Reinforcement Learning for Complex Financial Time Series Analysis

By

Omkar Ranadive

Table of Contents

Index	Topic	Page Number
1.	Abstract	2
2.	Introduction	3
3.	Related Work	4
4.	Market Data	5
5.	System Overview	6
6.	Environment	7
7.	Agent Design	13
8.	Experiments	16
9.	Results	18
10.	Conclusion	19
11.	Data and resources	19
12.	References	20

Committee Members:

- Han Liu
- David Demeter

Project Repository: https://github.com/Omkar-Ranadive/RL-Finance

Abstract:

We present an environment to process and analyze high-frequency trading data. High-frequency trading involves a large number of transactions happening within a fraction of a second, leading to a very complex financial time-series. Analyzing such data and developing an investment strategy to maximize returns in the high-frequency trading scenario involves complex decision making. While Reinforcement Learning (RL) has been used in the past to analyze normal stock data, there haven't been any significant advances in using RL to analyze such high-frequency data. Our environment provides the ability to process such data by building and maintaining a limit order book which is then passed on to RL agents for training and developing strategies. We develop and test the ability of RL agents to process such data and present results on their performance.

1. Introduction

In stock trading, the main goal is to develop an investment strategy to buy/sell stocks to maximize the profits. Traditionally, investment firms have used statistical techniques like calculating moving average and financial technical indicators like Relative Strength Index (RSI) and Commodity Channel Index (CSI) to develop investment strategies. However, the stock market can be thought of as a very complex process which cannot be modeled and predicted well enough by simply using these traditional techniques.

In recent years, Deep Learning (DL) has been widely used to model the stock market and predict the changes in stock prices. Deep Neural Networks are universal functional approximators and hence, they do a better job of modeling the stock market as compared to traditional techniques.

Reinforcement Learning techniques have also started to gain a lot of attention in the field of finance, as RL involves learning an optimal policy which maximizes rewards. This fits in very well with the financial framework, where learning an optimal policy can be thought of as learning an optimal investment strategy and maximizing rewards can be thought of as maximizing profits.

However, most of the literature has been focused on processing stock data on a scale of days/months/years. There are many reasons for this - Such stock data is readily and freely available over the internet and it is easy to process and deal with.

On the other hand, high-frequency data is not readily available to the public and requires massive computational power to process. But at the same time, the high-frequency data contains a plethora of information which if meaningfully processed can help develop profitable strategies.

This high-frequency data can be thought of as continuous time-series data where some trade takes place every time step (where the time step represents a fraction of a second). Therefore, Reinforcement Learning is a good fit for modeling such data as it is known to work well in model-free environments with complex environment dynamics like this.

In our work, we have created an environment to process the high-frequency data in an efficient way. This environment is wrapped around OpenAI's Gym interface so that it can be easily accessed by different RL agents. We train different RL agents and test their performance using various metrics like – Mean Reward, Sharpe Ratio and Hit Ratio.

2. Related Work

Most of the techniques in the literature use traditional statistics and technical indicators to form features in a meaningful way which are then passed to the Deep Learning model. Various Deep Learning architectures have been tested in the financial setting. *Dash et al., 2016* used a hybrid stock trading framework which integrates traditional technical analysis with Deep Neural Networks. *Chen et al., 2016* and *Seizer et al., 2018* use Deep Convolution Neural Networks to predict which action to take based on images. They form these images by using different technical indicators such that the values of these indicators correspond to a different pixel value.

Reinforcement Learning has also been used successfully by *Yang et al., 2020* and *Lee J.W, 2001* to predict stock movement. Reinforcement Learning has also recently been used for high frequency trading by *Briola et al., 2021* and *Rundo et al., 2019*.

3. Market Data

Level	OrderID	Time	BidSize	BidPrice	AskPrice	AskSize	Time	OrderID	Level
1	O101	12:02:36	2	$33.75	$33.77	2	12:03:25	O105	1
2	O104	12:03:18	1	$33.75	$33.78	3	12:03:11	O103	2
3	O102	12:03:07	5	$33.74					3
4									4
5									5

Figure 1: Example of limit order book, image courtesy of Richard Holowczak

We use market data from the IEX Trading Platform which can be downloaded from their website.

There are two types of market data available on their website:

- Depth of the book (DEEP): This data will reflect real time changes in the limit order book
- Top of the book (TOPS): This data reflects the Best Bid/Offer (BBO); i.e., the top of the book.

