DRL for Automated Portfolio Management

Exploring reinforcement learning agents for day-to-day stock trading

Adam Wang, Andrew Yuan, Sean Zhan, Paul Zhou
TA: Antony Sagayaraj

CSCI 1470 Fall 2020

INTRODUCTION

Profitable and reliable automated stock trading strategies are vital to investment companies and hedge funds, who manage the financial assets of millions of people. These strategies optimize capital allocation in order to maximize investment performance. Performance can be measured on the estimates of potential return and risk. However, it is challenging to design a profitable strategy in a complex and dynamic stock market.

We propose a deep reinforcement learning (DRL) solution to the algorithmic trading problem of determining the optimal trading position at any point in time during a trading period in the stock market. We designed an actor-critic policy gradient Reinforcement Learning (RL) agent, as well as designing a stock trading environment in which to manage portfolios of multiple stocks.

There has been some amount of previous work in this area. Wu et al. [1]proposed a Deep Q Network to perform stock trading which outperforms the turtle trading policy. Others[234] have experimented with combining different types of reinforcement models. However, most of these models have the limitation of only managing one stock at a time. We propose an DRL agent that is able to manage multiple stocks at the same time and make a buy/sell/hold decision at each time step. By managing different stocks at the same time, the agent should be able to learn across stock trends and make more informed decisions on which stocks to buy or sell.

Our code base can be found at: https://github.com/seanzhan0319/DRL-Stock-Trading.git

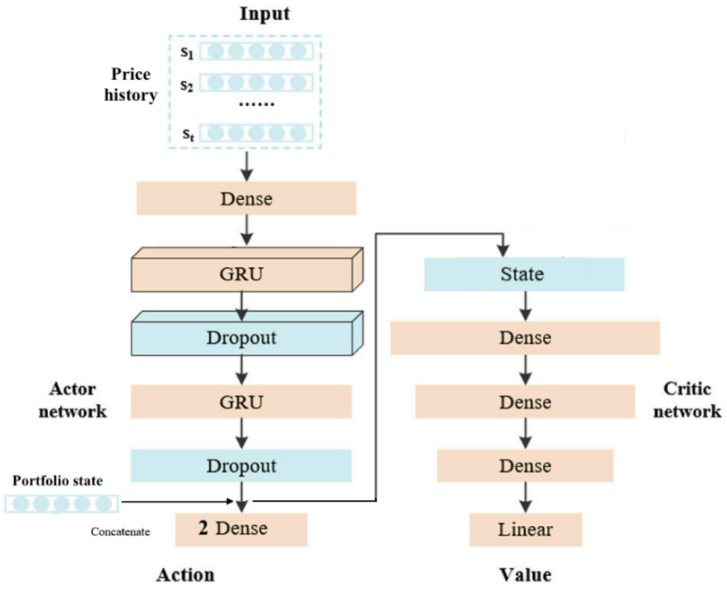
[1]Wu, Xing. Adaptive stock trading strategies with deep reinforcement learning methods. 2020.
[2] Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy
[3] Learning Financial Asset-Specific Trading Rules via Deep Reinforcement Learning
[4] Application of Deep Q-Network in Portfolio Management

METHODOLOGY

Data preprocessing: We import data from the yfinance Python package, which offers well-structured day-to-day ticker data from Yahoo Finance. We chose the timeline of 2010.1.1-2020.1.1, where there are no major volatilities in the stock market (e.g. major unpredictable crashes). And we choose AAPL, AMZN, GOOGL, MSFT as our training stocks and seen testing stocks, and REGN, WMT, JNJ, HON as unseen testing stocks. For each trading day (~261 per year), we import a data point consisting of open, high, low, and closing price points, as well as volume of trade, for each of the stocks in our portfolio.

Stock Environment: We have created a stock/portfolio environment that handles transaction fees, takes into account interest, borrowing, and inflation rates, and allows the agent to sample episodes of experience from historical financial data according to the current policy. In generating a trading episode, the environment initializes a portfolio containing no stock shares and some amount of initial cash (e.g. $1000). For each timestep of the episode, we sample an action (hold, buy, or sell) for each of the stocks in our portfolio and handle the exchange of stock shares accordingly. We chose to use a fixed cash amount (e.g. $100) to be used in buy and sell transactions. Our portfolio values are then recalculated in the next timestep using the change in each stock's closing price, before determining the next action to take. Note that our portfolio allows negative shares and negative cash, which emulates stock shorting and trading on margin (with appropriate fees implemented). However, if the total portfolio cash value becomes negative, the episode will terminate prematurely.

