Multi-Agent Reinforcement Learning in a Realistic Limit Order Book Market Simulation

Michaël Karpe[*]
Jin Fang[*]
michael.karpe@berkeley.edu
jin_fang@berkeley.edu
University of California, Berkeley
Berkeley, California

Zhongyao Ma[†]
Chen Wang[†]
mazy@berkeley.edu
chenwang@berkeley.edu
University of California, Berkeley
Berkeley, California

ABSTRACT

Optimal order execution is widely studied by industry practitioners and academic researchers because it determines the profitability of investment decisions and high-level trading strategies, particularly those involving large volumes of orders. However, complex and unknown market dynamics pose enormous challenges for the development and validation of optimal execution strategies. We propose a model-free approach by training Reinforcement Learning (RL) agents in a realistic market simulation environment with multiple agents. First, we have configured a multi-agent historical order book simulation environment for execution tasks based on an Agent-Based Interactive Discrete Event Simulation (ABIDES) [5]. Second, we formulated the problem of optimal execution in an RL setting in which an intelligent agent can make order execution and placement decisions based on market microstructure trading signals in HFT. Third, we developed and trained an RL execution agent using the Double Deep Q-Learning (DDQL) algorithm in the ABIDES environment. In some scenarios, our RL agent converges towards a Time-Weighted Average Price (TWAP) strategy. Finally, we evaluated the simulation with our RL agent by comparing the simulation on the actual market Limit Order Book (LOB) characteristics.

KEYWORDS

high-frequency trading, limit order book, market simulation, multi-agent reinforcement learning, optimal execution

[*]Both authors contributed equally to this research.
[†]Both authors contributed equally to this research.

1 INTRODUCTION

1.1 Agent-based Simulation for Reinforcement Learning in High-Frequency Trading

Simulation techniques form the basis for understanding market dynamics and evaluating trading strategies for both financial sector investment institutions and academic researchers. Current simulation methods are based on sound assumptions about the statistical properties of the market environment and the impact of transactions on the prices of financial instruments. Unfortunately, market characteristics are complex and existing simulation methods cannot replicate a realistic historical trading environment. The trading strategies tested by these simulations generally show lower profitability when implemented in real markets. It is therefore necessary to develop interactive agent-based simulations that allow trading strategy activities to interact with historical events in an environment close to reality.

High Frequency Trading (HFT) is a trading method that allows large volumes of trades to be executed in nanoseconds. In the United States, HFT companies account for more than 70% of daily equity trading volume. Execution strategies aim to execute a large volume of orders with minimal adverse market price impact. They are particularly important in HFT to reduce transaction costs. A common practice of execution strategies is to split a large order into several child orders and place them over a predefined period of time. However, developing an optimal execution strategy is difficult given the complexity of the HFT environment and the interactions between market participants.

The availability of NASDAQ's high-frequency LOB data allows researchers to develop model-free execution strategies based on RL through LOB simulation. These model-free approaches do not make assumptions or model market responses, but instead rely on realistic market simulations to train an RL agent to accumulate experience and generate optimal strategies. However, no existing research has implemented RL agents in realistic simulations, which makes the generated strategies suboptimal and not robust in real markets.

1.2 Related work

The use of RL for developing trading strategies has gained popularity in recent years. HFT makes necessary the use of RL automate and accelerate order placement. Many papers present such RL approaches, such as temporal-difference RL [12] and risk-sensitive RL [9].

Although RL strategies have proven their effectiveness, they suffer from a lack of explainability. Thus, the need to be able to explain these strategies in a business context has led to the development of representations of risk-sensitive RL strategies in the form of compact decision trees [16]. Advances in the development of RL agents for trading and order placement then showed the need to learn strategies in an environment close to the real market environment. Indeed, traditional RL approaches suffer from two main shortcomings.

First, each financial market agent adapts its strategy to the strategies of other agents, in addition to the market environment. This has led research in the field to consider the use of Multi-Agent Reinforcement Learning (MARL) for learning trading and order placement strategies [11]. Second, the market environment simulated in classical RL approaches was simplistic. The creation of a standardized market simulation environment for artificial intelligence agent research was then undertaken to allow agents to learn in conditions closer to reality, through the creation of ABIDES [5]. Research works on the metrics to be considered to evaluate the agents of RL in this environment was also supported within the framework of LOB simulation [15].