In our work, we use DEEP (Depth of the Book) data which contains high-frequency trading information. This high-frequency trading information is contained in the form of order book changes happening in real-time. These changes are provided in the data in the form of price-level updates. An order book stores limit orders which are reflected by the price-level updates. Limit orders are orders which are only executed if certain conditions are met. For example, consider the limit order book instance in figure 1. A separate order book exists for each stock and in each book, there are two sides – buyers and sellers. Whenever someone wants to buy a stock in the form of a limit order, they specify the maximum amount of price they are willing to pay and number of shares (volume) to buy. These orders are arranged descending order of bid prices

(highest to lowest). On the other side, whenever someone wants to sell a stock, they specify the minimum amount they are willing to sell it for and the number of shares they are willing to sell. These entries are arranged in ascending order of ask prices (lowest to highest). An order is executed only if the conditions are met on both sides.

Our environment maintains such an order book for every stock by processing the price level update information from the IEX data.

4. System overview

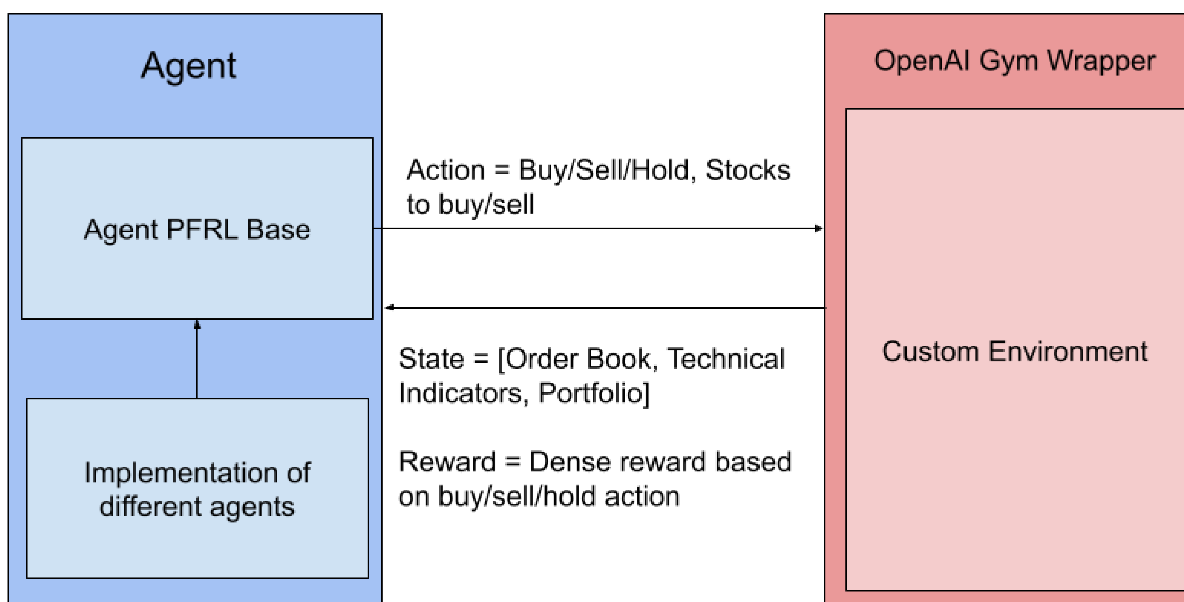


Figure 2: Overview of the RL framework

There are two major parts in our project as seen in figure 2. We have a custom environment which can process the limit order data. This environment is wrapped around OpenAI's Gym wrapper so that it can be easily used by RL agents.

At the agents' side, we use the PFRL library (*Fujita et al., 2019*) to develop and train the agents. There is a PFRL base class which takes in any custom-built agent built using PFRL functions and then using that agent the class interacts with our custom environment.

5. Environment

5.1 Building the order book

Like mentioned in the previous section, the order book is built by processing the price level update messages. We process these price-level update messages using iex_parser module. The end user can provide a list of stock to limit the messages only w.r.t those stocks. The IEX Trading platform has 430 stocks trading in a day, which is a big number and RL agents would have a difficult time predicting over the entire list of stocks. Hence, by providing a list of stocks, the complexity of the environment gets reduced. For our experiments, we chose the top 50 most traded stocks out of the 430. The top n most traded stocks can be dynamically calculated for any DEEP data file using the function which we provide in data_utils file.

The order book in our environment is of the following form (key -> value dictionary pair) :

$$stock\ name \rightarrow [[bid\ book], [ask\ book], \{volume\ info\ bid\}, \{volume\ info\ ask\}$$

The bid book and ask book lists contain the price information. The prices are sorted using bisect module. We maintain separate dictionaries to store volume information for fast access. These volume info dictionaries are of the form: price -> number of shares.

