A Financial Market Simulation Environment for Trading Agents Using Deep Reinforcement Learning

Chris Mascioli
University of Michigan
Ann Arbor, USA
cmasciol@umich.edu

Anri Gu
University of Michigan
Ann Arbor, USA
anrigu@umich.edu

Yongzhao Wang
University of Liverpool
The Alan Turing Institute
Liverpool, UK
yongzhao.wang@liverpool.ac.uk

Mithun Chakraborty
University of Michigan
Ann Arbor, USA
dcsmc@umich.edu

Michael P. Wellman
University of Michigan
Ann Arbor, USA
wellman@umich.edu

Abstract

We present PyMarketSim, a financial market simulation environment designed for training and evaluating trading agents using deep reinforcement learning (dRL). Our agent-based environment incorporates key elements such as private valuations, asymmetric information, and a flexible limit order book mechanism. We demonstrate the efficiency and versatility of our platform through experiments including both single-agent and multi-agent dRL settings. For single-agent settings, we showcase how our environment can be used to learn background trading strategies implemented as recurrent neural networks. These *trained response order networks* (TRON agents) can flexibly condition their behavior on observed market characteristics. At the multi-agent level, we use empirical game-theoretic techniques to identify equilibrium configurations of TRON agents. Our open-source implementation provides researchers and practitioners with a powerful tool for studying complex market dynamics, developing advanced trading algorithms, and exploring the emergent behaviors of financial ecosystems driven by machine learning.

ACM Reference Format:
Chris Mascioli, Anri Gu, Yongzhao Wang, Mithun Chakraborty, and Michael P. Wellman. 2024. A Financial Market Simulation Environment for Trading Agents Using Deep Reinforcement Learning. In *5th ACM International Conference on AI in Finance (ICAIF '24), November 14–17, 2024, Brooklyn, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3677052.3698639

1 Introduction

Financial market simulation has proved a useful tool for modeling the interplay of market microstructure and agent behavior. As technological advances enable new market designs and algorithmic trading behavior, understanding the implications becomes yet more important for market participants and other stakeholders. A particular development with transformational potential is the emergence of ***deep reinforcement learning*** (dRL) [34] as a powerful tool for automated generation of sophisticated trading strategies.

We address this need by updating and enhancing MarketSim: a comprehensive financial market simulation environment modeling agents who trade through mechanisms based on limit-order books (LOBs). MarketSim has been employed in numerous agent-based studies [33, 35, 36]. Our new implementation is designed to facilitate the study of markets with trading policies trained using dRL. By integrating dRL with flexible agent-based simulation of LOB markets, we enable researchers to explore the implications of AI for trading in a wide range of market scenarios with precise specification of market microstructure and strategic interaction.

The contributions of this paper are threefold

(1) We release PyMarketSim with an MIT license to provide researchers the tools to use this in their own work.[1] Our simulator is highly optimized to work inside deep RL workflows and comes along with wrappers to be used inside Gymnasium APIs along with a variety of different background traders to use inside experiments.
(2) We develop a new type of background agent, the *trained response order network* (TRON) agent, which employ policies implemented by a recurrent neural network. These networks are trained by dRL, in response to a specified market environment. TRON agents exhibit a flexible ability to condition behavior on market conditions, compared to heuristic trading strategies.
(3) We show how to equilibrate TRON agents using Policy Space Response Oracles (PSRO) [24], a form of empirical game-theoretic analysis [41] based on dRL.

In the following sections, we provide a detailed description of the PyMarketSim environment, including its valuation model, LOB implementation, and agent types. We then present experiments demonstrating the capabilities of our simulation platform in both single-agent and multi-agent RL scenarios. Finally, we discuss the implications of our work and outline directions for future research.

ICAIF '24, November 14–17, 2024, New York, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1081-0/24/11
https://doi.org/10.1145/3677052.3698639

[1]Available at github.com/umichsrg/pymarketsim. The package includes sample configurations as well as a tutorial.

2 Related Work

2.1 Market Simulation

Researchers have employed simulation to study financial markets since at least the early 1990s. Studies employing the Santa Fe Artificial Stock Market [28] kicked off a literature in agent-based finance [25], which produced many useful ideas for financial modeling.

PyMarketSim is a Python reimplementation and extension of MarketSim, originally developed in Java by Elaine Wah for an agent-based study of latency arbitrage [35]. MarketSim was based on a discrete-event scheduler, with a modular design to flexibly accommodate a diversity of trading strategies and market mechanisms, organized around an efficient order book data structure. The original version was applied by researchers at the University of Michigan in investigations covering such topics as spoofing [37], welfare effects of market making, [36], and benchmark manipulation [33]. The new version supports training of new trading strategies using dRL, and incorporation of trained traders in simulation.