As MARL and ABIDES allow a simulation much closer to real market conditions, additional research was undertaken to address the curse of dimensionality, as millions of agents compete in traditional market environments. The use of mean field MARL allows faster learning of strategies by approximating the behavior of each agent by the average behavior of its neighbors [17].

The notion of fairness also brings both efficiency and stability to MARL [7] by making it possible to avoid situations where agents could act disproportionately in the market, for example by executing large orders. The integration of fairness into MARL [2] has been studied as an evolution of traditional MARL strategies used for example for liquidation strategies [3].

1.3 Contributions

Our main contributions in HFT simulation and RL for optimal execution are the following:

- We have set up a multi-agent LOB simulation environment for the training of RL execution agents within ABIDES.
- we have formulated the problem of optimal execution within an RL framework, consisting of a combination of action spaces, private states, market states and reward functions. In particular, this is the first formulation with optimal execution and optimal placement combined in the action space.
- We have developed RL execution agents using the Double Deep Q-Learning (DDQL) algorithm in the ABIDES environment.
- We trained an RL agent in a multi-agent LOB simulation environment. Our RL agent converges to the TWAP strategy in some situations.
- We evaluated the multi-agent simulation with a trained RL agent based on the real market LOB characteristics and the observed order flow model is consistent with LOB stylized facts.

2 OPTIMAL EXECUTION USING DOUBLE DEEP Q-LEARNING

2.1 Optimal execution formulation

In our work, we allow RL agents not only to choose the order volume to be placed, but also to choose between a market order and one or more limit orders at different levels of the order book. In this section, we describe the states, actions, and rewards of our optimal execution problem formulation.

We defined the trading simulation as a T-period problem, which is denoted by the times $T_0 < T_1 < \cdots < T_N$ with $T_0 = 0$. We will focus on the time horizon from 10:00 to 15:30 for each trading day to avoid the most volatile periods during the agent training process. The time interval within each period is $\Delta T = 30$ seconds, so that there is a total of 660 periods within the time horizon we have defined in a trading day, lasting 5 hours (i.e. $T_N = T/\Delta T = 660$). The capital P represents the price, while the capital Q represents the quantity volume at a certain price in the limit order book. Our optimal execution problem is then formulated as follows:

(1) **State** s: the state space includes the information on the LOB at the beginning of each period. For each time period, we use a tuple containing the following characteristics to represent the current state:

- *time_remaining* t: the time remaining after the time period T_k. Since we assume that a trade can only take place at the beginning of each period, this variable also contains the number of remaining trading times. The variable is normalized to be in the $[-1, 1]$ range as follows: $t = 2 \times \frac{T-t}{T} - 1$
- *quantity_remaining* n: the quantity of remaining inventory at the time period T_k, which is also normalized: $n = 2 \times \frac{N - \sum_{i=0}^{t} n_i}{N} - 1$, where the capital N denotes the initial inventory

The above state variables are linked to specific execution tasks, called private states. In addition, we also use the following market state variables to capture the market situation at a given point in time:

- *bid-ask spread*: the difference between the highest bid price and the lowest ask price, which is intended to provide information on the liquidity of the asset in the market:

$$s = P_{best\_ask} - P_{best\_bid}$$

- *volume imbalance*: the difference between the existing order volume on the best bid and best ask price levels. This feature contains information on the current liquidity difference on both sides of the order book, indirectly reflecting the price trend.

$$v_{imbalance} = \frac{Q_{best\_ask} - Q_{best\_bid}}{Q_{best\_ask} + Q_{best\_bid}}$$

- *one-period price return*: the log-return of the stock price over two consecutive days measures the short-term price trend. We intend to allow the RL agent to take advantage of the mean-reverting characteristics of the stock price.

$$r_1 = \log\left(\frac{P_t}{P_{t-1}}\right)$$

- *t-period price return*: the log-return of the stock price since the beginning of the simulation measures the deviation between the stock price at time t and the initial price at time 0.

