Optimizing Stock Trading Strategy With Reinforcement Learning

Wilson Liang liangwil@stanford.edu
Bradford Lin bradfordlin@stanford.edu
Mai Lan Nguyen nmailan@stanford.edu

1. Introduction

Profitable stock-trading strategy is crucial to investment companies. Optimizing allocation of capital is challenging (even for experts), as it requires traders to take all relevant factors into consideration in a dynamic and complex stock market. Given the stochastic nature of the stock trading problem, one way of viewing it is by formulating it as a Markov Decision Process (MDP) and using reinforcement learning to solve for the optimal strategy. In the RL framework, an agent is trained in the environment through trial and error exploration of some action strategies. In contrast to supervised learning, where labelled data is provided, the RL algorithm receives a reward function, which indicates whether the agent is doing well or not. The algorithm's task is to find the optimal policy for obtaining large rewards.

In this project, we explore and compare the potential of three RL algorithms: discretized Q-learning, Hill Climbing, and Deep Q-learning, to optimize stock trading strategy and maximize our cumulative wealth. As input, we define two versions of the MDP (one with discretized state space for Q-learning and Hill Climbing and one with a continuous state space for Deep Q-learning). We output a policy (a sequence of actions) that maximizes our cumulative wealth. We measure and compare the performance of our algorithms against a random trading strategy and some common tried-and-true stock-trading strategies, such as "buy-and-hold" and MACD.

Our results indicate that Hill-Climbing, which is a policy-iteration based algorithm has better and more stable performance than the discrete Q-learning and Deep Q-learning algorithms, both of which are value-iteration based.

2. Related work

Applications of machine learning methods in financial markets often focus on analyzing the performance of critic-only (Yang et al.) approaches of reinforcement learning, where a Q-value function (which approximates discounted future rewards) is used to learn the optimal action-selection policy.

In particular, many works limit themselves to examining Deep Reinforcement Learning techniques such as Recurrent Reinforcement Learning (Li et al., 2015) and fitted Q-iteration (Antos et al., 2007). One of the first papers on the application of RL in trading (Moody & Saffell, 1998) found that the results for reinforcement learning trading systems outperform the SP 500 Stock Index over a 25-year test period, and thus demonstrate the presence of predictable structure in US stock prices. In comparing the performance of the classical discrete Q-learning algorithm (Watkins 1989) to Recurrent Reinforcement Learning, it also showed that Q-learning can suffer from the curse of dimensionality (where the state space grows exponentially with state space dimension) and is more difficult to use than [the] RRL approach. Similarly, another paper (Le, 2018) also showed that fitted Q iteration helps alleviate the dimension problem that the basic Q-learning algorithm faces in application to trading.

Another work (Xiong et al., 2018), explored the performance of an actor-critic framework algorithm called Deep Deterministic Policy Gradient, which is an improved version of DPG. DPG combines the frameworks of both Q-learning and policy gradient. Compared with DPG, DDPG uses neural networks as function approximator. It maintains an actor network which maps states to actions and a critic network which outputs the value of action under that state. While the paper showed that DDPG achieved better performance than the Dow Jones Industrial Average portfolio allocation strategy, it didn't discuss DDPG in comparison to other reinforcement learning methods.

We extend previous works by directly comparing and contrasting policy-search-based techniques (Hill Climbing) with value-search-based ones (Q-learning and Deep Q-learning).

3. Problem Statement

Our goal is to find an investment strategy on a single exchanged traded fund (SPY) that would maximize our cumulative wealth. We benchmark our results against the tried-and-true "buy-and-hold" and MACD techniques.

For our discretized RL techniques (tabular **Q-learning and Hill Climbing**), we define a corresponding **MDP** as follows:

- State s: set of states (u, t), where $u = 1$ indicates upward momentum (the average of the last five closing prices was higher than its 1-day-average (the closing price of the stock that day), and $u = 0$ indicates downward price pressure. Whereas, $t = 1$ if the agent can trade/buy, and $t = 0$ otherwise. For simplicity, we assume that an agent can be either all-in on cash or all-in on SPY. Hence, in total we have 4 possible states.
- Action a: set of actions on the SPY stock. Available actions include: buying, holding or selling. Certain actions are only accessible in certain states (i.e. you cannot buy when you are already in a stock position, and you cannot sell when you are already in a cash position).
- Reward r: the cumulative wealth when action a is taken at state s and results in arriving at new state s'

For our **Deep Q-learning method**, in turn, our **MDP** will be given as:

- State s: set of states (r, t), where $r \in [-1.0, 1.0]$ is the return difference between 5-day-average (geometric mean of immediate ratios of the closing prices - eg. $price_{day1}/price_{day0}$ - of the previous five days) and 1-day-average (current price divided by yesterday's price). Hence, r measures stock momentum. Whereas, $t = 1$ if the agent can trade/buy, and $t = 0$ otherwise.
- Action a: Here, actions are defined and follow similar constraints as in the MDP above.
- Reward r: the change in wealth when action a is taken at state s and results in arriving at new state s'

4. Dataset and Features

We considered the stock market performance of the last ten years. Specifically, we tested our reinforcement learning trading strategies on the S&P 500 from January 4, 2010 to December 30, 2020, using the index fund SPY as our trading instrument. We pulled our data from Yahoo Finance, exporting the following features: date, open price, and close price. We split our data set into train and test data, as outlined in Table 1.

Table 1. Start and end dates for data breakdown.

DATA SET	BEGIN DATE	END DATE
TRAINING SET	JANUARY 4, 2010	DECEMBER 31, 2015
TEST SET	JANUARY 4, 2016	DECEMBER 30, 2020

Beyond the procedure described above, we did no other pre-processing on the data.

5. Methods

We explore 3 RL approaches to single stock trading: discretized Q-learning, Deep Q Learning and Hill Climbing. We develop a custom environment from scratch in Python for our experiments. Specifically, we start off with a wealth of $100,000, and our step function returns the value of our portfolio at the end of the day, given our action and state. We will evaluate and compare each of them based on their performances against some commonly known trading strategies such as MACD and Buy-And-Hold.

5.1. Discrete Q-Learning

In reinforcement learning, the goal of the agent is to interact with the environment by selecting actions that maximize expected cumulative reward (where the future rewards are discounted by γ):

$$R_0 + \gamma R_1 + \gamma^2 R_2 + ... \tag{1}$$

We define the optimal action-value function $Q^*(s, a)$ as the maximum expected return by following any strategy at state s and then taking action a:

$$Q^*(s, a) = \max_{\pi} E[R_t | s_t = s, a_t = a, \pi] \tag{2}$$

Given that $Q^*(s, a)$ obeys the Bellman equation, given by:

$$Q^*(s, a) = \max_{\pi} E[r + \gamma \max_{a'} Q^*(s', a') | s_t = s, a_t = a, \pi], \tag{3}$$

the Q-learning algorithm iteratively updates the approximation of $Q^*(s, a)$ for each s, a by the following rule:

$$\mathrm{Q}^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max Q(s_{t+1}, a) - Q^{old}(s_t, a_t) \tag{4}$$

where $r(s_t, a, s_{t+1})r_t$ is the reward, γ is the discount factor and $\max Q(s_{t+1}, a)$ is the estimate of the optimal future value. With our discretized definition of the MDP, our state-space can be expressed as a 2x2 matrix for our tabular Q-learning (Watkins 1989) (Table 2).

Table 2. 2x2 Q-table for tabular Q-learning

	STOCK	CASH
1-DAY $<$ 5-DAY	Q-VALUE	Q-VALUE
1-DAY $>$ 5-DAY	Q-VALUE	Q-VALUE

We run 100 iterations of Q-learning on the training data from 2010 through the end of 2015 and use the following hyperparameters for our Q-learning algorithm:

$$\alpha = 0.05, \gamma = 0.15, \epsilon_{\texttt{init}} = 1$$

$$\texttt{epsilon decay rate} = 0.99,\ \epsilon_{\texttt{min}} = 0.15$$

Specifically, we choose a small step size of 0.05 to allow for granular learning and updates of our Q-matrix. We also choose to discount future periods of wealth by quite a bit (our discount factor is 0.15) because we found that with lower discount rates, the algorithm did not converge well and the Q-matrix values were very close to each other, meaning that the margin to make a decision for our final policy is very narrow. Lastly, we choose a standard range for epsilon, starting at 1 in the beginning (since we enter the problem choosing random actions), and decaying by 1 percent each time, until we get to an epsilon lower bound of 0.15, meaning that 15 percent of the time, we would still choose our action during training rounds randomly–in the case that said random action ends up being more optimal than the current best option we have.

5.2. Hill Climbing

For our hill climbing algorithm, we do not keep track of rewards in a Q matrix. Instead, we start by generating a random policy, and then modifying a single action within the policy, running it on the entirety of the training set, and comparing the ending portfolio value against the previous best policy's ending portfolio value.

