Recurrent Deep Reinforcement Learning with Applications to Financial Trading

Kylie Ying and Anubhav Guha
Massachusetts Institute of Technology

Abstract

In this project we implement both recurrent and standard feedforward PPO algorithms, and compare their performance on two standard control environments. We develop a stock trading environment and train a recurrent PPO agent to maximize return on a basic stock trading task. We find mixed results - in some years the trained policy outperforms the portfolio constituents, while in other years it underperforms. We recommend that more testing, training and development is needed to produce a viable and robust policy that can be used for live stock trading.

1 Introduction

Feedforward neural network architectures are commonly used in deep reinforcement learning algorithms. However, when observations are sequentially ordered in time, simple feedforward networks may be insufficient to capture the temporal nature of data. In these instances, it may be more advantageous to use recurrent architectures that can parse sequences effectively. Recurrent neural nets have the advantage that they were designed to learn sequential or time-varying patterns [1]. In this project, we explore the use of feedforward neural nets and recurrent neural nets in two physics-based environments, where we have modified the observations to be solely positional data over a certain number of timesteps.

One particular example of temporal data is stock market data. Financial data used to make trading decisions is often in the form of time series. In this project our goal is to train a deep reinforcement learning (RL) agent to trade a small portfolio of stocks over a given time period. Given current and historical measurements (of variables like equity prices, moving averages, trade volume, etc.) of the portfolio stocks, the goal is to learn a policy that can make decisions on which stocks to buy, sell, or hold in order to maximize profit. Furthermore, the agent is resource limited (finite balance/account value), so the problem amounts to a portfolio optimization task.

In this paper, we present results demonstrating that recurrent deep reinforcement learning policies outperform feedforward policies when applied to a modified Cartpole environment, a modified Half Cheetah environment, and a stock trading environment for a small selection of stocks. Furthermore, we present a trading agent that successfully outperforms buy-and-hold strategies for each individual stock during out-of-sample periods of "normal" stock market behavior (2017-2019), but fails to do so in periods of extreme volatility and unpredictable events (2020-2021).

2 Related Work

Recent works have studied the application of reinforcement learning to stock trading. Xiong et al. applied DDPG on Dow Jones (DJIA) stocks and found better results compared to both a min-variance trading strategy and a buy-and-hold DJIA strategy [2]. Théate et al. introduced a new algorithm, the Trading Deep Q-Network (TDQN) algorithm, which attempts to maximize the Sharpe ratio of a portfolio by determining the optimal trading position at any point in time [3]. Furthermore, previous

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

work involving recurrent neural architecture has been applied to finance via direct reinforcement [4]. The utilization of RNNs has also been assessed in DQN algorithms applied to non-financial, sequentially-observed real world problems [5].

3 Methods

3.1 Proximal Policy Optimization

We develop two Proximal Policy Optimization (PPO) implementations - one with a purely feedforward architecture, and the other with a recurrent architecture. The actor/critic architectures were inspired by [6]. For the recurrent network, a sequence of observations is passed through a gated recurrent unit (GRU), and the final output is fed into a small fully connected head, which produces an output for the actor or critic. The recurrent architecture is shown in Figure 1. In order to compare the recurrent architecture to the feedforward architecture, we also include the ability to switch the GRU layers with fully connected feedforward layers. The linear output of this feedforward net acts as the intermediate input into the fully connected head that produces the final output, just as in the recurrent model. It is also worth noting that the dimensionality of the output equals the dimension of the action space for the actor and is a single dimension for the critic.

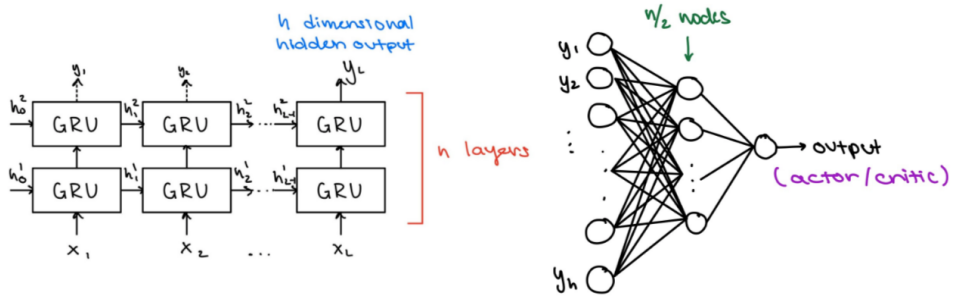


Figure 1: Actor/critic architecture for the recurrent PPO agent. The GRU model inputs each observation at a timestep into the GRU cells. The recurrent network can be transformed into a feedforward model that takes all the inputs at once, by replacing the left GRU network with a fully connected feedforward network. Both produce an intermediate output that feeds into the same head network to produce the final actor/critic values.