$$r_t = \log\left(\frac{P_t}{P_0}\right)$$

(2) **Action a:** the action space defines a possible executed order in a given state, i.e. the possible quantity of remaining inventory affected by the order. In this case, the order can be either a market order or a limit order, either from the bid side or from the ask side. Therefore, the action for each state would be a combination of the quantity to be executed and the direction for placement. We use the execution choice to indicate the former and the placement choice to represent the latter.
  - *Execution Choices:* At the beginning of each period, the agent shall decide on an execution quantity $N_t = a \cdot N_{TWAP}$, where a is a scalar the agent chooses from a set of numbers to increase or decrease the order quantity placed using the TWAP strategy. The scalar a is in $[0.1, 0.5, 1.0, \ldots, 2.5]$.
  - *Placement Choices:* The agent can choose one of the following order placement methods:
    - choice 0: Market Order
    - choice 1: Limit Order - place 100% on top-level of LOB
    - choice 2: Limit Order - place 50% on each of top 2 levels
    - choice 3: Limit Order - place 33% on each of top 3 levels

(3) **Reward** r**:** the reward is intended to reflect the feedback from the environment after agents have taken a given action in a given state. It is usually represented by a reward function consisting of useful information obtained from the state or the environment. In our formulation, the reward function R_t measures the execution price slippage and quantity.

$$R_t = \left(1 - \frac{|P_{fill} - P_{arrival}|}{P_{arrival}}\right) \cdot \lambda \frac{N_t}{N}$$

where λ is a constant for scaling the effect of the quantity component.

2.2 Double Deep Q-Learning algorithm

Aiming to achieve the optimal execution policy, RL enables agents to learn the best action to take through interaction with the environment. The agent follows the strategy that can maximize the expectation of cumulative reward. In Q-Learning, it is represented by the function below [10]:

$$Q(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \times R(s, a_t, s')\right]$$

Since the above approach would be redundant with larger dimensions of the state space where states cannot be visited in depth, instead of directly using a matrix as the Q-function, we can learn a feature-based value function $Q(s, a|\theta)$, where θ is the weighting factor that is updated by a stochastic gradient descent.

The Q-value with parametric representation can be estimated by multiple flexible means, in which the Deep Q-Learning (DQL) best fits our problem. In the DQL, the Q-function $Q(s, a|\theta)$ is combined with a Deep Q-Network (DQN), and θ is the network parameters.

The network memory database contains samples with tuples of information recording the current state, action, reward and next state (s, a, r, s'). For each period, we generate samples according to an ε-greedy policy and store them in memory. We replace the samples when the memory database is full, following the First-In-First-Out (FIFO) principle, which means that old samples are removed first.

At each iteration, we batch a given quantity of samples from the memory database, and compute the target Q-value for each sample, which is defined as [10]:

$$y = R(s, a) + \gamma \times \max_a Q(s, a|\theta)$$

The network parameter θ is updated by minimizing the loss between the target Q-value and the estimated Q-value calculated from the network based on the current parameters.

However, the DQL algorithm suffers from both instability and overestimation problems, since the neural network is used to generate the current target Q-value as well as to update the parameters. A common method to solve this problem is to introduce another neural network with the same structure and calculate the Q-value separately, which is called Double Deep Q-Learning (DDQL) [14].

In DDQL, we use two neural networks, the evaluation network and the target network, to generate the appropriate Q-value. The evaluation network is used to select the best action a^* for each state in the sample, while the target network is used to estimate the target Q-value. We update the evaluation network parameters θ_E every period, and replace the target network parameters θ_T with $\theta_T = \theta_E$ after several iterations.

3 REINFORCEMENT LEARNING IN ABIDES

3.1 ABIDES environment

ABIDES is an Agent-Based Interactive Discrete Event Simulation environment primarily for the development of Artificial Intelligence (AI) research in financial market simulations [5].

The first version of ABIDES *(0.1)* was released in April 2019. The ABIDES development team released a second version *(1.0)* in September 2019, which is supposed to be the first stable release. Finally, the latest version *(1.1)*, released in March 2020, adds many new functionalities, including the implementation of new agents such as Q-Learning agents, as well as the implementation of the realism metrics.

ABIDES aims to replicate a realistic financial market environment by largely implementing the characteristics of real financial markets such as NASDAQ, including *nanosecond time resolution, network latency and agent computation delays and communication solely by means of standardized message protocols* [5]. In addition, by providing ABIDES with historical LOB data, we are able to reproduce a given period of this history using ABIDES *marketreplay* configuration file.