ABIDES (Agent-Based Interactive Discrete Event Simulation) [6] is likewise based on discrete-event processing, and designed for flexible incorporation of agent trading strategies. ABIDES further employs a uniform message-passing architecture based on standard market protocols. The system has achieved widespread use in the research community, thanks to the support of JP Morgan AI Research. An augmented version, ABIDES-Gym [1], provides an interface to the popular OpenAI Gym environment for dRL.

An alternative to full agent-based simulation is to focus on interaction with a LOB defined by an exogenous order process. This allows simulation of trading strategies with respect to historical data (i.e., *backtests*), or with a mathematical LOB model [19]. Recently, Frey et al. [12] introduced JAX-LOB, a LOB simulator designed to exploit GPU computation via JAX [5] libraries. Jerome et al. [20] likewise developed a LOB simulator with interfaces to facilitate training of dRL agents.

Other recent works have also employed LOB market simulation to train dRL trading agents. Maeda et al. [26] use an advantage-actor-critic network for this purpose in a simulated market of their design. Karpe et al. [22] apply double deep Q-learning to the optimal order execution task within the ABIDES framework. Yao et al. [44] study a simulated LOB market including RL-based agents with designated roles (liquidity-providers and liquidity-takers) based on prior work by Ardon et al. [2].

2.2 Trading Agents

Work in agent-based finance and AI introduced a variety of trading agents based on heuristic strategies for LOB markets [40]. Financial market simulations typically start with basic strategies from this literature. Perhaps most prominent is the ***zero intelligence*** (ZI) strategy introduced by Gode and Sunder [14] to illustrate that markets could reach competitive equilibrium without sophisticated agents. Farmer et al. [11] argued that ZI agents are sufficient to produce realistic results. Several authors proposed extensions of ZI that adjust parameters based on experience in the market. ZI Plus (ZIP) agents [8] incorporate a learning rate and momentum term to adapt behavior to observed conditions. An alternative approach to incorporating experience is to explicitly learn a belief function assessing the probability of transaction as a function of bid price [13].

Researchers have also developed offline learning methods for trading agents. Phelps et al. [31] used genetic search to set parameters for a heuristic strategy, in order to maximize evolutionary fitness. Cliff [9] defined a version of ZIP with 60 parameters, also tuned via a genetic algorithm. Schvartzman and Wellman [32] employed RL to find profit-maximizing policies over a tile-coded state-action space. Many works since these have explored RL and other learning methods for LOB environments.

3 PyMarketSim

A scenario in PyMarketSim comprises N agents or traders interacting to buy and sell securities through K markets. The agents are partitioned into a set of ***roles*** $\mathcal{R}$, for example we might have distinct roles for background traders and market makers. Each market is associated with a single security and follows specified rules of its ***market mechanism***. For simplicity in the descriptions and examples below we focus on a single market mechanism ($K = 1$), however many applications of financial market simulation hinge on issues involving multiple inter-related markets and PyMarketSim is built with the more complicated cases in mind.

3.1 Simulation Scheme

At the core of PyMarketSim is the Simulator object, which serves as the central orchestrator: managing markets, agents, and their interactions. As depicted in Fig. 1, the simulator controls the temporal flow of the simulation and mediates how and when agents interact with the market environment. The simulation unfolds in discrete time steps, with each agent of role $r \in \mathcal{R}$ having a probability λ_r of arrival to the market at any given time step. This activation probability allows for varying reaction times and trading frequencies among different agent roles. The simulator employs an efficient event-driven approach, focusing computational resources on time steps where market activity occurs. This approach significantly reduces runtime compared to a naive implementation that would process every time step regardless of activity.

When at least one agent arrives[2] or the market needs to take an action during a time step, the simulator executes the following sequence of operations:

(1) **Observation Phase**. For each arriving agent, the simulator retrieves observations from all markets the agent is permitted to interact with. These observations typically include the current state of the order book, recent trades, and other relevant market data.
(2) **Action Phase**. The agents who arrive are randomly shuffled, and then place orders in turn. Each order consists of a price, quantity, time step, and a unique order identifier. The random shuffling prevents any systematic bias that could arise from a fixed order of agent actions.
(3) **Market Operation**. The market takes its actions such as handling trades, paying dividends, posting information, etc.

[2]Given sufficiently fine time granularity, multiple agents arriving in the same time step will be rare. The system needs to handle these situations, however, so we describe the general case.