The price level update message from the parser gives us the stock name, side (Buyer/seller) and available volume. Based on this info, we perform order book update. When the available volume is zero, we delete the entry from the order book, i.e., we assume that the trade got executed.

Algorithm:

```
If new stock:
        Initialize entry in order book
                Order_book[stock] = [[], [], {}, {}]

If Buyers side:
        Add entry to the buyers list using bisect module and update available vol
        info for it

        If available vol == 0:
                Pop last entry (highest bid price)

If Sellers side:
        Add entry to order sellers list using bisect module and update available vol
        info for it

        If available vol == 0:
                Pop first entry (lowest ask price)

Note: When available vol == 0, then orders corresponding to the highest bid and
lowest ask should be traded but we found that there are rare cases when this is not
the case. This can be attributed to the fact that buyer/seller may retract their
order which may lead to available vol == 0 condition; hence, our implementation also
checks for this case
```

Limitations:

- As we are only getting the price level update information, we cannot distinguish between two orders if they have the exact same price value. We simply assume same that the same price value belongs to the same order and only update the volume information for that price level.

- The order book cannot take in external limit order; i.e., from the RL agent as this would disrupt the follow of the entries being parsed from the IEX data. Hence, we assume that the agent can only place market orders.

5.2 State Representation

The state is made of three parts – Order Book Array, Feature Array and Portfolio Array.

- **Order Book array:** The order book array contains the top n entries from the order book for each stock. For our experiments, we choose n = 10. Empty values are denoted using -1.
- **Feature array:** Five features are calculated for each stock at each time step. They are as follows:
    - **Spread:** It is calculated as the difference between highest bid price and lowest ask price
    - **Total volume at buyers' side:** This is calculated by summing over all volume entries on the buyers' side for a stock
    - **Total volume at sellers' side:** This is calculated by summing over all volume entries on the sellers' side for a stock
    - **Mid-price:** The mid-price gives a unique single price to a stock. This is calculated as the average of the highest bid and lowest ask price.

Let P^b denote the highest bid price and P^a denote the lowest ask price. Then, mid-price at time step t is as follows:

$$\text{Midprice}_t = \frac{P_t^a + P_t^b}{2}.$$

- **Micro Price:** Micro-price is similar to mid-price but it also weights the average by the volume of the highest bid and lowest ask price. Let V^b be the volume of shares at highest bid and V^a be the volume of shares at lowest ask. Then micro-price at time step t is given as follows:

$$\text{Microprice}_t = \frac{V_t^b P_t^a + V_t^a P_t^b}{V_t^b + V_t^a},$$

- **Portfolio Array:** The portfolio array maintains three pieces of information for each stock – Number of shares bought, total amount spent on buying those shares and the value of the bought shares at current time step t.

The values in order book array and feature array are normalized between -1 and 1. Portfolio array values are normalized between 0 and 1. These three arrays are then combined into a list and returned to the agent at each time step.

5.3 Action Space

The action space of the environment is discrete and can take one of the three values. They are as follows: Buy (0), Sell (1), Hold (2).

Along with these three discrete actions, the agent will also output a stock matrix indicating which of the n stocks to buy (in our experiments, n = 50). So, this stock matrix will 50 entries where each entry is either 1 or 0. Each action works as follows:

- **Buy (0):** We take the stock matrix outputted by the agent and pass it through a liquidity checker which ensures that the stock which the agents want to buy are actually available at present, i.e., there is at least one sell entry in the order book for that stock. In case a stock is not available, it is set to 0 in the stock matrix. After passing it through the liquidity checker, the stock is purchased and added to the agent's portfolio. We only allow one share to be purchased per share per time step to simplify the environment. Fractional shares cannot be purchased as of now, as that too would increase the complexity of the environment.
- **Sell (1):** Similar to the buy case, we ensure using the liquidity checker that there is actually someone to buy the stock, i.e., there is at least one buy entry in the order book. We assume that the agent wants to sell all of the stocks to simply the environment. The stocks are sold at the highest bid price based on the current order book.
- **Hold (2):** This action implies that the agent does not wish to either buy or sell for that time step.