ABIDES also aims to help researchers to answer questions related to the understanding of market behavior, such as the influence of delays in sending orders to an exchange, the price impact of placing large orders or the implementation of AI agents into real markets [5].

ABIDES uses a hierarchical structure in order to ease the development of complex agents such as AI agents. Indeed, thanks to

Python object-oriented programming and *inheritance*, we can, for example, create a new *ComplexAgent* class which inherits from the *Agent* and thus benefits from all functionalities available in the *Agent* class. We can then use *overriding* if we want to change a *Agent* function in order to make it specific for our *ComplexAgent.*

Given that ABIDES not only aims to implement financial market simulations, the base *Agent* class has nothing related to financial markets is only provided with functions for Discrete Event Simulation. The *FinancialAgent* class inherits from *Agent* and has supplementary functionalities to deal with currencies. On the one hand, the *ExchangeAgent* class inherits from *FinancialAgent* and simulates an financial exchange. On the other hand, the *TradingAgent* also inherits from *FinancialAgent* and is the base class for all trading agents which will communicate with the *ExchangeAgent* during financial market simulations.

Some trading agents – i.e. inheriting from the *TradingAgent* class – are already provided in ABIDES, such as the *MomentumAgent* which places orders depending on a given number of previous observations of the simulated stock price. In the next sections, unless otherwise mentioned, the agents we refer to are all trading agents.

We present in Figure 1 an example of market simulation in ABIDES using real historical data. The graph below presents a *ZeroIntelligenceAgent* placing orders on a stock in a market simulation with 100 *ZeroIntelligenceAgent* trading against an *ExchangeAgent.* Each agent is able to place long, short or exit orders, competing with thousands of other agents to maximize their reward.
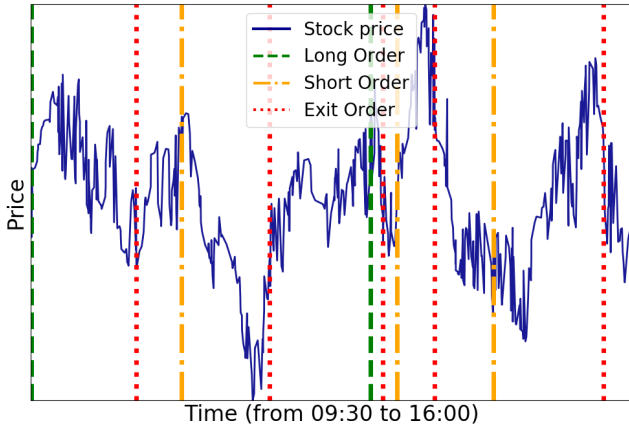

Figure 1: Agent placing orders on simulated stock price

3.2 DDQN implementation in ABIDES

In order to train the DDQL agent in ABIDES during a *marketreplay* simulation, the learning process needs to be integrated with the simulation process. The training process starts by initializing the ABIDES execution simulation kernel and instantiating a *DDQLExecutionAgent* object. The same agent object needs to complete B simulations, which is referred to as the number of training episodes.

Within each training episode, the simulation is divided into N discrete periods. For each period T_i, the agent chooses an action a_{T_i} for the current period according to the ε-greedy policy in order

to achieve a balance of exploration and exploitation. Then, an order schedule is generated based on the quantity and placement strategy defined in the chosen action. The current-period order could be broken into small orders and placed on different levels of the LOB. Then, the current experience $\left(s_{T_i}, a_{T_i}, s_{T_{i+1}}, r_{T_i}\right)$ is stored in the replay buffer $\mathcal{D}$.

The replay buffer removes the oldest experience when its size reaches to the maximum capacity specified. This intends to use relatively recent experiences to train the agent. As long as the size of the replay buffer $\mathcal{D}$ reaches a minimum training size, a random minibatch $\left(s_{(j)}, x_{(j)}, r_{(j)}, s_{(j)}^{s,x}\right)$ is sampled from $\mathcal{D}$ for training the evaluation network.

The target network is updated after training the evaluation network 5 times. The final step within time period T_i is to update the state $s_{T_{i+1}}$ and compute the reward r_{T_i} for the current period. The entire process is summarized in the algorithm below.

Algorithm 1 Training of DDQL for optimal execution in ABIDES.