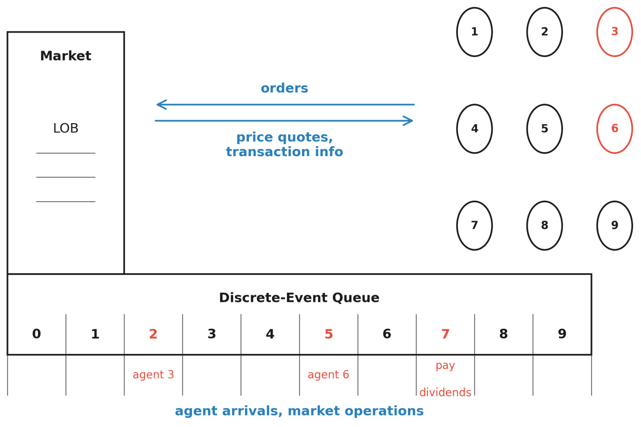

Figure 1: Interaction between traders and a market in MarketSim. A discrete-event queue controls the arrivals of traders and market-initiated events.

(4) **Time Advancement**. The simulator advances to the next time step where at least one agent is scheduled to arrive or the market has a scheduled action.

After each arrival, PyMarketSim generates the agent's next arrival time in advance using the geometric distribution. This allows it to record the arrival event on the scheduler. Inactive time steps are simply skipped, reducing the effective number of steps processed from T (total simulation time) to approximately $N \times \lambda \times T$, where N is the number of agents and λ the average arrival rate.

The simulator also manages market events, such as dividend payments or information releases, through the scheduling mechanism. Market operations may be scheduled for specific time steps, or triggered by other events.

The simulation scheme in PyMarketSim is designed to be flexible and extensible. Researchers can implement custom market mechanisms, introduce new agent types, or modify the activation process to suit specific needs. This flexibility makes PyMarketSim a powerful tool for exploring a wide range of market scenarios and trading strategies.

3.2 Market

MarketSim implements market mechanisms for simulating financial asset trading. While the framework supports multiple market types and assets, our experiments focus on a continuously clearing market for a single asset. The market module in PyMarketSim has the following key components:

(1) **Agent Interaction**: The market interfaces with multiple agents, processing their orders and providing market information.
(2) **Information Dissemination**: It provides agents with:
   - a (noisy) observation of the fundamental value
   - current best price offered to buy (***BID***) and best price to sell (***ASK***)
   - current time step
(3) **Order Processing**: Orders received from agents are passed to the LOB for processing.

Operation	Complexity
Insert	$O(\log(n))$
Remove	$\approx O(\log(n))$
Quote	$O(1)$
Clear	$O(\lvert S_{in}\rvert)$

Table 1: Complexity of LOB operations using 4-Heap, with n the size of the largest heap.

(4) **Clearing Mechanism**: The market matches compatible buys and sells, determines the price of transaction, and clears the mathing orders out of the LOB. In a continuous market, clearing occurs at each time step after an agent acts. In a call market (also called *batch auction*), clears are invoked periodically.
(5) **Transaction Recording**: The market maintains a record of matched orders from previous time steps consisting of the matched price, quantity, and order IDs.

The market module is designed to work in conjunction with the LOB module, which handles order storage, matching, and execution. This modular design allows for potential modifications to market rules or order matching algorithms without altering the core market logic. The description below focuses on a continuous market for a single asset, but the LOB structure is designed to support call markets as well. MarketSim also supports extensions to other market types and multi-asset scenarios.

3.3 Limit Order Book

Our LOB is implemented by the 4-Heap data structure of Wurman et al. [43]. At any point, an order is classified as *matched* or *unmatched*. A continuous market, by definition, has only unmatched orders as matches are immediately transacted. The orders are organized in four heaps:

B_{in}	A min-heap of matched buy orders.
B_{out}	A max-heap of unmatched buy orders.
S_{in}	A max-heap of matched sell orders.
S_{out}	A min-heap of unmatched sell orders.

Let $V(H)$ denote the order price of the item at the top of heap H.

The LOB has to support four main operations:

(1) *Insert*: Place a new order into the appropriate heaps. Since the orders can be of any quantity, this may involve splitting the order or past orders between the *in* and *out* heaps.
(2) *Remove*: Update the heaps to reflect order removal. Our implementation speeds up this computation by removing unmatched orders in a lazy manner.
(3) *Quote*: The BID quote is $\max(V(S_{in}), V(B_{out}))$ and the ASK quote is $\min(V(S_{out}), V(B_{in}))$.
(4) *Clear*: Matched buy and sell orders are removed from the LOB and assembled into transactions.