If the ending portfolio value of the current policy is better than the ending portfolio value of the prior best policy, then we update the best policy and best portfolio value. We iterate this algorithm and take 100 passes over the training data, where the algorithm converges every single time. We then test the performance of the algorithm on the test data, comparing to the strategies mentioned at the beginning of the section.

A potential drawback of hill-climbing is that we could get stuck at a local optimum. However, after running the algorithm multiple times, we found that overall, it converges quickly and rarely gets stuck on a local optimum. Secondly, it is possible that our data overfits the training set; however, given the small number of iterations and the fact that that hill climbing actually derives a solid trading strategy that has been well-researched and backed up by investment logic, the worry of overfitting may to a degree be alleviated.

5.3. Deep Q-learning with Experience Replay

Deep Q-learning is a generalization of discrete Q-learning, which directly deals with continuous state spaces. In contrast to the discrete Q-learning algorithm, Deep Q-learning uses a neural network to approximate the Q-value function. We use the same Deep Q-learning algorithm with Experience Replay as outlined in (Mnih et al., 2013). In Experience Replay, we continuously generate experiences (s_t, a_t, r, s_{t+1}) as the "labelled" data (where r is the label)

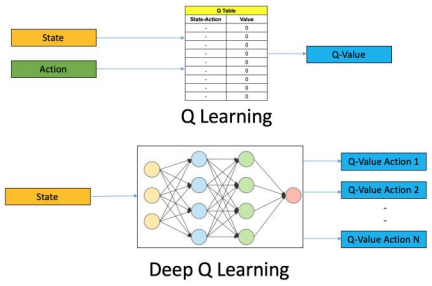

Figure 1. Representation of how Q-values are approximated in Deep Q Learning vs Q learning.

and store them in a fixed buffer, randomly sampling from the buffer to learn the network. This helps eliminate the inherent correlation in the sequence of experiences, since the sampled experiences are randomly ordered. "It also allows the network to learn from the same experience multiple times, which helps average the behavior distribution over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters" (Mnih et al., 2013). Instead of estimating the Q-value for each state (as in the discrete case), we update parameters θ of the Q-function using the Adam optimizer (which is an extension of stochastic gradient descent).

Our network is a simple 3-layer feed-forward network, which uses ReLU as the activation function for the first two layers, and outputs the *softmax* transformation of the final layer. We don't apply ReLU to the final layer and instead feed the linear transformation of the previous layer to *softmax*, so as not to truncate negative values. The network takes in a state and produces the Q-values for all possible actions as the output in a forward pass. Since we have a two dimensional state space and three possible actions, the overall network dimension is a $2 \to 64 \to 32 \to 3$, where the intermediate layer sizes (64, 32) are chosen arbitrarily. The optimal action determined by the network is the action corresponding to the maximal output value. We define the loss function as the mean-squared error of the predicted Q-values and the target Q-values:

$$MSE = (Q_{target} - Q_{local})^2 \tag{5}$$

Since this is a reinforcement learning problem and the target values are unknown, unlike labeled data in standard deep learning, we will need to estimate both the target and predicted values.

Since the target values and predicted values change with each iteration, there could be a lot of oscillation in the network. We mitigate this by maintaining two networks, a local network and target network. At each iteration, the local network is updated according to new data. Every c

iterations, the parameters of the local network are copied to the target network, making the training more stable since the target values are fixed for some duration c (Singh, 2019).

6. Experiments

To better understand the methods at hand, we conducted many different experiments. First, we tried discretizing with a different formulation of the state space, where the states constituted a tuple representing the percent change of 1-day average prices and 5-day average prices of the stock (discretized as integer values from -5 to 5), as well as whether the agent can still trade. Nevertheless, we found that this discretization performed worse in comparison to our simpler formulation of the state space (as outlined in the Problem Statement section), most probably due to the fact that features chosen for the worse-performing state space were not significant for the agent's decision-making process.

Prior to setting the hyperparameters as outlined above, we experimented with different values of the discount rate (i.e. $\gamma = 0.15$ and $\gamma = 0.99$), and found that we achieved faster convergence for $\gamma = 0.15$.

Most importantly, however, our experiment was in comparing the performance of our three different methods. We ran each algorithm with the above outlined hyperparameters. Our primary metric for measuring model performance was the ending portfolio value on the validation set, as this gives us a direct way to compare each of our policies against other well-known trading strategies.