5.4 Reward Design

Most of the literature only uses profit/loss as the reward signal. To incorporate a denser reward structure, we also calculate rewards in the buy/hold scenarios. The reward structure for each case is as follows:

- **Buy (0):** To check whether buying a stock is a good or bad move, we apply a lookahead on the order book; i.e., how will the order book look f time steps ahead. We use f = 50 for our experiments. So, if after f time steps ahead, the buy price has gone done for a stock, then buying it right now was a bad idea, if it has gone up, then buying it at present is a good idea. The reward is calculated as follows:

  $$M_f - M_c$$

  Where M_f is the mean cost to buy the shares in the future (50 time steps ahead) and M_c is the mean cost to buy the shares at current time step. There is another case to consider. In case, the stocks are no longer available in the future, it means they got sold out. So, in this case too, we reward the agent positively and give it a fixed reward of +5.

- **Sell (1):** The reward for the sell action is profit/loss after selling the shares.
- **Hold (2):** As this is a high frequency scenario, we want the agent to be "active". So, we penalize the agent if it holds (does nothing) for too long. We have a hold threshold which is set to 10, so if the agent performs the hold action for 10 consecutive time steps then a reward of -10 is given. Else, a reward of 0 is returned.

6. Agent Design

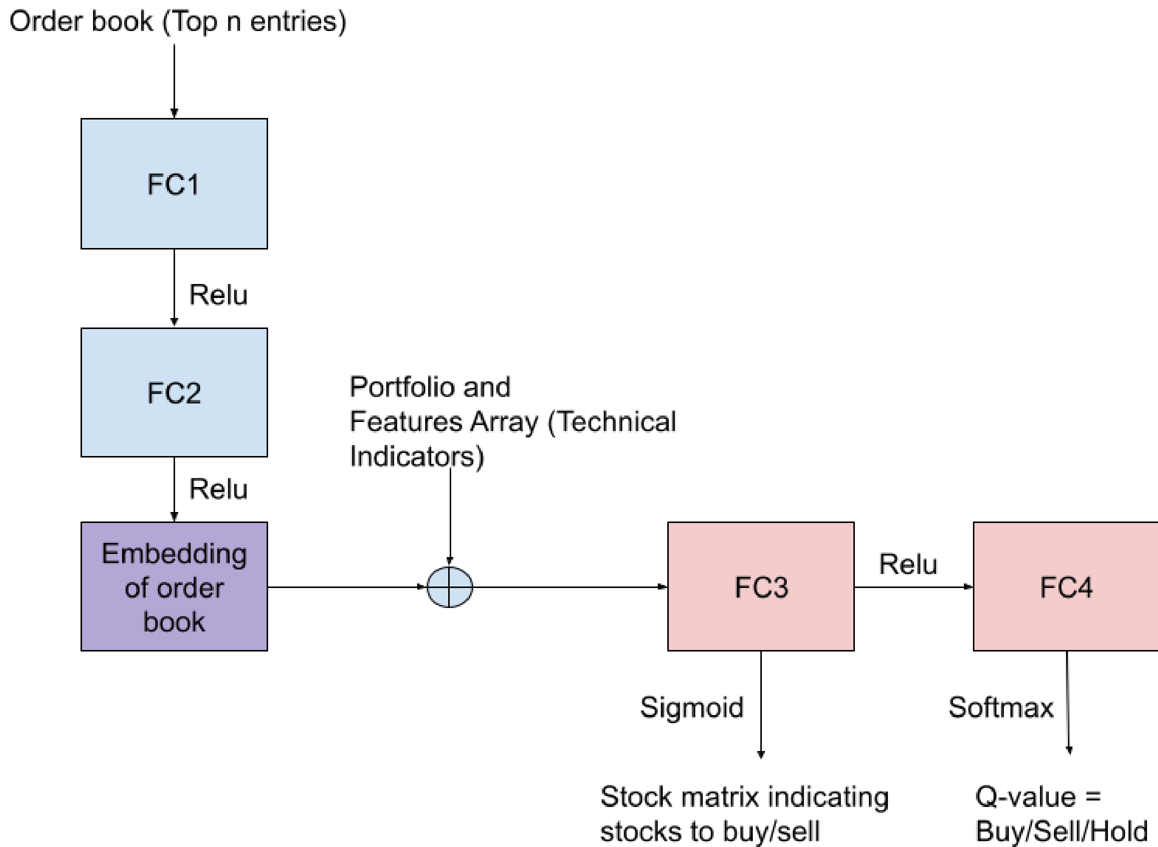


Figure 3: DDQN Agent Architecture

6.1 Double Deep Q-Network Agent

6.1.1 Q-Learning

A q-value tells us about the quality of an action given a (state, action) pair. This "quality" is expressed as the expected reward gained after taking some action a in some state s. it can be shown as follows:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a\right]$$

This is essentially the Bellman Equation which expresses Q-value as expectation over rewards such that the immediate reward is separated out from the future rewards.