1: **for** training episode $b \in B$ **do**
2:      **for** $i \leftarrow 0$ to $N - 1$ **do**
3:          With probability ε select random action a_{T_i}
4:          Otherwise select optimal action a_{T_i}, based on target_net
5:          Schedule orders o_i according to a_i and submit o_i
6:          Store experience $\left(s_{T_i}, a_{T_i}, s_{T_{i+1}}, r_{T_i}\right)$ in replay buffer $\mathcal{D}$
7:          **if** $length(\mathcal{D}) > max\_experience$ **then**
8:             Remove oldest experience
9:          **if** $length(\mathcal{D}) \geq min\_experience$ AND $i \mod 5 == 0$ **then**
10:             Sample random minibatch from $\mathcal{D}$
11:             Train eval_net and update target_net
12:          **if** orders o_i accepted or executed **then**
13:             Observe environment and update $s_{T_{i+1}}$
14:             Compute and update r_{T_i}

To train a DDQL agent, we implement our neural network based on a Multi Layer Perceptron (MLP). We stack multiple dense layers together as illustrated in Figure 2, and we set the activation function to be ReLU to introduce non-linearity. Dropout is used to avoid overfitting. The optimization algorithm used for backpropagation is the *Root Mean Square back-propagation* (RMSprop) [13] with a learning rate of 0.01. The loss function we choose is the *mean squared error* (MSE). The size of the output layer size is the number of actions to choose from.

4 EXPERIMENTS AND RESULTS

In this section, we describe our experiment for training a DDQL agent in a multi-agent environment and observe the behavior of the agent during testing.

4.1 Data for experiments

The data we used for the experiments is NASDAQ data we converted to LOBSTER format to fit the simulation environment. We extracted the order flow for 5 stocks (CSCO, IBM, INTC, MSFT and YHOO) from January 13 to February 6, 2003. We trained the model over 9 days and tested it over the following 9 days. The training data
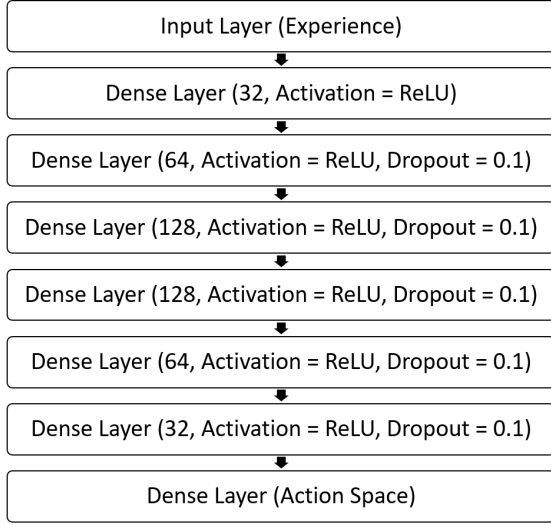
Figure 2: Multi-Layer Perceptron (MLP) architecture

is concatenated into a single sequence, and the training process is continuous for consecutive days while the model parameters are stored in intermediate files.

4.2 Multi-agent configuration

The multi-agent environment that we have set up for the training of DDQL agents at ABIDES consists of an *ExchangeAgent*, a *MarketreplayAgent*, six *MomentumAgents*, a *TWAPExecutionAgent* and our *DDQLExecutionAgent*.

- *ExchangeAgent* acts as a centralized exchange that keeps the order book and matches orders on the bid and ask sides.
- *MarketreplayAgent* accurately replays all market and limit orders recorded in the historical LOB data.
- *MomentumAgent* compares the last 20 mid-price observations with the last 50 mid-price observations and places a buy limit order if the 20 mid-price average is not lower than the 50 mid-price average, or a sell limit order if it is.
- *TWAPAgent* adopts the TWAP strategy. This strategy minimizes the price impact by dividing a large order equally into several smaller orders. Its execution price is the average price of the recent k time periods. The agent's optimal trading rate is calculated by dividing the total size of the order by the total execution time, which means that the trading quantity is constant. When the stock price follows a Brownian motion and the price impact is assumed to be constant, this is the optimal strategy [6]. In RL, if there is no penalty for any running inventory, but a significant penalty for the ending inventory, the TWAP strategy is also optimal.