Table 1 summarizes the operations, their complexity and runtime at different book sizes and order quantities.

3.4 Valuation Model

The valuation of the security for each agent is determined by two components: one common to all agents and another (optional) private to each agent. The common component, called the ***fundamental value***, represents the intrinsic economic value of the security. The private component reflects agent-specific factors bearing on a security's value, for example liquidity needs, risk preferences, and correlations with other (unmodeled) agent holdings.

The fundamental value is modeled using a mean-reverting stochastic process parameterized by the fundamental mean $\bar{f}$; the mean reversion parameter $\kappa \in [0, 1]$ which quantifies the speed at which the fundamental value reverts to its mean; and the shock variance $\sigma_s^2 \in [0, \infty)$ which impacts the zero-mean random Gaussian shock $\epsilon_t \sim \mathcal{N}(0, \sigma_s^2)$ applied to the fundamental at each time step t. The fundamental value f_t at time step t is recursively generated as follows, with $f_0 = \bar{f}$:

$$f_t = \kappa \bar{f} + (1 - \kappa) f_{t-1} + \epsilon_t, \tag{1}$$

In MarketSim, rather than computing f_t at each time step, the fundamental value is explicitly generated only when an agent interacts with the market. Since no agents arrive in most time steps, this *lazy* approach results in a large speedup compared to the naive fundamental generation.

$$f_t = (1 - \kappa)^{dt} f_{t_{\text{latest}}} + r\bar{f} \sum_{i=0}^{dt-1} (1 - \kappa)^i + \sum_{i=0}^{dt-1} (1 - \kappa)^i \epsilon_i \tag{2}$$

Agents are assumed to know this process and its parameters $\bar{f}$ and κ. Thus, based on an observed fundamental at time t they can estimate the final fundamental value:

$$\tilde{f}_T = (1 - \rho)\bar{f} + \rho f_t, \tag{3}$$

where $\rho = (1 - \kappa)^{T-t}$.

Private valuations are based on the model of Goettler et al. [15] and are parameterized by the maximum holding, $q_{\max} \in [0, \infty)$, and the private value variance, $\sigma_{pv}^2 \in [0, \infty)$. Each agent is assigned a vector Θ_i with $|\Theta_i| = 2q_{\max}$, of private values representing the marginal value of buying or selling an additional security. The vector components θ_i^k are drawn from $\mathcal{N}(0, \sigma_{pv}^2)$, and sorted in descending order to ensure decreasing marginal value of extra units.

$$\Theta_i = (\theta_i^{-q_{\max}+1}, \ldots, \theta_i^0, \theta_i^1, \ldots, \theta_i^{q_{\max}})$$

Given agent i currently holds q_i units of the security, their valuation of an additional unit is:

$$v_i^q = f_T + \begin{cases} \theta_i^{q_i+1} & \text{if buying} \\ \theta_i^{q_i} & \text{if selling} \end{cases}$$

The value of each agent's portfolio at the end of the simulation is a sum of their cash, their holdings' value ($q_T < 0$ if the agent is short), and sum of their private values

$$c_T + q_T f_T + \begin{cases} \sum_{j=1}^{q_T} \theta_i^j & q_T > 0 \\ 0 & q_T = 0 \\ \sum_{j=q_T+1}^{0} -\theta_i^j & q_T < 0 \end{cases} \tag{4}$$

3.5 dRL Wrappers

PyMarketSim provides Gym APIs for single-agent dRL. Fig. 2 shows how a learning agent interacts with the market mechanism and background traders. Typically, in a Gym setup, a learning agent engages with the environment through a *step* function, which takes the agent's action as input and returns the observation, reward, and episode termination status. From the learning agent's viewpoint, the background traders and the market mechanism are all part of the environment. At each time step, only the agents that have arrived can take actions and their actions collectively influence the current market state; until then, all other traders continue their trading activities. In particular, upon the learning agent's arrival, it receives the current market observation and selects an action based on this observation and its strategy. Once the action is communicated to the market, background traders carry on their trading activities until the learning agent arrives again. At this point, the effect of the action has been revealed, providing both the reward and a new observation of the current market. A transition tuple, which includes the observation, the action, the reward, and the next observation, is then formed. This information can be used by dRL to update the strategy. Note that by providing the transition information, our dRL wrapper is agnostic to specific RL algorithms.