7. Results & Discussion

7.1. Q-learning

After 100 iterations of Q-learning, our algorithm converges upon a policy that is essentially buy-and-hold (Table 3).

Table 3. Q-learning policy output

	STOCK	CASH
1-DAY $<$ 5-DAY	HOLD	HOLD
1-DAY $>$ 5-DAY	HOLD	BUY

The optimal policy, as calculated by Q-learning, says to buy SPY at the first date at which the 1-day average exceeds the 5-day average (i.e. the first day where we see upward momentum), and then hold into perpetuity.

When designing our Q-learning environment and state space, we had hypothesized that our Q-learning algorithm would have converged to MACD due to the way MACD trades based on momentum by comparing 1-day to 5-day averages; however, it seems that Q-learning converged to a policy that outperformed MACD. A possible reason why Q-learning/buy-and-hold outperform MACD is that unlike MACD, which is a reactionary, Q-learning learns to not miss the days where the market grew the most. History has shown that missing the five best trading days in the market cuts returns in by nearly a half, and so Q-learning produces a better result(Housman, 2020). For reference, the MACD trading strategy can be shown below (Table 4).

Table 4. MACD policy

	STOCK	CASH
1-DAY $<$ 5-DAY	SELL	HOLD
1-DAY $>$ 5-DAY	HOLD	BUY

7.2. Hill climbing

In Figure 2, we see the results of our Hill Climbing policy, tested against the other policies described in the methods and experiments section. We see that our algorithm converges to a policy that beats buy-and-hold, and therefore Q-learning, from the section above. The policy we obtain from hill climbing can be described in the following table (Table 5).

Table 5. Hill climbing policy output

	STOCK	CASH
1-DAY $<$ 5-DAY	SELL	BUY
1-DAY $>$ 5-DAY	HOLD	BUY

The optimal policy outputted from hill climbing resembles MACD in many respects. However, rather than sit out on cash when the 1-day average is less than the 5-day average (like in MACD), tit buys the stock whenever in a cash position, regardless of what the market is doing. Essentially, the hill climbing policy flees the market every time there is significant downward momentum, and re-enters every time it has the opportunity to do so (in cash). This is a good strategy, because as evidenced by many studies, missing even a few of the best trading days will cut cumulative returns significantly, especially since using moving averages as momentum indicators is considered a reactionary strategy.

Moreover, hill climbing also produced slightly superior results to buy-and-hold / Q-learning throughout 2016 to 2018, only lagging behind buy-and-hold / Q-learning in parts of 2019, perhaps due to the frequent entry and exit strategy in a more volatile market. We can conclude that our hill climbing policy would significantly out-perform buy-and-hold / Q-learning in a market environment where prices rapidly drop, and then increase. However, in a more volatile environment where prices zigzag and stay mostly flat, hill

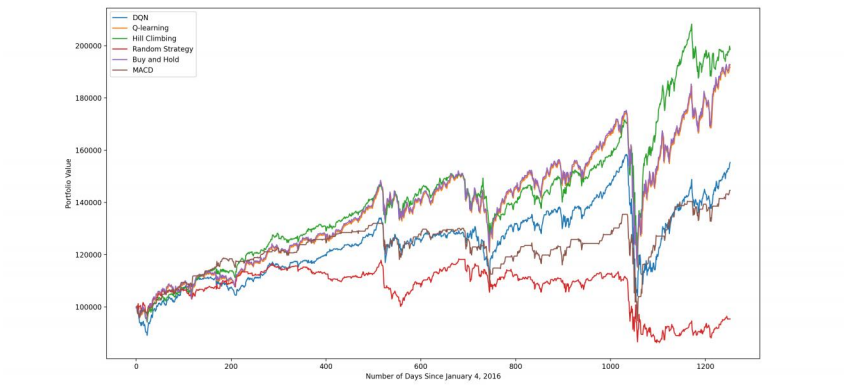

Figure 2. Various reinforcement learning policies on validation set.

climbing might do worse since it tries to exit the market during downward drops, perhaps missing moderate but otherwise positive trading days.

Furthermore, the policy-iteration method (hill climbing) outperformed the value iteration method (Q-learning). As policy requires solving a linear system of equations at each iteration, it can be expensive for problems with a large state space. We manage to avoid this problem by defining a discretized MDP with a small state space (Ng).

7.3. DQN

DQN produces a non-trivial policy for a continuous state space. Its best performance is slightly under the buy-and-hold. Hence, it does worse off than the best performance of the discretized Q-learning and Hill-Climbing algorithm.

