***Abstract*—Finding a strategy to invest in and profit from the stock market is a problem of great interest. We apply Q-learning to stock prices from Alphabet, training on data from the years 2013 - 2017 and testing on 2018. While the learned policies perform well on the training data, they perform poorly on the test data. This is likely indicative of vastly different behavior in Alphabet's stock between the two time periods.**

I. Introduction

Wisely investing in the stock market can be very profitable. However, stock prices fluctuate unpredictably, making investing a difficult task. As a result, finding a way make optimal decisions about when to purchase and sell stock is a problem of great interest.

The setting purchasing and selling stock lends itself well to reinforcement learning. The price of a given stock changes frequently, without a clear mechanism governing its transitions. There is a clear goal of earning as much money as possible, which can be easily quantified when determining a reward function. Finally, because stock prices are constantly changing and the stock market has existed for many years, there is plenty of data on stock prices available. Together, these factors suggest that model-free reinforcement learning from the large available data set could offer a way find an optimal way to invest in the stock market.

Previous Work

Applying reinforcement learning to stock markets has attracted interest that has both taken the form of papers aimed at researchers and blog posts aimed at amateur investors.

Chen et al. combine Sarsa learning with genetic programming to maximize profits from buying and selling sixteen stocks. They trained with three years of data and tested with one year, ultimately achieving performance better than a baseline policy with fourteen of these stocks [2]. Lee defines a Markov decision process that uses the daily opening and closing prices as well as the high and low prices as a state; to this, she applies a temporal difference algorithm combined with a neural net. However, this failed to produce desirable results [4].

Outside of papers, there exist advice blogs for amateur investors. Britz explains the basics of developing a partially observable Markov decision process suited for trading Bitcoin without developing a specific model or running experiments. Instead, he aims his explanation at an audience familiar with the basics of machine learning, with the intent of teaching them enough so that they can design their own models [1].

II. Experimental Setup

A. Model

The problem of investing in stocks is quite general, as the number of companies to consider investing in is large. Thus, we restrict to the case of investing in Alphabet (stock symbol GOOGL). Investing in the technology sector has been an area of great interest in recent years, and Alphabet has existed as a publicly traded stock and been a leader in the technology sector for many years, which provides enough stock data to use for reinforcement learning.

With this choice, we can define a model. This is well-suited to be modeled as a Markov decision process, as the price of the stock changes over time and the investor must decide when to buy and sell stock. We define time steps such that a single time step is one day. The state is a two-dimensional vector, consisting of the percent increase in the stock price from market close on the previous day to market close on the current day, and the amount of stock currently owned. Each of these are continuous quantities. To make the problem tractable, both quantities were discretized. The percent change in stock price was rounded to two decimal places. The number of shares owned was limited to be an integer in the range $[0, 99]$. The upper bound is imposed to limit the size of the state space and to model having a limited amount of resources to invest. The action space consists of at most three actions: buy one share, sell one share, or do nothing.

The reward function is defined as the change in net worth as a result of the stock price's change on a particular day. More specifically, it is given as

$$R((p, n), a) = \frac{p}{100} * n * c \qquad (1)$$

for p as the percent change in stock price, n as the number of shares owned, and c as price of the stock at the close of yesterday. When summed over the entire time horizon, this is exactly equivalent to having a cost of the stock's price to buy a share and having a reward of the stock's price when selling a share. However, paying a penalty to purchase stock would serve in training to discourage ever purchasing stock. The above definition instead rewards balancing owning enough stock to profit but little enough to protect against a fall in stock price. The reward function relies on the previous day's stock price to calculate the reward; however, since the Markov decision process does not have access to the reward function, this does not violate any Markov assumptions. The transition function of the percent change in stock price is unknown, and the number of shares owned transitions deterministically according to the chosen action.

The data used in training the model was derived from Alphabet's stock values between January 2, 2013, and December 29, 2017, which corresponds to exactly five years of data, noting that the New York Stock Exchange is closed on weekends and holidays. To test the model, we used the stock values between January 2, 2018, and November 14, 2018, which corresponds to nearly a year of stock values. Together, these give $1,259$ training values and 221 test values. Over these six years, the percent change in closing stock price varied between -5.41 and $+16.26$, allowing this quantity to take $2,168$ values. Note that this is larger than the number of observed values; furthermore, this gives a total state space size of $216,800$.