If π is an optimal policy, then the optimal Q-value can be expressed as follows:

$$Q_*(s,a) = \max_\pi Q_\pi(s,a).$$

We can combine this idea with the Bellman equation shown to express Q-value update as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

This Q-update eventually converges to the optimal Q-value as at each time step we are getting the reward R_{t+1} from the environment.

6.1.2 Q-Network

We can easily expand the idea to Q-learning to Neural Networks. Instead of manually updating the q-values, we can approximate them directly using a neural network. The loss function can be expressed as follows:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right],$$

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a\right]$$

The gradient of the loss then becomes as follows:

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right].$$

We can see that there is one problem with this loss function. We can see that the max over the actions is being taken over the same network which is being used to predict the Q-values themselves. So, in a way the gradient is being pulled from both directions which leads to instability.

6.1.3 Double Deep Q-Networks

To solve the issue of loss instability, double Deep Q-networks use two copies of the Q-network. One is the target network which is kept stable for n iterations and the other is the policy network which is updated in every interval.

6.1.4 DDQN Agent Architecture

We can see from Figure 3, that the network is made up of a stack of fully connected (FC) layers. The reason for using only FC layers is that the data is in tabular form (order book array) and other features are simply numeric values (portfolio, technical indicators). Hence, specialized architectures do not exist to work on such data as usually a fully connected dense layer is enough to approximate functions emerging from such data.

First, we pass the order book array through two FC layers and get an embedding of order book. Then this embedding is concatenated with the portfolio array and the features array which are then further passed through two fully connected layers.

The final layer outputs a discrete action (buy/sell/hold), while the penultimate layer outputs the stocks to buy/sell. The reason for designing the architecture in such a way is that Q-value algorithm is usually designed to choose only one action and as we are using the PFRL framework, having multiple actions at the final layer would require changes to the library. By outputting the stock matrix, we eliminate the need for doing so. Also, as the two fully connected layers (FC3 and FC4) are in sequential order, the gradient is passed through both of them so they should learn meaningful representations such that both set of actions (discrete output and multi stock output) makes sense.

7. Experiments

7.1 Metrics

We calculate the efficacy of the model using different metrics on a time scale window of size n. For our experiments, we use n = 1000. That is, the metrics are calculated every 1000 steps using the values generated during those 1000 steps.

- **Mean Reward:** The mean reward simply calculates the mean of all rewards (dense reward structure) every 1000 steps.
- **Hit Ratio:** The hit ratio is defined as profitable trades over the total number of trades. We consider a trade profitable if the reward given by the environment is > 0.
- **Sharpe Ratio:** Sharpe ratio gives a measure of the risk adjusted return. It is given by the following formula:

$$Sharpe\ Ratio = \frac{R_p - R_f}{\sigma_p}$$

where:

R_p = return of portfolio

R_f = risk-free rate

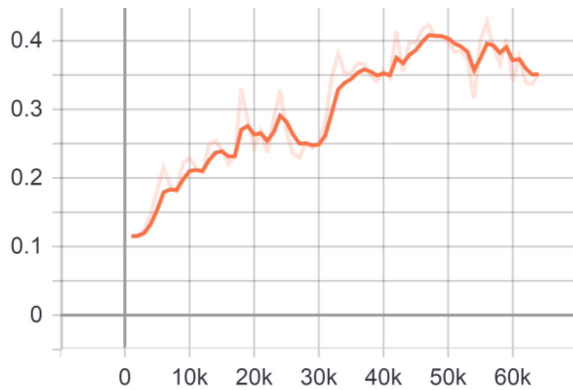
σ_p = standard deviation of the portfolio's excess return

The return of portfolio in our case is the mean return over 1000 steps (here return = reward from taking sell action) and the standard deviation of return is the s.d calculated over those mean returns. Risk-free rates are usually taken over larger time windows (months/years etc), so we chose a miniscule risk-free return value of 0.01% for a time window of 1000 steps.

7.2 DDQN Agent

For the DDQN agent, we used a learning rate of 1e-4 and discount factor of 0.99. A linear epsilon decay strategy was used which decays the epsilon from (1.0 to 0.1) up to 50k time steps. The target (stable) network was updated every 500th time step.