RL agent: Our main model consists of an actor-critic policy gradient RL agent using a lifting layer, 2 GRU layers and 2 Dense layers as the actor, and 3 Dense layers as the critic. The agent is able to perform a forward pass on a batch of data and calculate the loss on that using standard actor-critic loss method. It also uses experience replay to stabilize training. We choose the following representations for our state, action, and reward spaces:



- State: The state space consists of two components:
    - Pricing state: This state contains the pricing data (open, high, low, close, volume) for each stock in our portfolio for the last X timesteps (e.g. X=50). This is the input into our actor's 2 GRU layers.
    - Portfolio state: This state contains the cash amount invested in each stock of our portfolio, as well as the cash on hand at the current timestep. This is concatenated with the output of our actor's 2 GRU layers, and is then passed through the remaining dense layers of the actor.
- Action: The action space at a given time step consists of a probability distribution for each possible action (hold, buy, sell), for each stock in our portfolio (i.e. a 3 by <num_stocks> tensor). During training, we sample an action from each stock's distribution to determine the action for a given timestep. During testing, we take the max-probability action.
- Reward: The reward is calculated as the discounted sum over each timestep's total portfolio cash value. We also experimented with using each timestep's change in total portfolio cash value, but this produced unstable results. Given more time, we would want to experiment with a weighted average of the two to balance short-term and long-term profit-seeking.

Training, Validation, and Testing: We used a 8:1:1 split for our training, validation, and testing data. Due to our relatively limited stock data, we implemented two training methods:

- The first method trains on data from a fixed set of stocks *S*, validates on *S* (but different time period), and tests on *S* (again, a different time period) as well as on a different set of stocks *T* (using data from the same time period as *S*). The reason for testing on a set of different stocks is to see whether or not our model was able to learn the specific interdependent price behaviors of the stocks in *S*. For example, *S* could include 4 large-cap technology stocks. We would expect the model to learn how prices between the stocks in *S* may be correlated; however, the model should then perform differently (almost always worse) on *T*, since the model's learnings of the interdependent price behavior of *S* does not apply to *T*.
- The second method trains on data from a dynamic set of stocks *S** taken from a larger pool of stocks *S*, where *S** is resampled from *S* for each episode run. Having a large pool of stocks *S* gives us much more training data to work with and produces a more general trading model, as opposed to the first method. The model validates on *S** and tests on *S** as well as a different set of stocks *T*, which is disjoint from *S*. Here, we cannot learn interdependent price behavior since our *S** changes from episode to episode, so we would expect the testing results on *S** to be similar to the results on *T*.

RNN to predict stock price changes: we engineered a RNN that's designed to predict changes in stock prices at the current time step given a sequence of price histories for a

particular stock. This model mainly consists of a GRU and several Dense layers, and is trained independently of the main RL agent using supervised learning and data from 12 diverse stocks over a span of 10 years. It is designed that the second to last Dense layer of this RNN model should be treated as an embedding space and fed into the RL agent as input in order for the RL agent to make more informed decisions. However, the testing accuracy on this RNN is around 100% in Mean Absolute Percentage Error and doesn't perform well, so it is eventually not used in our main model.

RESULTS

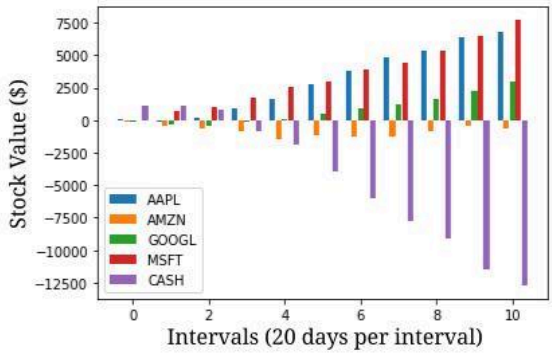
All agents start with $1000 in cash and test for 200 days	Final portfolio value[5] (testing on the **same** set of stocks as training: AAPL, AMZN, GOOGL, & MSFT)	Final portfolio value (testing on set of stocks **different** from training: REGN, WMT, JNJ, & HON)
Portfolio Agent (4 stocks)	**$ 4170.20**	**$ 2342.88**
Single Stock Agent (the combination of 4 portfolio agents with $250 and trading 1 stock each)	$ 1129.22	$ 685.37
Buy and Hold (4 stocks)	$ 1308.24 (benchmark)	$ 1098.47 (benchmark)