B. Algorithm

This setting is well suited to Q-learning with local approximation. The goal is to approximate $Q(s,a)$ via learning $\boldsymbol{\theta}$ that satisfies

$$Q(s,a) = \sum_{s'} \theta_{s',a}\beta(s,s') = \boldsymbol{\theta}_a^T \boldsymbol{\beta}(s) \tag{2}$$

where $\beta(s,s')$ is a weighting function that satisfies $\sum_{s'} \beta(s,s') = 1 \ \forall s$ and assigns a higher value to state pairs that are more "similar", and $\boldsymbol{\theta}_a$ and $\boldsymbol{\beta}(s)$ are vectors whose dot product is equivalent to the above sum. We collect all of the $\boldsymbol{\theta}_a$ vectors so that they can be used to determine the best actions for the test data. With these, we generate a policy π by choosing

$$\pi(s) = \max_a \boldsymbol{\theta}_a^T \boldsymbol{\beta}(s) \tag{3}$$

for all states s. To obtain these $\boldsymbol{\theta}_a$ vectors, we apply Algorithm 1 [3].

Algorithm 1 Linear approximation Q-learning

```
function LinearApproximationQLearning
    $t \leftarrow 0$
    $s_0 \leftarrow$ initial state
    Initialize all $\boldsymbol{\theta}_a$
    loop
        Choose action $a_t$ based on $\boldsymbol{\theta}_a^T \boldsymbol{\beta}(s_t)$ and
            some exploration strategy
        Observe new state $s_{t+1}$ and reward $r_t$
        $\boldsymbol{\theta}_{a_t} \leftarrow \boldsymbol{\theta}_{a_t} +$
            $\alpha(r_t + \gamma \max_a \boldsymbol{\theta}_a^T \boldsymbol{\beta}(s_{t+1}) - \boldsymbol{\theta}_{a_t}^T \boldsymbol{\beta}(s_t))\boldsymbol{\beta}(s_t)$
        $t \leftarrow t+1$
    end loop
    return $\boldsymbol{\theta}_a$ for all $a$
end function
```

We initialize s_0 at the first data point (i.e. the percent change in stock value on January 2, 2013) and with zero stocks owned. Since we have no expert domain knowledge, we initialize all $\boldsymbol{\theta}_a$ with zeros. The next state s_{t+1} is composed of the next day's percent change in stock price and the number of stocks owned after taking an action, both of which are deterministic given t and s_t. The reward function is given in Equation 1.

There are several key implementation choices. First, there is the choice of the exploration strategy and the parameter(s) associated with it. Because our time horizon is very large, the optimal exploration strategy is intractable; instead, we choose to implement the softmax strategy for its ease of implementation. This has the precision parameter λ, which controls the tendency towards exploration or exploitation. Next, there is the choice of $\boldsymbol{\beta}(\cdot)$. We choose a nearest-neighbor-like function, in which states that have both percent change and number of stocks within a small range around the current state are given equal (non-zero) weight, and all other states are given zero weight. If states within the range are invalid states (e.g. have negative shares), the weights are increased on the closest valid state. This offers ease of implementation; note that a function akin to a Gaussian kernel would have many more states to sum over, increasing computational complexity. This choice of $\boldsymbol{\beta}(\cdot)$ has parameters controlling the range of states that are given weight. Finally, there are the parameters inherent to this algorithm, namely the number of times to loop through all of the data, the discount factor γ, and the learning rate α. We explore the effects of different setting for these parameters in our results.