In addition, our implementation includes a rollout buffer that simulates a user-specified number of rollouts of a given policy. This rollout buffer samples the given actions and steps in the environment according to the action, and stores values, such as observations, actions, values, and rewards. Each rollout is specified by a constant number of timesteps in the environment. We sample observations, actions, and other associated values from this buffer during the PPO training. The PPO implementation is based on [7].

3.2 Stock Trading Environment

3.2.1 Assumptions

We formulate a stock trading/portfolio optimization problem as a reinforcement learning task. Specifically, given a set of m stocks, the objective of the agent is to trade shares of these stocks so that the cumulative net gain in portfolio value is maximized. The following assumptions are made:

- A1: Trades are made once per day at market open.
- A2: Buy & sell actions occur at market open prices.
- A3: Trade friction is less than $.04\%$ of the total trade volume, in dollars.
- A4: Trades at arbitrarily fractional levels can be made.

For many stocks Assumption A2 may not be realistic - for example, if the bid-ask spread is large, it may be infeasible to place trades at the stated open price. For stocks trading at extremely low volumes it may also be impossible to find buyers/sellers at the exact time of market open. For this reason, we

choose to only trade stocks that are highly weighted in the S&P index: by nature these equities are highly liquid and trade in high volumes, and therefore A2 is more likely to be satisfied. Related to this assumption is A3, which assumes that the total transactional friction of a trade amounts to no more than $.04\%$ of the trade value. This friction sums up broker fees, price differences incurred by the bid-ask spread, and any other fees that may accumulate. Given that we choose to trade highly liquid large-cap stocks, this is a reasonable assumption: for such stocks the bid-ask spread is generally low (indeed, many financial trading backtesters assume that stated open/close prices can be traded on). Furthermore, many brokerages (such as Alpaca [8], which can be used for algorithmic trading) do not have fees. A4 assumes that x shares of a stock can be traded, where $x \in \mathbb{R}^+$. While most brokerages allow for trading fractional shares, the precision of these fractions is capped. In practice, this would require rounding of our agent's trade decisions to the nearest available fraction and would possibly reduce the agent's edge/return *slightly*. We posit, however, that the magnitude of this effect is small and can be ignored.

3.2.2 Data Aggregation

We trade shares of the following four companies: Home Depot, J.P. Morgan, Johnson & Johnson, and Disney, with tickers HD, JPM, JNJ, and DIS respectively. These stocks were chosen because they are highly weighted in the S&P and are liquid enough to satisfy the required assumptions. Notably, we did not include any stocks from the technical industry, such as Apple. The historical data used for training is from the years 2000-2017, in which tech companies such as Apple experienced massive growth and dominated the market. As a result, the returns generated by these tech companies eclipsed the returns of companies in other sectors by such a large margin that any agent we trained on a portfolio that contained a large tech stock learned to just buy and hold that tech stock. This appears to be a locally optimal policy that was difficult to escape.

We use the `yfinance` Python library to scrape daily stock price data from 2000-2021. This is a relatively small amount of data (roughly 5000 days worth of trading data) and likely insufficient for reinforcement learning purposes. In [9] a similar problem was encountered, in which there was not enough real-world data to train an RL policy for a stratospheric balloon. To overcome this issue, the researchers in [9] used procedural noise to perturb the data and create an effectively infinite set of realistic data. We adopt a similar approach here. Referring to the stock price at a day t as p_t, the daily change percentages are calculated as $c_t = p_{t+1}/p_t$. To generate a synthetic time-series, we calculate new percent changes $c'_t = fc_t$, where $f \sim \mathcal{U}(1-\epsilon, 1+\epsilon)$ for a small ϵ. The new synthetic price data can be calculated: $p'_0 = p_0$ and $p'_t = c'_{t-1}p'_{t-1}$ for $t > 0$. This process is applied to each of the relevant price quantities (open, close, high, low) for each stock in the portfolio, and can be used to produce an arbitrarily large number of synthetic datasets. For each of these synthetic datasets, technical indicators are calculated using the `stockstats` Python library. Technical indicators use the price history of a stock to generate numerical quantities that may be used to predict future price movement [10, 11, 12]. In this project, we use 14 technical indicators, including the popular Moving Average Convergence Divergence (MACD), Bollinger Bands, and Relative Strength Index (RSI).