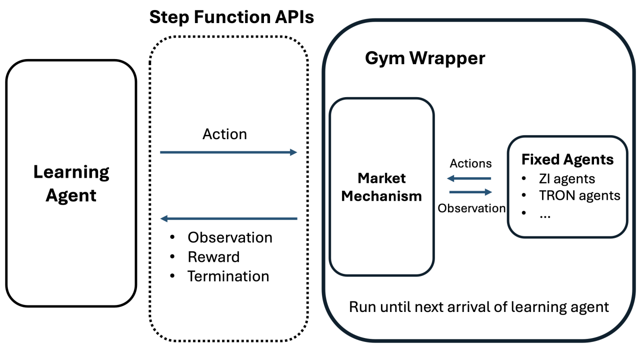

Figure 2: Deep RL wrapper workflow.

Our dRL wrapper is highly customizable, allowing for tailored reward functions, action spaces, and state (observation) spaces to accommodate various agent architectures, market scenarios, and dRL algorithms. For instance, the return for a learning agent is defined as the net cash at the end of a simulation, assuming liquidation of holdings at the final fundamental value. Realizing this return only at the end of a simulation provides a sparse reward signal, which presents challenges for learning. We can facilitate effective training by defining intermediate rewards. A natural choice in this instance is the change in cash plus valuation of holdings at the next arrival given the current state. Valuation is with respect to the final fundamental, which we can use for training even though it is not observable by the agent until the end of the run. It can be easily demonstrated that these intermediate rewards are guaranteed to sum to the final total return.

Additionally, beyond the ability to customize your own observation spaces, we offer several ready-made features that can be used as components of observations, including:

- time remaining $T - t$
- current fundamental value r_t , or a noisy observation
- current best BID and ASK prices in the order book (if any)
- agent's current holdings q
- agent's private values
- market summary statistics, such as volatility measures

4 Agents

PyMarketSim provides a library of agent classes, extensible by user implementations that extend the abstract agent base class. Agents are designated within roles, which dictate their arrival rates, valuation models, and the observations they get on each arrival.

An important role category is that of ***background trader***.[3] Background traders have private values as described in §3.4. On each arrival, background traders get an observation of the fundamental (exact or noisy), and also observe elements of market state, such as BID and ASK quotes and recent transaction prices. They may then submit an order to buy or sell a single unit of the security, replacing their existing order in the LOB, if any. Whether they buy or sell is determined by a fair coin flip.

Another canonical trader role is that of ***market maker*** (MM). MMs get the same kind of observation as background traders, but have no private values. Our MM agents submit ladders of unit buy and sell orders, as in the MM model of Chakraborty and Kearns [7]. More specifically, we follow the strategy described by Wah et al. [36], generating a ladder with $K > 0$ rungs spaced $\xi > 0$ price units apart.

The following sections describe two of the heuristic strategies we provide for background traders, as well as PyMarketSim's class of background traders derived via dRL.

4.1 Zero Intelligence

Zero Intelligence (ZI) agents [14] implement a simple strategy based on requesting a randomized surplus from their current valuation. Recall $\tilde{f}_T$ is the agent's estimate the value of the final fundamental, as computed by (3). θ denotes the agent's private values, and q its current holdings. The ZI agent draws a random offset, $s \sim \mathcal{U}(R_{\min}, R_{\max})$, according to its range parameters $R_{\min}$ and $R_{\max}$. It then submits a limit order at price p, requesting a surplus of s from its current valuation:

$$p = \tilde{f}_T + \begin{cases} \theta^{q+1} - s & \text{if assigned buy,} \\ \theta^{q} + s & \text{if assigned sell.} \end{cases} \tag{5}$$

Our ZI agents further include a threshold parameter $\eta \in [0, 1]$, by which they determine whether the currently available surplus is attractive enough to take immediately. Let p^* denote the currently available market price (BID if the agent is selling, ASK if buying). From (5), s is the surplus demanded by the ZI agent. If the agent can get a fraction η or better of its demanded surplus at the current market price:

$$\begin{cases} (\tilde{f}_T + \theta^{q+1}) - p^* > \eta s & \text{if assigned buy} \\ p^* - (\tilde{f}_T + \theta^{q}) > \eta s & \text{if assigned sell} \end{cases}$$

[3]We refer to it as a category because a simulation configuration can have multiple background trader roles, for example differing in arrival rate or parameters of the valuation model.

the agent places its limit order at p^* instead of p, resulting in an immediate transaction. Setting $\eta = 1$ is the same as ZI without this threshold decision.