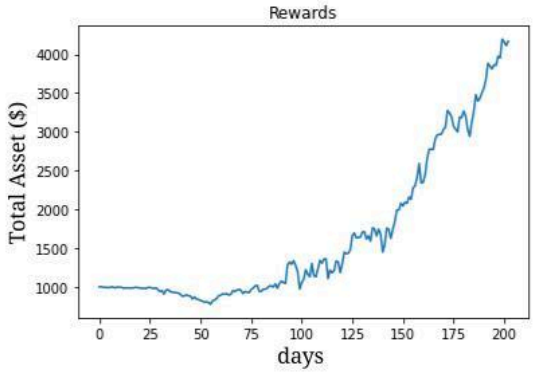
[5] Portfolio value here is defined as the combined value of the cash at hand and the current value of all stocks bought, subtracting the debt the agent holds.

Buy and Hold Agent

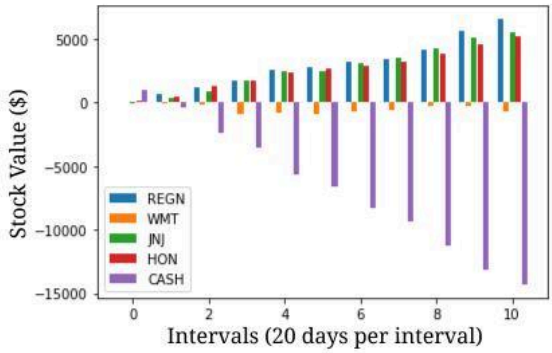
Portfolio Agent

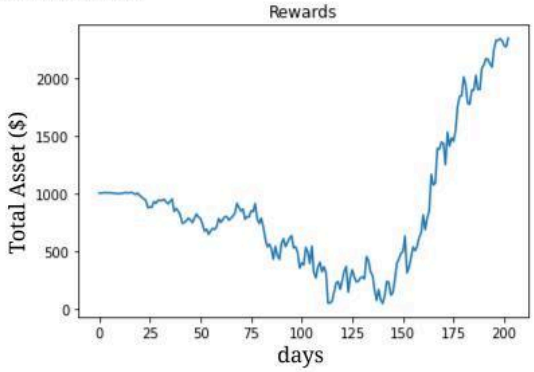
Testing on the same stocks





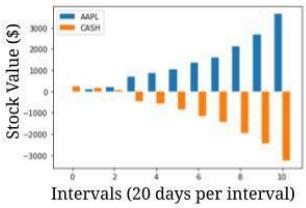
Testing on different stocks



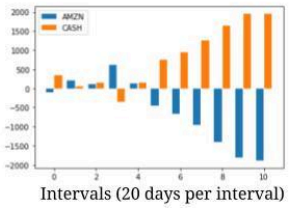


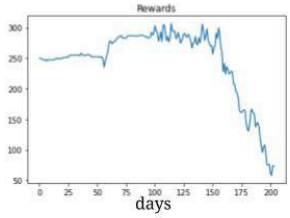
Single Stocks Agent

Apple

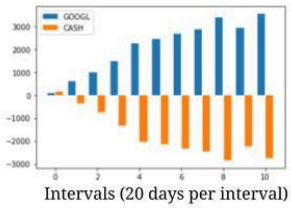


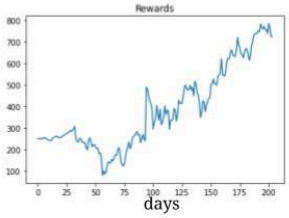
Amazon



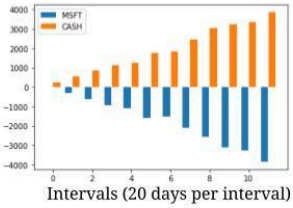


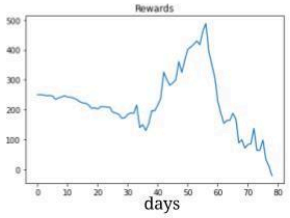
Google





Microsoft





1) Comparing the RL agent against several baselines:

We compare our model against the performance of two baseline agents

- **Portfolio agent**: our agent described in the previous section
- **Single stock agent**: another agent we trained using similar methodology but is only trained on the data of 1 stock and can only trade one stock. We inputted data for a single stock to the deep network and allowed the RL agent to trade this single stock. We repeated this for all the stocks and combined the earnings of the 4 agents.
- **Buy and hold agent**: this agent uses a deterministic trading policy. It invests all its cash evenly distributed in all the stocks the portfolio agent trades in on the first day, and maintains this position until the end of the experiment..

At test time, all agents were given $1000 to trade over ~200 trading days. Based on the experiment results, we make two important observations.

First, we note that our **Portfolio Agent performs about 220% better than the Buy and Hold Agent**. The Buy and Hold agent represents the growth of the stock market (a.k.a the market grew and made the initial $1000 into $1308 in around 200 trading days). This shows that our RL agent is able to actually make money by performing trade actions and not just taking advantage of market growth. The agent outperforming the Buy and Hold agent shows that it was able to learn the optimal policy given the general trend of the stock prices. As tech stock prices have risen tremendously in the past 10 years, the agent tended to trade on margin and borrow money in order to invest in the stocks, as seen by the largely negative amount of cash.

Second, we see that our **Portfolio Agent performs ~270% better than Single Stock Agent**. This result shows that trading a portfolio is indeed better than trading in single stocks. This is because when trading in portfolio, the agent has more information about whether a particular stock is worth buying/selling or not, compared to the other stocks.

2) Testing agent on unseen stocks

The experiment result shows that our agent is a lot more unstable when testing on stocks unseen during test time. The reward graph above shows a huge drop in the middle of the 200-day episode. However, the agent is able to save itself from that situation and recovers. In the end, the agent does not perform as well as when tested on the same training stocks. This shows that our agent is **able to learn stock-specific behavior, but can't generalize well** across stocks.

3) Lifting Layer

Another major factor affecting the performance of our agent is the lifting layer. Before passing price history through the two GRU layers, we first use a dense layer to lift the data to a higher dimension, much similar to what we have done in the “Neural Networks on Graphs” assignment. This lifting layer helps our agent learn about stock behavior with augmented data. Excluding this dense layer results in decreasing rewards and net loss in portfolio value.

4) Randomizing

To allow our portfolio agent to train on more data, we implemented a randomization method in which a set of stocks is sampled from a larger pool of stocks in each training episode. However, due to time constraints, we were unable to develop a profitable model using this training method. Given more time, we would look into developing a more robust and complex model so that it can take advantage of the larger amount of data.

5) Learning Rate

We also noted that the learning rate had a large impact on the behaviour of the agent. The above results were obtained using an exponential decay learning schedule (initial learning rate 0.03, decay factor 0.98, decay steps 10000). We also tested exponential decay learning schedule with initial learning rate at 0.01, 0.003, and the RL agent had a very similar performance.

CHALLENGES

Our main challenge was limited data. Because the price data of all stocks in our portfolio had to come from the same time period, we had to select stocks that had sufficiently-long price histories. Furthermore, since our model does not take into account factors such as news sentiment or overall market indices/indicators, we decided not to use data from before 2010 due to stock market events that our model would not be able to predict. This leaves us with only ~2600 data points for each stock, which is not enough data for such an intricate task (without overfitting). We tried several methods for overcoming this challenge, such as training on portfolios sampled from a larger pool of stocks, but this causes our model to lose information pertaining to interdependent price behavior specific to a given set of stocks. In the end, we found that we were most successful with models that trained on a portfolio of a specific set of stocks, despite the fact that our data was more limited.