III. Results

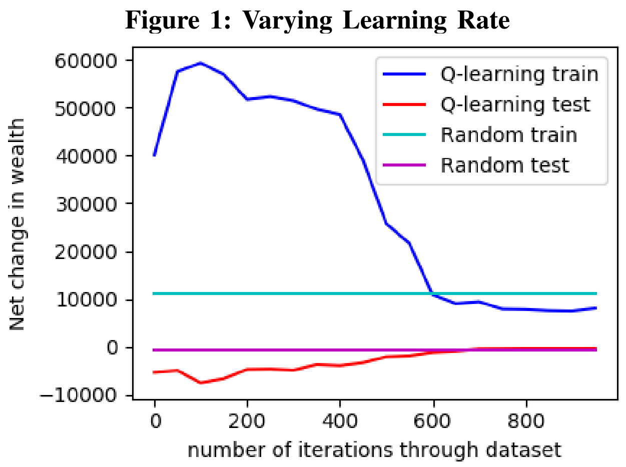
To evaluate the results of Q-learning, we generate a baseline for comparison. This baseline is calculated as the average result of $100,000$ policies obtained by choosing each day's action uniformly at random, with the same bounds on the amount of stock owned imposed. This baseline is a net gain of $\$11,030$ over the training data and a net loss of \$630 over the test data.

The first parameter we tune is the number of iterations through the data set, as this affects the time taken for the algorithm to run. Setting a fixed number of iterations risks wasting computation if the result converges quickly and also risks ending before convergence. Thus, we end iterations when the policies for both the training data set and test data set do not change for 100 iterations. If the policies are still changing after $1,000$ iterations, we end iterations early for practicality.

Next, we consider the precision parameter λ of the softmax function. As $\lambda \to 0$, the softmax function approaches a uniform distribution, encouraging exploration. In contrast, as $\lambda \to \infty$, the softmax function approaches a delta function centered on the action that maximizes $Q(s,a)$, encouraging exploitation. During early iterations, we want to encourage exploration, to mitigate the long-lasting effects of a poor initialization. During late iterations, we want to encourage exploitation, so that computation is not wasted on states and actions that yield poor results. To obtain this, we initialize with $\lambda = 0.5$, and increase λ by 0.5 every 100 iterations.

The next parameter we consider is the learning rate α. Setting $\alpha = 1$ allows for fast convergence, but the large updates to $\boldsymbol{\theta}_a$ often cause overflow error. Instead, setting $\alpha = 0.1$ achieves a reasonable convergence speed without causing errors. Setting $\alpha = 0.05$ achieves poor performance over the test set, yet good performance over the training set, as shown in Figure 1.

The next parameter we consider is the discount factor γ. Because this is a finite horizon problem, restricting $\gamma < 1$ is not necessary. However, in practice, setting $\gamma = 1$ causes overflow errors when $\alpha = 0.1$, and setting $\alpha = 0.05$ leads to the same poor performance on the test set as shown in Figure 1,



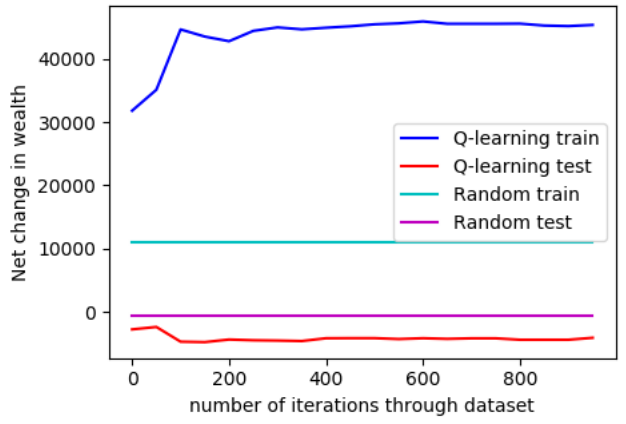


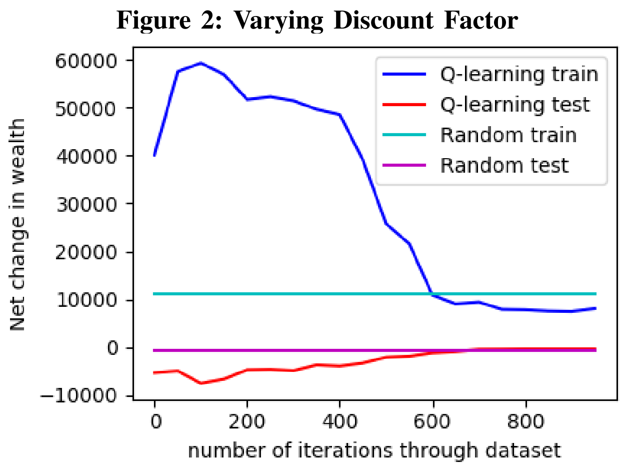
Fig. 1. The effect of varying α on the training and test policies. The top figure shows $\alpha = 0.1$, while the bottom shows $\alpha = 0.05$. The smaller step size offers better performance on the training data set, but worse performance on the test data set.

