Stock Trading with Reinforcement Learning Project Report

Xinlu Liu
ANLY-591

Abstract
In this project, we explored the application of reinforcement learning on stock price data. The goal is to train our RL agent to make stock trading decisions to gain positive returns over time. Following the sequence of data simulation, environment construction, and agent policy optimization, we built a basic framework for the application with various interchangeable entries involved. We used both simulated stock prices and the historical stock price from the market for the data. We examined different methods of defining the rewards for our environment. We used deep learning algorithms for the agents' policy as function approximators between the states and rewards. We also involved several reinforcement algorithms as rules for training and optimizing our RL agents, aiming to find the optimal way to update agent policy. We discussed the attempts we made for our applications along the process.

Keywords
Reinforcement learning, stock price, stock trading, policy optimization, deep learning

Introduction
Among all the popular applications of reinforcement learning in financial market, stock trading is surprising not one of them. But it seems fairly easy to implement what we have learned about reinforcement learning on this process. So, we want to try to set up the system and reflect on why it is not that widely applied. We follow the steps as data simulation, environment simulation, agent training for this project and we also develop a live agent training system at the end of the project as we find it could be potentially more beneficial. And we discuss how and why this application may not be so ideal as we proceed.

Method
We use historical and simulated data for the stock price data. Historical stock price data can be collected using numerous ways, but they are always subject to some limitations, especially for open-sourced data. For our agent to be an auto trading robot, it should be able to perform on a high frequency and need to be trained over a long period. But open-sourced data tends to have long intervals. For the same quantity of data in different intervals, their underlying function could differ from factors like seasonality. We test the stationarity of the Google stock price using the Dickey-Fuller test for 2000 records of daily data and 2000 hourly data count back from when we performed the test; the p-value for the daily data is over 0.6 while lower than 0.1 for hourly data.
To eliminate the constraints of historical data, we also generate the stock price based on their Markov property. The stock price only increases or decreases over time, so the current price depends on the past price. We also consider that the price has mean-reverting behavior1, as it tends to change following the direction of its mean. There are three ways to simulate stock price

data with these two rules: a logistic function, a sigmoid function, and a leaner function. The data generated using the sigmoid function is closer to the historical data1, so we use the sigmoid one.

To build our environment, we have several parameters. The state space is defined using the state shape of the records from the current past with a parameter called window size. The size can be 5 or 100, depending on the available total data. The action space contains the probability of taking three actions: hold, buy, and sell. We have a reply memory iterable to store the past trajectories, an inventory iterable to keep the current share held by the agent, as well as an iterable to store the cumulative returns. And we define gamma as the discount rate. We only used Epsilon-Greedy as our rule for balancing exploration and exploitation. We limit the epsilon's maximum and minimum and let it decay over training iterations. The reward is calculated in the training loop. There are several ways to calculate the rewards. One of the common ways online is using the positive return from making a sale and keeping the reward 0 when creating a hold, or buy, as well as when the return is negative, which is what we use in the code. But the reward can also be defined as the return as it is. It can also be defined as the last reward when the action is called to hold. We also use a profit variable to record the cumulative returns of all the transactions.

For the agent class, we define four functions, including the policy function, which stores the neural network, the action function, which calls the NN and outputs the action, the inventory function, which updates the inventory based on the action when being called; and the optimize function, which update the policy function under the exploration-exploitation constraints when called.

For each episode, the first state is the first timestamp of the data. For each iteration, we query a state value and a single record right after this state as the current price. Then we calculate the action using the policy function. We use an inventory variable to store the current price when taking a buy action, and the value stays the same till a sell made. When receiving a sell action, if the inventory is empty, the reward is 0 since we have nothing to trade; if the inventory is not empty, we can calculate the reward as we stated before. The next stage is the next window in our data.

Another fact about stock price data is that there is no absolute termination. As long as there is a data feed, the training can continue without returning to the first state. Since we have a limited volume of data, we have several ways to define the terminal condition. For simulated data, we can terminate when we run through all the records in one simulation and simulate a new data set with the same initial condition for the next episode. For the historical data, we can either run through the same set of data every episode or take slices of the data for different episodes. In this case, we need to either use the return of the stock price or the log value of the price to ensure stationarity.

We use a plain update for the learning method by calling the nn fit function on the same policy function and deep Q learning. We use two different neural network structures, both taking the state and output the possibilities for all actions. One is a simple linear neural network containing three linear layers with a relu activation function and one linear layer for outputting desired action array. The loss function is defined using the mean squared error; the optimizer is the Adam optimizer from the Pytorch infrastructure. The other nn contains one LSTM layer and

one linear output layer; the loss function is defined with mean absolute error, and the optimizer is RMSprop.
As our machine does not have the capacity to run cross-validation over the different methods, we proposed. We run the combinations with mix-matching parameters individually, so we have free RAM constantly. We use the total profit from each episode and the policy function's training loss to determine the methods' convergence and performance.
To overcome the limitation of the data, using our framework, we build a system that can train the agent with live market quotes, which enables us to collect data at a higher frequency. There are several APIs available in the Python environment that provides instant stock price quotes. For each episode, we quote a window-sized price for the initial state, then for each iteration, we generate the action using the current state. We then can quote the current market price, can calculate the reward. The new state is the window with the quote. Along the process, we add the trajectory to the reply memory and fit the model once the memory is full. This method is also subject to the limitation of free tier API, as they tend to seize the quoting when we reach certain constraints. But it should work fine if we use an API like Interactive Broker with subscriptions.

Results and Discussion

With generating new simulated data for each episode, most of our proposed methods produced similar results. The training loss converges at around 30 episodes. The total profits of each episode gradually increase after 20 episodes. The LSTM NN takes approximately 40 episodes to converge, but the total profit the LSTM model can reach is five times higher than the vanilla one.
One of the famous auto trading algorithms is to use the past five-time steps and assume the next step will follow the same trend in this time frame and make trading decisions. This process takes significantly less computing power and generally results well when used on a second-based interval. On the contrary, our auto trading bot needs significant training time, which may cause gaps and delays in utilization. This could be one of the reasons why RL is not broadly used in stock trading.
Considering the reinforcement learning process as finding the optimal function that takes the state value and outputs the action, it is easy to rationalize some of its typical applications, like balancing the weights of each equity in portfolio management and leveraging risk factors of option pricing. For stock trading, the function seems as easy as using the past stock price to predict the next stock price. The difference is that when we want our agent to make trading signals, the actions are discrete, while the more common usage is under the situation where both the states and the actions are continuous. Researchers may generally consider the better choice is to have the agent determine the number of shares to trade instead of acting on fixed amounts.

Conclusion

Since we do not compare our results to other machine learning algorithms, we cannot conclude that this algorithm is better. And as is the nature of the stock market, this type of project will be limited by the accessibility of stock price data. But as we present in previous chapters, it is programmable, and the environment has a lot of potential structures. The size and values used

for the state, the definition of the rewards, and the learning method can all be changed and fitted just as the policy function.

Reference

1. Rao, A., & Jelvis, T. (2022). *Foundations of Reinforcementn Learning with Apppications in Finance*. CRC Press.