We were challenged by the problem of setting up a realistic stock environment. We made the decision to simplify transactions by setting a fixed amount for buy/sell actions (e.g. $100); that way, our action space would be able to be discrete, and the agent would be focused on a simpler task. We also made the decision to allow for the borrowing of stocks and cash (i.e. negative shares and balances) so that our agent would have more room to

trade, especially since we had fixed the transaction amount. Given our portfolio design, we thought that it was important to include the portfolio's information (i.e. amount invested in each stock, as well as cash on hand) in the state space, so that the agent would have an idea of how much cash it had to spend, how many shares of stock X it was borrowing, etc.

As always, the design of the architecture of the RL model is a challenge. Based on the stock environment, we experimented with multiple designs and found a locally optimal one. The strategy that the agent has ultimately learned is borrowing cash and buying stocks because it learns that stock prices will generally rise over time. This trading method seems to be a brute-force approach to investment, and the agent seems to ignore short-term fluctuations in stocks. We have tried to adjust different parameters such as exponentially decaying learning rate, layers sizes, number of layers, training to validation to testing data set ratio, etc. However, our agent performs best only when it keeps borrowing cash and buying stocks, but maybe that's the best strategy to trade in the real world if we have a lot of money.

REFLECTION

How do you feel your project ultimately turned out? How did you do relative to your base/target/stretch goals?

We found that the task of tuning the model to produce reliable results was an intensive and interesting process. Though we do not feel that our model has been tuned perfectly (i.e. there is still some variance in the results between different models), we have been able to produce models that are profitable when tested over 200 trading days (roughly a year). Given how difficult the task of producing profitable agents is, we are proud of our work on this project and are satisfied with the results. We've surpassed our base goal of implementing a profitable agent with the capabilities of holding, buying, and selling single stocks, and we've met our target goal of implementing an agent that performs as well as/better than benchmarks (e.g. buy-and-hold) and stock indices (e.g. S&P 500) on a portfolio of multiple stocks. Due to time constraints, we have not yet implemented our stretch goal of options trading, though our model does support trading on margin and stock borrowing.

Did your model work out the way you expected it to?

Our model did work out the way we expected it to. Because it is a relatively small model, we did not expect it to be extremely consistent or profitable. However, we did expect that our model should beat the buy-and-hold benchmark, as well as the performance of stock indices. Though we were able to do so, we had hoped that our model would have less variability during training/testing, which is possibly a result of limited data.

How did your approach change over time? What kind of pivots did you make, if

any? Would you have done differently if you could do your project over again?

Our approach did not change much over time since we thoroughly discussed implementations before deciding on a direction to go. We did not make any significant pivots, although we made several improvements to our model by refactoring certain parts of our training and testing pipeline. For example, we added a randomization feature to our model which allows the model to sample training sets from a larger pool of stocks (to attempt to solve the problem of limited data), and we added informative validation logs and visualizations after each training epoch. If we could do our project over again, we would focus more on short-term trading as opposed to long-term portfolio balancing. By changing the action space to allow the agent to decide transaction quantities, and by tuning the reward function to better balance short-term and long-term profits, we would allow the agent to learn better strategies for short-term trading.

What do you think you can further improve on if you had more time?

We did notice that our project did not always produce models that were consistently profitable. It seems that the models did not always converge smoothly; however, our model trains in ~10 minutes (due to the relatively small amount of data available) so we were able to train multiple models and compare their performances. If we had more time, we would explore ways to increase our dataset either by generating artificial data (which is a challenging project on its own) or by improving our implementation of the method of training on samples from a pool of stocks (which we could not produce a profitable model from, partially due to time constraints). Given more data, we would be able to train larger models as well.

What are your biggest takeaways from this project/what did you learn?

One of our takeaways was the difficulty training deep learning models when there is not a lot of data and the data is noisy, as is the case for real-world stock prices. Through experimentation, we developed creative methods to augment our data such as using stock data from multiple stocks (i.e. portfolio trading) to stabilize results as well as randomizing the set of stocks to train on to create more data points. With the limited data we have, we set our goal accordingly by expecting the agent to learn portfolio-specific stock trading strategies. In general, we learned about developing realistic stock environments, designing states, actions, and rewards for DRL trading agents, implementing new methods such as experience replay, and visualizing the model's results in comparison with benchmarks.