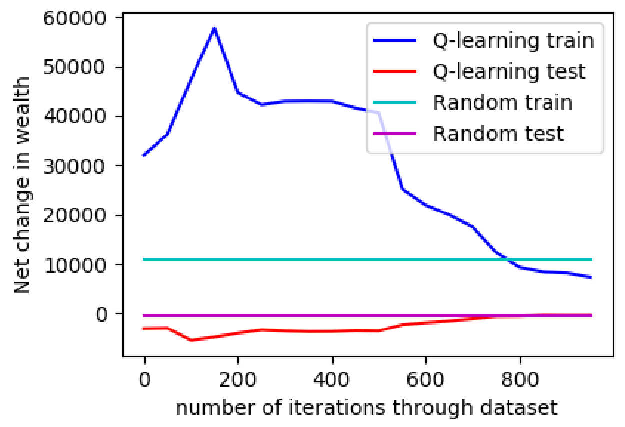
although we do not present these results in this paper. Holding all other parameters fixed, the effects of varying γ between 0.4, 0.55, and 0.7 can be seen in Figure 2. When varying other parameters, we fix $\gamma = 0.55$. Ultimately, the choice of γ has little effect on the score of the final policy.

For the range of states that have non-zero weight in the linear approximation, we consider the problem setting. The number of shares owned exists in the range $[0, 99]$; in this context, owning 10 shares is vastly different than owning 20 shares. To reflect this, we set the nearest-neighbor range as within ± 2 shares of the current amount owned. In contrast, the percent change in stock is discretized up to one hundredth of one percent, for a total of 2168 possible values. This setting suggests that having a larger number of "similar" values will allow better generalization, so we set the nearest-neighbor range as within ± 25 values of percent change. We also consider the effects of a smaller nearest-neighbor range and a larger nearest-neighbor range, as shown in Figure 3.

IV. Discussion

Over all iterations and all parameter values, Q-learning never achieved a positive change in net wealth over the test data





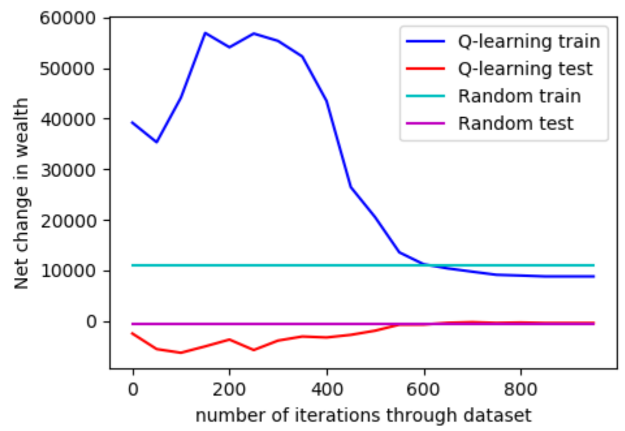


Fig. 2. The effects of varying γ on the training and test policies. From top to bottom: $\gamma = 0.7$, $\gamma = 0.55$,$\gamma = 0.4$. The final training results for each are \$8,076, \$7,328 and \$8,826, respectively, and the final test results for each are \$ − 384, \$ − 371, and −\$380, respectively.