4.2 HBL

Unlike ZI, HBL agents [13] not only utilize their own observations and private values but also exploit order book information. They estimate the probabilities of given limit prices transacting based on historical transactions and submit orders that maximize their expected surplus at the current valuation estimate. The HBL agents form their probability estimates by analyzing the outcomes of the last L trades, where L represents the agent's memory length and serves as a strategic parameter. Upon arrival at time t, the agent constructs a belief function $f_t(P)$. This function is designed to estimate the likelihood that placing an order at price P will lead to a successful transaction:

$$f_t(P) = \begin{cases} \frac{TBL_t(P)+AL_t(P)}{TBL_t(P)+AL_t(P)+RBG_t(P)} & \text{if assigned buy} \\ \\ \frac{TAG_t(P)+BG_t(P)}{TAG_t(P)+BG_t(P)+RAL_t(P)} & \text{if assigned sell} \end{cases} \tag{6}$$

In this context, we use specific abbreviations to describe different types of orders and their outcomes. T represents transacted orders, while R denotes rejected orders. A and B represent sell and buy orders, respectively. L and G describe orders with prices *less than or equal* or *greater than or equal* to P, respectively.

Our market model allows for order cancellations and maintains active orders in the order book, which complicates the definition of a rejected order. To address this, we introduce two time-based concepts: the grace period (τ_{gp}) and the alive period (τ_{al}) of an order. The grace period is defined as $\tau_{gp} = 1/\lambda_a$, where λ_a is the agent's arrival rate. The alive period (τ_{al}) of an order is measured from its submission until one of three timesteps depending on the corresponding event: the order transaction time (if it transacts), the order withdrawal time (if it is inactive), or the current time (if it is still active). We consider an order fully rejected only if its alive period exceeds the grace period ($\tau_{al} \geq \tau_{gp}$). If the alive period is shorter than the grace period, we treat the order as partially rejected, with the degree of rejection calculated as the τ_{al}/τ_{gp}. The belief function is initially defined only for prices observed within the agent's memory window. To extend this function across the entire price range, we employ cubic spline interpolation. For computational efficiency, we select a predetermined number of knot points and perform the interpolation between these points. The agent then determines its expected profit-maximizing price:

$$p_i^*(t) = \begin{cases} \arg\max_p (\tilde{f}_t + \theta_i^{q+1} - p) f_t(p) & \text{if assigned buy} \\ \arg\max_p (p - \theta_i^{q} - \tilde{f}_t) f_t(p) & \text{if assigned sell} \end{cases} \tag{7}$$

In certain specific scenarios, HBL agents temporarily adopt the behavior of ZI agents. This occurs at the start of a trading period when there are fewer than L transactions recorded or when one side of the order book is completely empty. However, these cases are infrequent, so reverting to ZI strategy during these brief periods does not significantly impact their overall performance. The behavior of HBL agents is further explored in [16] inside the PyMarketSim environment.

4.3 TRON Agents

Heuristic trading strategies, as described above, play an important role in agent-based finance studies. Strategies derived through dRL represent a qualitatively distinct class, which in our setting we call ***trained-response order networks***, or ***TRON agents***. TRON agents are drawn from a large space of candidate policies represented by neural networks, affording a much greater degree of expressivity compared to heuristic strategies. Because they are optimized for particular settings, TRON agents may be especially capable in those settings. There is also, however, a risk that TRON agents overfit to their training context, and due to their complexity they may be less interpretable than hand-designed heuristics.

The TRON background trading agents we employ map observed market state to action parameters (s, η), corresponding to the requested surplus and threshold parameters employed by ZI agents as described in §4.1. Unlike ZI, TRON agents specify a specific s rather than drawing from a range $[R_{\min}, R_{\max}]$. More significantly, its action is conditional on the agent's observations of the market. While other heuristic strategies, like HBL or adaptive extensions of ZI, do respond to market conditions, their responses are regular and limited. This contrasts with the general conditioning that a neural network representation provides.

Our TRON agent is implemented with the architecture from R2D2 [21] using the dueling DQN [39] and an LSTM [18] as its recurrent component. Fig. 3 depicts the network architecture. Its inputs are the features provided by our dRL wrapper, described in §3.5, plus an indicator for the side (buy or sell) the TRON agent has been assigned for this arrival. The output consists of the order parameters s and η. On each arrival, the TRON agent processes inputs according to current observations, generating then bids according to the ZI policy as defined by (5), substituting an executable order instead if the threshold condition applies.