3.2.3 Environment

Only the first 17 years of data (synthetic or otherwise) is used for training. A gym-style environment was created from scratch, in which days are stepped through appropriately and the flow of information respects the time ordering of the stock data (similar to in backtesting software). At each reset of the environment a synthetic dataset is chosen at random. Observations consist of the most recent observed percent change in stock price: at day t the agent sees c'_{t-2}. We chose to use this percent change rather as opposed to absolute price in order to reduce the dependence of the policy on specific value of stocks. Given that large-cap stock prices tend to increase monotonically, we wanted to avoid a case in which the actor network memorized specific stock values. In addition to the percent change data, the observation vector consists of the technical indicator data for each stock. Lastly, the observation vector also contains the current portfolio allocations and cash balance.

The action is a vector of length $m + 1$, where m is the number of stocks being traded (4, in our experiment). The action vector consists of all positive entries which sum to 1 - these constraints are enforced by applying a softmax to the output of the actor network. The action represents the percentage of the agent's portfolio that is to be allocated to each stock, with the extra dimension representing the percentage that should be kept in cash. For example, if the action a at time $t-1$ was

$a_{t-1} = [.5, 0.0, 0.0, 0.0, .5]$ and the action at t is $a_t = [.3, 0.2, 0.2, 0.2, .1]$ then at time t 80% of the agent's cash-on-hand will be used to purchase stocks $2-4$, and 40% of the agent's holdings in stock 1 will be sold to purchase the remaining desired shares of stocks $2-4$. This action space formulation appears to be unique in the literature, and we believe it to be preferable to other methods, such as in [2], that require the use of a variable action space. Given the volume of buy/sell actions (in the example above, this volume amounts to 80% of the portfolio value), a frictional penalty is calculated using the maximum assumed friction. The reward is then given as the change in portfolio value after stepping the environment into the next day minus the frictional penalty. Each episode consists of 30 trading days, and each observation contains 14 days of lookback return data.

4 Experiments

In this section, we compare the feedforward and recurrent PPO models. The aim of the recurrent model is to better encapsulate the temporal nature of data into the actor/critic. Therefore, we consider two simple examples, Cartpole and Half Cheetah, before applying the algorithms to the stock trading task.

4.1 Cartpole

We developed a modified Cartpole environment [13], in which continuous control actions between -1 and $+1$ are accepted. We also modify the agent observations: traditionally the environment includes both cartesian/angular position and cartesian/angular velocity measurements $(x_t, \theta_t, \dot{x}_t, \dot{\theta}_t)$. In order to challenge the agent to learn temporal relations, we instead provide the agent with the past 3 positional datapoints: $(x_{t-2}, \theta_{t-2}, x_{t-1}, \theta_{t-1}, x_t, \theta_t)$. We set the time horizon to be $T = 200$ frames for each episode, $\lambda = 0.95$, and $\gamma = 0.99$. For our model, we experiment with both $n = 1$ and $n = 2$ layers for the feedforward and recurrent models. The hidden output size of the recurrent model and the number of neurons in the feedforward fully connected layers are set to 32. We use tanh activations in the network and actor output layer, and linear activations for the critic output. We set the learning rate to 5e-4 and train using the Adam optimizer. The results of the feedforward and recurrent architectures for both single and double layer models are shown in Figure 2. In both cases, we can see that the recurrent architecture converges to a slightly higher reward at the end of the training period, and generally attains a higher reward faster, as in the single layer case, or with less variance, as in the double layer case, when compared to the feedforward architecture.

(a) Single layer of fully connected/GRU units (b) Double layer of fully connected/GRU units

Figure 2: Smooth reward of modified Cartpole experiment, averaged across 5 random starting seeds, comparing results of feedforward and recurrent architectures. In both the single layer network and the double layer network, the recurrent model outperforms the feedforward model.