4.3 Result of our experiments

We have observed that our RL agent converges to the TWAP strategy after 9 consecutive days of training regardless of the stock chosen. The agent places a top-level limit order or market order. However, the execution quantity chosen by the agent throughout the test period changes with the stock. The possible reason for this

result is that, as our RL agent places an order every 30 seconds, it is not able to capture the trading signals existing during shorter periods of time.

5 REALISM OF OUR LOB SIMULATION

Numerous research papers have studied the behaviour of the LOB. A recent research paper presents a review of these LOB characteristics which can be referred to as *stylized facts* [15]. In this section, we compare our simulation with real markets based on these *realism metrics*, in order to assess whether our market simulation, mainly based on ABIDES, is realistic.

We can mainly distinguish two sets of metrics for the analysis of the LOB behavior. The first set includes metrics related to asset return distributions and the second set includes metrics related to volume and order flow distributions [15].

Asset returns metrics generally relate to price return or percentage change. For the LOB, it includes the mid-price trend, which is the average of the best bid price and the best ask price. Volumes and order flow metrics relate to the behavior of incoming order flows, including new buy orders, new sell orders, order modifications or order cancellations.

We briefly recall three main stylized facts related to order flows [15]:

- **Order volume in a fixed time interval:** Order volume in a fixed interval or time window likely follows a positively skewed log-normal distribution or gamma distribution [1].
- **Order interarrival time:** The time interval of two consecutive limit orders likely follows an exponential distribution [8] or a Weibull distribution [1].
- **Intraday volume patterns:** Limit order volume within a given time interval for each trading day can be approximated by a U-shaped polynomial curve, where the volume is higher at the start and the end of the trading day [4].

These realism metrics are implemented in ABIDES. We verified the order flow stylized facts mentioned above, as well as the asset returns stylized facts that are also implemented in ABIDES. We did it on a *marketreplay* simulation before adding our *DDQLExecutionAgent*, and then on a simulation with our *DDQLExecutionAgent*. We first observe that adding a single new agent to a simulation does not significantly alter the observation of stylized facts. This means that evaluating the realism of our simulation with a single *DDQLExecutionAgent* is equivalent to evaluating the realism of the LOB data provided as input to the simulation.

On our NASDAQ LOB 2003 data, we always observe the two order flow stylized facts mentioned above, however we do not always observe intraday volume patterns. Figure 3 illustrates the stylized fact about order volume in a fixed time interval, for IBM stock on January 13, 2003. We verify that order volume in a fixed time interval follows a gamma distribution.

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

In the work presented, we built our *DDQLExecutionAgent* in ABIDES by implementing our own optimal execution problem formulation through RL in a financial market simulation, and set up a
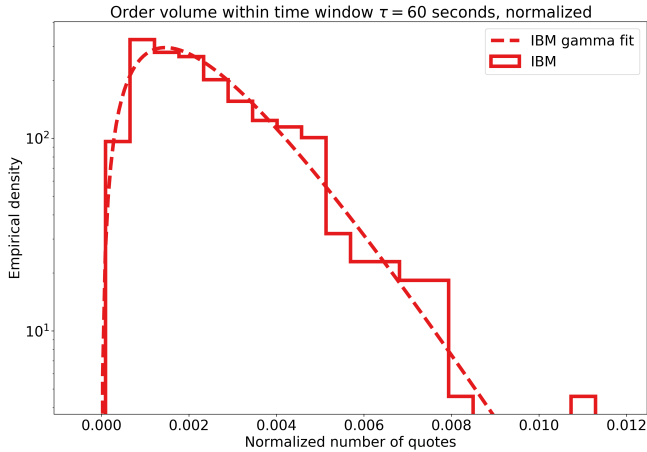
Figure 3: Order volume in fixed time interval for IBM stock on January 13, 2003

multi-agent simulation environment accordingly. In addition, we conducted experiments to train our *DDQLExecutionAgent* in the ABIDES environment and compared the agent strategy with TWAP. Finally, we evaluated the multi-agent simulation environment in the financial field which showed that it follows stylized facts about order flow patterns.

In our experiments, our *DDQLExecutionAgent* learned how to perform a TWAP strategy because of its trading frequency which is not high enough. However, this work shows the potential of MARL for developing optimal trading strategies in real financial markets, by implementing agents with an higher trading frequency in the realistic ABIDES market simulation.