Figure 3: Varying Number of Nearest Neighbors

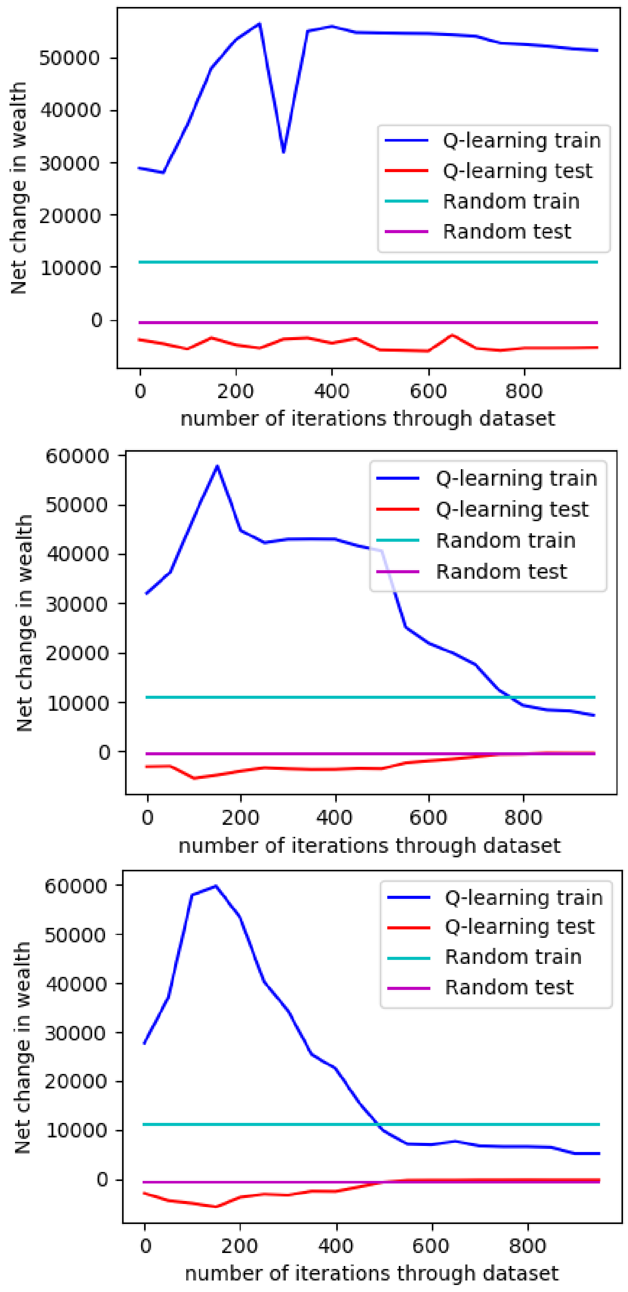


Fig. 3. The effects of varying the range of states used in linear approximation on the training and test policies. From top to bottom: $(\pm 15, \pm 1)$, $(\pm 25, \pm 2)$,$(\pm 40, \pm 3)$. The final training results for each are \$51,342, \$7,328 and \$5,202, respectively, and the final test results for each are \$ − 5,394, \$ − 371, and −\$201, respectively.

Figure 4: Comparison of Share-Purchasing Policies

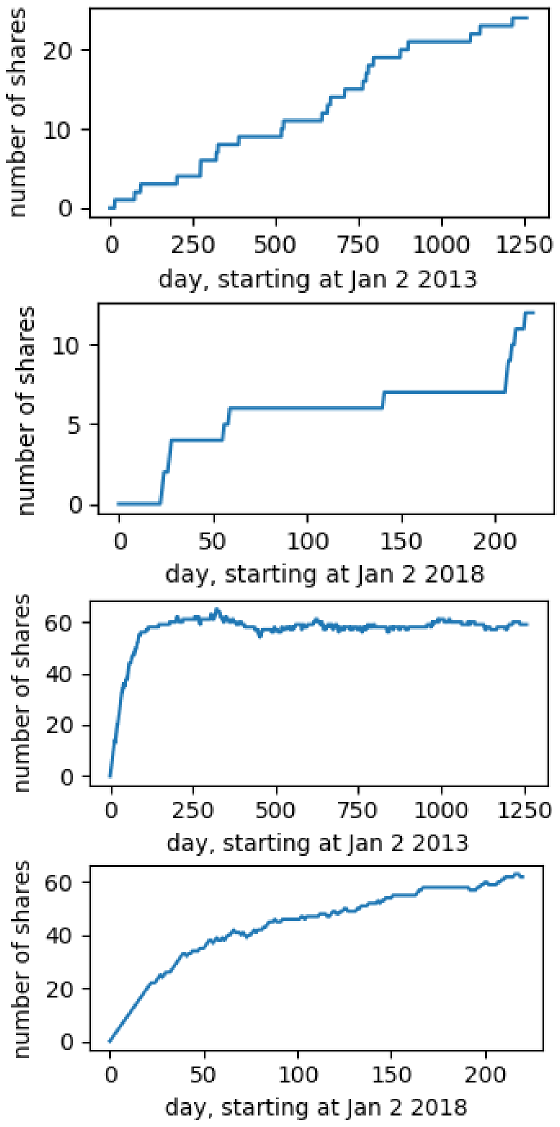


Fig. 4. The number of shares owned over the course of a policy for the test data, as a function of time. The top two figures show the results for a policy that performs worse than average on training and better on test data, while the bottom two figures show the results for a policy that performs better than average on training and worse on test data.