7.4. Different Random Seeds and Model Instability

After running the methods with different random seeds, we conclude that the policy-search based algorithm, Hill Climbing, is the most stable strategy, as it produces a consistent outcome cumulative wealth across the runs. In contrast, Q-learning and Deep Q-learning produce very different outcomes for different seeds. Furthermore, due to the small state space, the variability in the output policy from the discretized Q-learning algorithm results in a larger difference in the output wealth than the Deep Q-learning. As the policy for the Deep Q-learning is more granular, and considers a larger state space, the variance of output cumulative wealth from the Deep Q-learning is smaller, and the Deep Q-learning averages around a policy that is similar to MACD.

The policy-search based algorithm is expected to produce a more stable result, as with each iteration the algorithm monotonically improves the policy by directly updating it to one that produces better results. In contrast, the value-search-based approach only estimates the policy from the approximation of the maximized value function. Hence, since we run the algorithms for a finite number of iterations, Hill Climbing might be producing more reliable results.

8. Conclusion

We find that Hill Climbing is the most stable method compared to the value-search-based methods: Q-learning and Deep Q-learning. Hill Climbing manages to find a policy that outperforms Q-learning and Deep Q-learning. This may be due to the fact that Q-learning and Deep Q-learning are value-search-based approaches that highly depend upon our estimation of the value function and state space definition. In our case Q-learning was doing worse because our discretized state space might not have captured all important information of an essentially continuous space problem. Hence, the Q-value function approximated by Q-learning was not completely accurate. On the other hand in policy iteration method we are guaranteed to have a policy at least as good as the policy at previous iteration and so we're constantly improving and our result is more reliable.

For future work, instead of running our algorithms for a set number of epochs, we could try to run them directly until convergence. Since we conducted our experiments on static data, our results were produced for simplified financial market dynamics, as our transitions are close to being deterministic. Hence, we suggest to conduct experiments on a simulation of market dynamics. Furthermore, after investigating our Deep Q-learning implementation, we found that the action values produced from the Q-network hover around a uniform distribution with a small margin of preference for any particular action. There might be some further work to be done through debugging or experimenting with the definition of the Q-network.

Acknowledgements and Contributions

BL developed and coded the Q-learning and hill climbing algorithms. WL and MN developed and coded the Deep Q-learning algorithm from (Singh, 2019). BL, WL, and MN all contributed to the preparation of the final manuscript. We would like to thank the CS229 teaching staff for their guidance.

References

Antos, A., Munos, R., and Szepesvári, C. Fitted q-iteration in continuous action-space mdps. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pp. 9–16, Red Hook, NY, USA, 2007. Curran Associates Inc. ISBN 9781605603520.

baeldung. Epsilon-greedy q-learning. https://www.baeldung.com/cs/epsilon-greedy-q-learning, 2021.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Housman, D. Why mistiming the market can be disastrous. https://www.thesimpledollar.com/investing/stocks/tempted-to-sell-missing-just-a-handful-of-the-best-stock-market-days-can-tank-your-returns/, 2020. Accessed: 2021-05-29.

Le, S. Algorithmic trading with fitted q iteration and heston model, 2018.

Li, X., Li, L., Gao, J., He, X., Chen, J., Deng, L., and He, J. Recurrent reinforcement learning: A hybrid approach, 2015.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.

Moody, J. and Saffell, M. Reinforcement learning for trading. In *Proceedings of the 11th International Conference on Neural Information Processing Systems*, NIPS'98, pp. 917–923, Cambridge, MA, USA, 1998. MIT Press.

Ng, A. Cs229 lecture notes: Reinforcement learning and control. http://cs229.stanford.edu/notes2020fall/notes2020fall/cs229-notes12.pdf.

pandas development team, T. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito,

Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Singh, U. Deep q-network with pytorch. https://unnatsingh.medium.com/deep-q-network-with-pytorch-d1ca6f40bfda, 2019.

Xiong, Z., Liu, X.-Y., Zhong, S., Yang, H., and Walid, A. Practical deep reinforcement learning approach for stock trading, 2018.

Yang, H., Liu, X.-Y., Zhong, S., and Anwar Walid year=2020, h. . h. Deep reinforcement learning for automated stock trading: An ensemble strategy.

Zhang, J. Reinforcement learning — generalisation in continuous state space. https://towardsdatascience.com/reinforcement-learning-generalisation-in-continuous-state-space-df943b04ebfa, 2019.