6.2 Future work

Due to limited computing resources and lack of data, the experiments we have been able to do are limited. Our current model can be improved in many ways. The agent period of time that we set can be refined to a shorter time interval, closer to the nanosecond, in order to be closer to real HFT. More features can be added in the state space, and the action space can be expanded to include more types of execution actions. In addition, the reward function can be enhanced to include more information and feedback from both the market and other agents.

Regarding the RL algorithm, we can try several advanced methods to implement an updated approach on our *DDQLExecutionAgent*. Directions include the use of prioritized experience replay to increase the frequency of batching important transitions from memory, or the combination of bootstrapping with DDQL to improve exploration efficiency. In addition, the performance of the neural network itself can also be improved by increasing the complexity of the architecture. For example, since agents' trading decisions may also depend on previous observations, several LSTM layers can be added to take advantage of the agent's past experience.

So far, we have focused on a relatively monotonous set of multiple agents, which is not able to fully capture the influence of the interaction between the agents. To remedy this situation, more and different types of agents can be added to the configuration to study collaboration and competition among agents in more detail. Moreover, after introducing a more complex combination of agents in the ABIDES environment, we can try to perform the financial market simulation on the basis of this configuration, which should be much more realistic than the existing one.

The approach to evaluation is also an aspect that can be further expanded. Our experience does not currently allow us to clearly distinguish the difference between our agents and the benchmark. In order to assess the model more accurately, we can further improve our evaluation methods to examine both the parameters of RL and financial performance. For example, by conducting a simulation of the financial market in the ABIDES environment, we can use the realism metrics we have designed to evaluate our agents.

ACKNOWLEDGMENTS

REFERENCES

[1] Frédéric Abergel, Marouane Anane, Anirban Chakraborti, Aymen Jedidi, and Ioane Muni Toke. 2016. *Limit order books*. Cambridge University Press.
[2] Wenhang Bao. 2019. Fairness in Multi-agent Reinforcement Learning for Stock Trading. *arXiv preprint arXiv:2001.00918* (2019).
[3] Wenhang Bao and Xiao-yang Liu. 2019. Multi-agent deep reinforcement learning for liquidation strategy analysis. *arXiv preprint arXiv:1906.11046* (2019).
[4] Jean-Philippe Bouchaud, Julius Bonart, Jonathan Donier, and Martin Gould. 2018. *Trades, quotes and prices: financial markets under the microscope*. Cambridge University Press.
[5] David Byrd, Maria Hybinette, and Tucker Hybinette Balch. 2019. Abides: Towards high-fidelity market simulation for ai research. *arXiv preprint arXiv:1904.12066* (2019).
[6] Kevin Dabérius, Elvin Granat, and Patrik Karlsson. 2019. Deep Execution-Value and Policy Based Reinforcement Learning for Trading and Beating Market Benchmarks. *Available at SSRN 3374766* (2019).
[7] Jiechuan Jiang and Zongqing Lu. 2019. Learning Fairness in Multi-Agent Systems. In *Advances in Neural Information Processing Systems*. 13854–13865.
[8] Junyi Li, Xintong Wang, Yaoyang Lin, Arunesh Sinha, and Michael P Wellman. 2018. Generating Realistic Stock Market Order Streams. (2018).
[9] Mohammad Mani, Steve Phelps, and Simon Parsons. 2019. Applications of Reinforcement Learning in Automated Market-Making. (2019).
[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
[11] Yagna Patel. 2018. Optimizing Market Making using Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:1812.10252* (2018).
[12] Thomas Spooner, John Fearnley, Rahul Savani, and Andreas Koukorinis. 2018. Market making via reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 434–442.
[13] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012), 26–31.
[14] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*.
[15] Svitlana Vyetrenko, David Byrd, Nick Petosa, Mahmoud Mahfouz, Danial Dervovic, Manuela Veloso, and Tucker Hybinette Balch. 2019. Get Real: Realism Metrics for Robust Limit Order Book Market Simulations. *arXiv preprint arXiv:1912.04941* (2019).
[16] Svitlana Vyetrenko and Shaojie Xu. 2019. Risk-Sensitive Compact Decision Trees for Autonomous Execution in Presence of Simulated Market Response. *arXiv preprint arXiv:1906.02312* (2019).
[17] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. 2018. Mean field multi-agent reinforcement learning. *arXiv preprint arXiv:1802.05438* (2018).