set. However, it consistently achieved a positive change over the training data set, with a maximum value of over 60,000. Interestingly, parameters that caused Q-learning to score better on the test data caused always caused it to score worse on the training data.

Fundamentally, this result is indicative of the training data and test data being more different than anticipated; however, there are several high-level explanations of this issue. First, it could be the result of overfitting, a common problem in machine learning. This occurs when the model optimizes its performance on the training data to an extent that hurts performance on any other data sets. However, this is typically associated with allowing the algorithm to iterate for a long time, letting its parameters converge to the optimal values for the training data. This does not seem to be the case, as the performance on the test set is poor over all iterations.

One plausible explanation is that information about Alphabet's stock prices from 2013 to 2017 is not useful for making decisions in 2018. Alphabet had an immense gain in value between 2013 and 2017, with the price per share roughly tripling over this time period. This favors policies that purchase stock early, and typically keep the number of shares owned large. In contrast, over the course of 2018, Alphabet's stock has lost approximately 2% of its value. This would favor keeping the overall amount of stock owned small. Examining the number of shares owned as a function of the day for the final policies over the test set, as seen in Figure 4, reinforces that this is what is occurring.

Tellingly, in all trials, the iteration at which the Q-learning policy started to score better than the random policy over the test set was exactly the same point as when the Q-learning policy started to score worse than random over the training set. This supports the explanation that the strategies that perform well (i.e. better than average) on the training data set are the same ones that perform poorly (i.e. worse than average) on the test data set, and vice versa. This also fits the explanation that policies that perform well on the test set favor keeping the number of shares owned small, as this would pull the net change in wealth towards zero for both the training and test policies.

Finally, a wider nearest-neighbor range means that the exact current state influences the chosen action less, instead "spreading" the influence out to nearby states. Intuitively, this would result in a more conservative policy, as the optimal action taken from states that are "far apart" would differ, encouraging the policy to favor holding instead of buying or selling shares. The results match this intuition, showing that increasing the size of the nearest-neighbor range brings net change in value closer to zero for both the training and test data. This reinforces the explanation that policies that perform well on the test set are ones that shy away from purchasing many shares.

V. Conclusions and Future Work

From this work, we conclude that stock information from Alphabet between 2013 and 2017 is only somewhat useful for making decisions about when to purchase Alphabet stock in 2018. While Q-learning was able to learn a policy that performed better than a random policy, it could not make a profit. This is likely because of how differently Alphabet's stock prices behaved during the two time periods.

Future work could extend this analysis to a stock with more uniform behavior, such as an index fund. This would likely have less aberrant behavior, and so the training data and test data would be more similar. With this change, future work could also be done to more thoroughly optimize the parameters, especially in relation to each other. Doing so would require a more efficient implementation of this work or a more powerful computer, as each policy takes roughly twelve minutes to compute. Finally, future work could explore the effects of a different exploration strategy and a different linear approximation function.

References

[1] D. Britz, "Introduction to Learning to Trade with Reinforcement Learning" Feb 2018. Available http://www.wildml.com/2018/02/introduction-to-learning-to-trade-with-reinforcement-learning/. [Accessed December 3, 2018].

[2] Y. Chen, S. Mabu, K. Hirasawa, and J. Hu, "Trading Rules on Stock Markets Using Genetic Network Programming with Sarsa Learning". In Proc. Genetic and Evolutionary Computation Conference '07 2007.

[3] M. Kochenderfer, "Model Uncertainty" in *Decision Making Under Uncertainty*. Cambridge, Massachussetts: MIT Press, 2015.

[4] J. W. Lee, "Stock Price Prediction Using Reinforcement Learning". In Proc. International Symposium on Industrial Electronics '01 2001.