4.2 Half Cheetah

As a second test, we evaluate the recurrent and feedforward architectures on a modified Half Cheetah [13] environment. In this environment, we make similar observation modifications. The traditional observations in the original environment include the longitudinal position of the cheetah, the angle of the body, the 6 angles of the joints (thigh, shin, and foot for the front and back

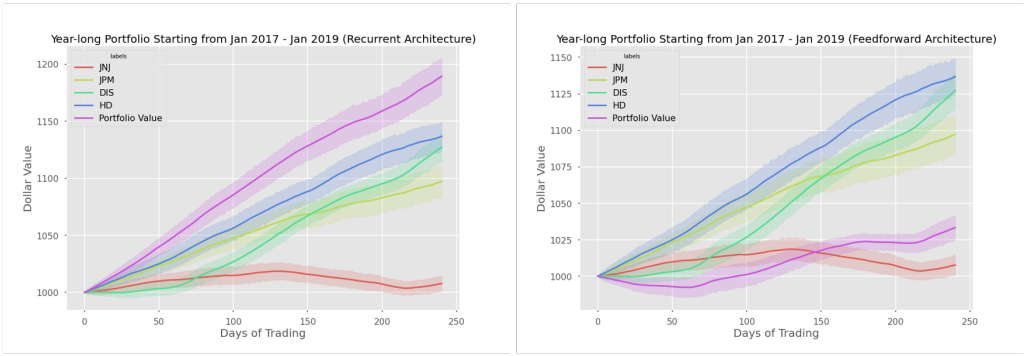
leg), and the 6 corresponding joint velocities. Once again, in our modified environment, we remove the velocity information and instead include the position vectors from the latest 3 timesteps: $(x_{t-2}, \theta^y_{t-2}, \theta^{(1\ldots6)}_{t-2}, x_{t-1}, \theta^y_{t-1}, \theta^{(1\ldots6)}_{t-1}, x_t, \theta^y_t, \theta^{(1\ldots6)}_t)$ where $\theta^{(1\ldots6)}_t$ is shorthand to represent the six angles. We set the time horizon to be $T = 300$ frames for each episode. The other parameters and setup we keep the same as in Section 4.1. The results of the feedforward and recurrent architectures for both single and double layer models are shown in Figure 3. In both scenarios, the recurrent network outperforms the feedforward network, by converging to a higher reward on average.

(a) Single layer of fully connected/GRU units (b) Double layer of fully connected/GRU units

Figure 3: Smooth reward of modified Half Cheetah experiment, averaged across 5 random starting seeds. As before, in both the single layer network and the double layer network, the recurrent model outperforms the feedforward model.

4.3 Stock Trading

We next trained both the feedforward and recurrent PPO methods on the stock trading task described in Section 3.2, with each algorithm training for 2.1 million frames. Figure 4 compares the test

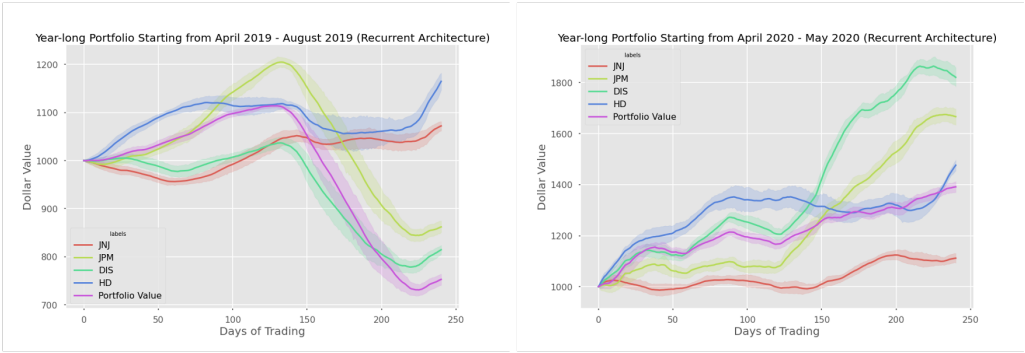


(a) Recurrent Policy (b) Feedforward Policy

Figure 4: Yearly returns of the recurrent and feedforward policies, as compared to the yearly returns of each individual stock in the portfolio. Portfolio balance and stock prices are normalized to have a starting value of 1000\$. For each valid day in the given trading range, the environment is started and run forward for 243 trading days, which provides nearly a year of trading return data. The next day in the trading range is then chosen, and a trading period of 243 days is evaluated. This process repeats for all days and provides a distribution of returns as a function of days-of-trading; the mean and 2σ of these distributions are plotted. In 4a, the trained policy achieves an averaged annualized rate of return of 19%, while the average portfolio achieves a rate of 9%. Over the same time period, the S&P gained at an annualized rate of 12%. The policy in 4a had an average Sharpe ratio of 1.034, while the average Sharpe ratio of the stocks in the portfolio was 0.52. A Sharpe ratio of 1 is generally considered "good" but not excellent [14].

performances of the feedforward and recurrent agents when trading between 2017 and 2020. As suspected at the outset of the project, a recurrent architecture is able to outperform the basic feedforward algorithm. Moreover, from the period between 2017 and 2020 studied in Figure 4a, we see that the

recurrent policy leads to positive results - out performing all the stocks in the portfolio as well as the S&P. While this is a promising finding, many more tests are needed to explore if this is a fluke or if the performance is repeatable.