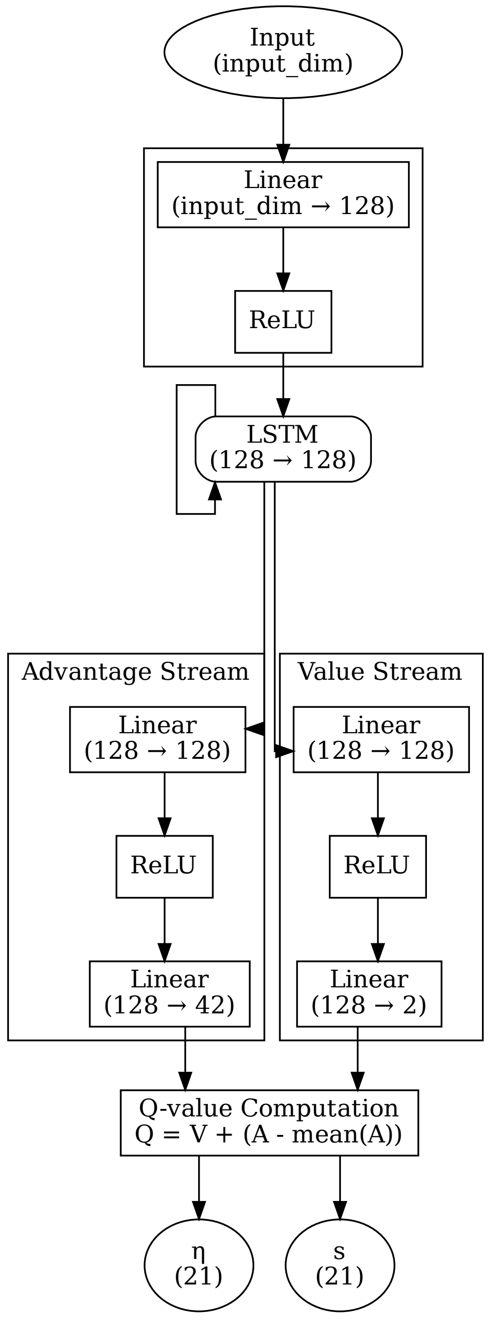

Figure 3: TRON neural network architecture.

5 Experiments

5.1 Operation Time

Since PyMarketSim is designed as an environment for deep learning experiments in financial markets, computational efficiency is a critical factor. To demonstrate the performance of our implementation, we conducted a series of preliminary experiments measuring the average runtime for order operations in the LOB.

We generated orders with random prices and sides (buy/sell). These orders were added to the LOB without performing any clear operations, allowing the LOB size to grow to match the number of orders. We performed 2×2 sets of experiments with respect to order sizes and the number of orders (equivalently LOB size): unit orders (quantity = 1) vs. orders with quantities drawn from a uniform distribution $\mathcal{U}(1, 10^3)$; small (10^3 orders) vs. large (10^6 orders) LOB size.

We present our results in Table 2. Compared to the results presented in Table 1 of Frey et al. [12], our average runtime is orders of magnitude lower than that of JAX-LOB (e.g., average time needed by JAX-LOB for the add operation is ~ 0.17 ms for an LOB capacity of 10^3) and we support larger total size for LOBs. These results highlight the following properties of PyMarketSim:

Efficiency: Even for large order books with 10^6 orders, the average operation time remains under 0.005 milliseconds, indicating high performance.

Scalability: The operation time increases only moderately (less than 2x) when moving from 10^3 to 10^6 orders, suggesting scalability.

Order complexity: Variable-quantity orders incur a slight performance penalty compared to unit orders, but the difference is relatively small (about 1.5x slower).

LOB size	Order Size	Average Operation Time (μs)
10^3	$q = 1$	1.89
10^3	$q \sim \mathcal{U}(1, 10^3)$	2.98
10^6	$q = 1$	3.40
10^6	$q \sim \mathcal{U}(1, 10^3)$	4.71

Table 2: Runtime of LOB operations in 4-Heap.

Given the difference in architecture between PyMarketSim and ABIDES [6], it is harder to provide a direct comparison between the runtime of the two. One possible benchmark would be to compare the average messages per second in ABIDES to the average number of agent and market actions per second in PyMarketSim: ABIDES processes approximately 29,000 messages per second while PyMarketSim can handle up to 300,000 LOB actions or 15,000 agent actions per second. This suggests that PyMarketSim is better suited to scenarios with fewer agents and a longer runtime. Another qualitative difference between PyMarketSim and ABIDES is noteworthy. One particular strength of PyMarketSim is its modularity: it can, in principle, simulate an arbitrary number of markets each trading any number of (potentially shared) securities; it also allows enhanced customization in handling the latency of agents instead of it being dependent on agent runtime as in ABIDES.

These performance characteristics make PyMarketSim well-suited for large-scale simulations and reinforcement learning experiments, where millions of market interactions may need to be simulated rapidly. The efficiency of the LOB operations allows users to focus on developing and testing sophisticated trading strategies without being constrained by computational limitations.