8. Results



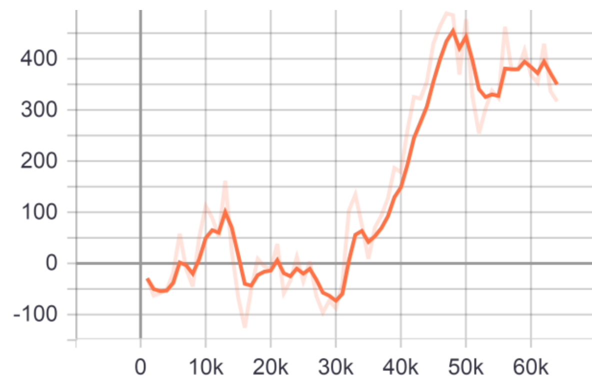


Fig 4a: Hit Ratio

Fig 4b: Mean Reward

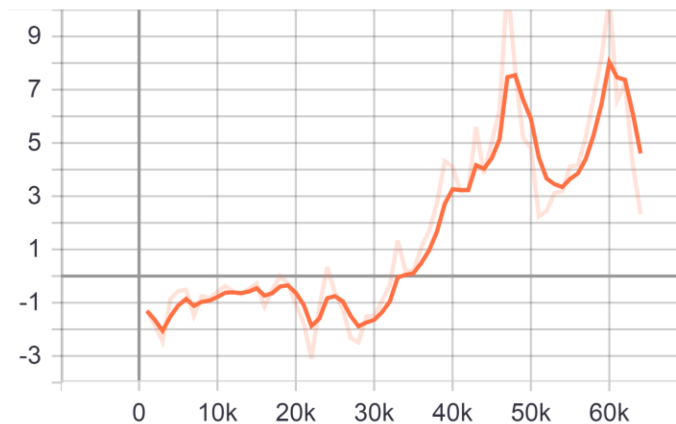


Fig 4c: Sharpe Ratio

From figure 4 we can see that the agent progressively started performing better in terms of all metrics. This is a promising result as it shows that the agent can leverage the limit order data to make meaningful decisions. After analyzing the actions taken by the agent, we noticed that the agent was performing far more buying actions as compared to the sell/hold action. This indicates that the denser reward of buy action is actually incentivizing the agent to choose to buy action over other actions; this may not always be a good thing. But from the Sharpe Ratio plot we can

see that when the agent does sell, it is usually a good high positive Sharpe ratio value indicating that the agent is making intelligent selling decisions.

9. Conclusion

Our environment provides an easy way to process the limit order data and build limit order books from it. From our experiments, we can see that the Reinforcement Learning agents are available to leverage this data to make intelligent decisions. As future work, we plan to test the performance of more agents (Ex – A2C, PPO) on the environment and design more robust architectures for these agents.

Data and resources

The limit order data used in this project is downloaded from the IEX Trading platform:

https://iextrading.com/trading/market-data/

The code of our project is available at our repository: https://github.com/Omkar-Ranadive/RL-Finance

References

[1] Dash, R., & Dash, P. K. (2016). A hybrid stock trading framework integrating technical analysis with machine learning techniques. The Journal of Finance and Data Science, 2(1), 42-57.

[2] Sezer, O. B., & Ozbayoglu, A. M. (2018). Algorithmic financial trading with deep convolutional neural networks: Time series to image conversion approach. Applied Soft Computing, 70, 525-538.

[3] Chen, J. F., Chen, W. L., Huang, C. P., Huang, S. H., & Chen, A. P. (2016, November). Financial time-series data analysis using deep convolutional neural networks. In 2016 7th International conference on cloud computing and big data (CCBD) (pp. 87-92). IEEE.

[4] Yang, H., Liu, X. Y., Zhong, S., & Walid, A. (2020). Deep reinforcement learning for automated stock trading: An ensemble strategy. Available at SSRN.

[5] Lee, J. W. (2001, June). Stock price prediction using reinforcement learning. In ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570) (Vol. 1, pp. 690-695). IEEE.

[6] Briola, A., Turiel, J., Marcaccioli, R., & Aste, T. (2021). Deep Reinforcement Learning for Active High Frequency Trading. arXiv preprint arXiv:2101.07107.

[7] Rundo, F. (2019). Deep LSTM with reinforcement learning layer for financial trend prediction in FX high frequency trading systems. Applied Sciences, 9(20), 4460.

[8] Fujita, Y., Kataoka, T., Nagarajan, P., & Ishikawa, T. (2019). ChainerRL: A deep reinforcement learning library. arXiv preprint arXiv:1912.03905.