(a) Recurrent Policy Starting in Mid 2019 (b) Recurrent Policy Starting in Mid 2020

Figure 5: Recurrent policy performance for time periods starting in 2019 onwards. Refer to Figure 4 for details on plot generation. Overall, the policy performs abysmally over these time periods and even underperforms every stock in the portfolio for the year-long periods starting in mid 2019.

Figure 5 indicates that these positive results do not hold past early 2020. We propose a few possible explanations. One centers on the fact that early 2020 marked a calamitous event for the world-wide markets - starting in late February, the advent of the SARS-CoV-2 virus and the resulting global pandemic shuttered the economies of nearly every nation and caused a rapid and deep global recession. Although the recovery was substantial and quick, we suspect that the dynamics of the market from 2020 onwards were substantially changed. This is a hallmark of a non-stationary environment - many reinforcement learning algorithms (including PPO) are not well-equipped to handle such a distributional shift without additional machinery. As seen in Figure 5a the RL trading performance is the worst, relative to the portfolio, soon after the market crash, while in 5b the performance is somewhat improved as the market and relevant equities recover. Another potential explanation is that distributional shift would have occurred with or without SARS-CoV-2 and the market crash. As the test set gets farther away (temporally) from the training set, it is likely that the distributional shift grows larger and the policy trained on a test set will generalize more poorly (another example of non-stationarity).

5 Discussion & Conclusion

In this project, we designed and implemented a recurrent PPO algorithm from scratch, and demonstrated it's efficacy on two toy problems as well as on a real-world stock trading task. We additionally developed a data-processing pipeline and trading environment that can be used for policy training and backtesting. We had mixed results in applying the trained trading policy to test data from 2017-2021, finding that the policy could outperform portfolio averaging techniques, buy-and-hold strategies, and the S&P over certain time ranges (2017-2020), but was unable to repeat this performance past February of 2020. Given limited resources and time constraints, we were unable to perform a thorough analysis of the recurrent algorithm trained for the stock trading task. While our preliminary results indicate that the trained policy *might* provide some edge in the right circumstances, significantly more tests would be needed. For example, at least 5-10 separate policies should be trained with different random seeds, and a larger variety of stock portfolios containing different companies should be tested. Developing reinforcement learning algorithms that can more effectively deal with the non-stationarity of the stock-trading environment is a direction for future research. Additionally, we would like to explore the use of more advanced state/observation representations - for example, using Twitter feeds and web-scrapers to construct market sentiment representations.

References

[1] Lakhmi C. Jain and Larry Medsker. *Recurrent neural networks: design and applications*. CRC Press, 2000.

[2] Zhuoran Xiong, Xiao-Yang Liu, Shan Zhong, Hongyang Yang, and Anwar Walid. Practical deep reinforcement learning approach for stock trading, 2018.

[3] Thibaut Théate and Damien Ernst. An application of deep reinforcement learning to algorithmic trading, 2020.

[4] J. Moody and M. Saffell. Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12(4):875–889, 2001.

[5] Xiujun Li, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He. Recurrent reinforcement learning: A hybrid approach, 2015.

[6] Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[8] Alpaca trading platform. `https://alpaca.markets/`.

[9] Marc G Bellemare, Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C Machado, Subhodeep Moitra, Sameera S Ponda, and Ziyu Wang. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588(7836):77–82, 2020.

[10] Christopher J Neely, David E Rapach, Jun Tu, and Guofu Zhou. Forecasting the equity risk premium: the role of technical indicators. *Management science*, 60(7):1772–1791, 2014.

[11] Yuzheng Zhai, Arthur Hsu, and Saman K Halgamuge. Combining news and technical indicators in daily stock price trends prediction. In *International symposium on neural networks*, pages 1087–1096. Springer, 2007.

[12] Michael AH Dempster, Tom W Payne, Yazann Romahi, and Giles WP Thompson. Computational learning techniques for intraday fx trading using popular technical indicators. *IEEE Transactions on neural networks*, 12(4):744–754, 2001.

[13] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[14] Arun Muralidhar. The sharpe ratio revisited: What it really tells us. *Journal of Performance Measurement*, 19(3), 2015.