In comparing the performance of market simulators, it is important to acknowledge differences in the underlying rationales of the design choices of these systems. PyMarketSim is designed to enable efficient multiagent reinforcement learning for markets with a moderate total number of agents and assuming multiple machines are available. Inspired by Sample Factory [30] and IMPALA [10], PyMarketSim runs fast on CPU and can be run on an arbitrary number of nodes to increase the total simulations run via parallelization. One the other hand, it is evident that ABIDES is designed to efficiently handle a much larger number of agents whereas JAX-LOB emphasizes running a larger number of simulations on a single machine.

5.2 Simulation Settings

The environments in our experiments are characterized by several key parameters that define the market conditions. The values we use in our experiments are adopted from prior studies [36, 42] and described here. Settings that vary by environment are shown in Table 3.

(1) $N = 25$. Total number of agents in the market.
(2) $q_{\max} = 10$. Maximum holding, limiting the magnitude of an agent's position.
(3) $\bar{f} = 1 \times 10^5$. Mean fundamental value.
(4) $\kappa = .01$. Mean-reversion parameter, determining how quickly the fundamental value returns to its mean.
(5) $T = 2000$. Number of time steps in each simulation episode.
(6) λ (varies). Reentry rate, determining how frequently agents interact with the market.
(7) σ_s (varies). Shock variance of the fundamental.
(8) σ_{pv} (varies). Variance of the private values, representing the diversity of agent valuations.

Env	λ	σ_s^2	σ_{PV}^2	ZI	TRON
A	0.0005	1×10^6	5×10^6	106	114
B	.005	1×10^6	5×10^6	152	170
C	0.012	2×10^4	2×10^7	1259	1402

Table 3: Environment parameters and average profit for the two agent strategies.

5.3 Single-Agent RL

For each environment, we fixed 24 agents to play equilibrium ZI settings, and used dRL to derive a TRON agent. We trained our TRON agent using R2D2 [21], run for 3 million simulations. The results for the three environments are presented in Table 3. The equilibrium ZI settings are as reported in prior studies of these environments [36, 42]. The TRON agent consistently outperformed the baseline ZI agents: by 8%, 12%, and 11%, respectively in the three test environments.

5.4 Policy-Space Response Oracles

5.4.1 Overview. PSRO [3, 24] is a flexible framework for computing approximate Nash equilibria in complex games, based on simulation and dRL. The core idea of PSRO is to iteratively expand a restricted game by adding best responses to *some* (mixed) strategy profiles in the current restricted game. Specifically, PSRO proceeds as follows:

(1) Initialize the restricted game with a set of some policies (e.g., random policies) for each player;
(2) Compute an equilibrium over the restricted game with respect to a solution concept, yielding a *target profile*;
(3) For each player, compute a best response to the target profile using dRL;
(4) Add these best responses to the restricted game and repeat from step 2 until convergence or a computational budget is exhausted.

Depending on the solver used to extract a target profile, PSRO implements different approaches to strategy exploration. For example, using Nash equilibrium corresponds to the double oracle method [27], and using a uniform distribution corresponds to fictitious play [17]. Other choices such as projected replicator dynamics [24] or regularized replicator dynamics [38] have shown advantages for some games.

In our experiments with PyMarketSim, we apply PSRO to identify equilibrium strategies between two competing TRON agents, holding the other players fixed. This approach enables us to explore the strategic landscape of our simulated market, revealing trading strategies that are resilient against an adaptive of opponents.

5.4.2 Experiment. We use PSRO to train two TRON agents at once (along with 23 fixed ZI agents), for the environment labeled C in Table 3. Starting from the singleton comprising the equilibrium ZI strategy, we augment the set of available strategies by training a new TRON agent at each iteration. Let S_i denote the set of available strategies following PSRO iteration i. The training target for iteration i is defined by the equilibrium over two players from strategy set S_{i-1}, trading in this market environment along with the 23 fixed ZI agents. Let TRON_i denote the TRON agent trained at iteration i. Thus, $S_{i+1} = S_i \cup \{\text{TRON}_i\}$.

The results from our PSRO experiment are shown in Fig. 4. We measure convergence to equilibrium by an estimate of ***regret***: how much additional profit an agent could gain by deviating from the current candidate solution to an alternative strategy. Specifically, the regret at PSRO iteration i is estimated by the difference between profit of TRON_{i+1} and the profit of the equilibrium over S_i. For instance, the regret at iteration 0 is the difference in profit between TRON_1 and the ZI baseline when played along with the 24 fixed ZI. From Table 3 this is $1402 - 1259 = 143$.

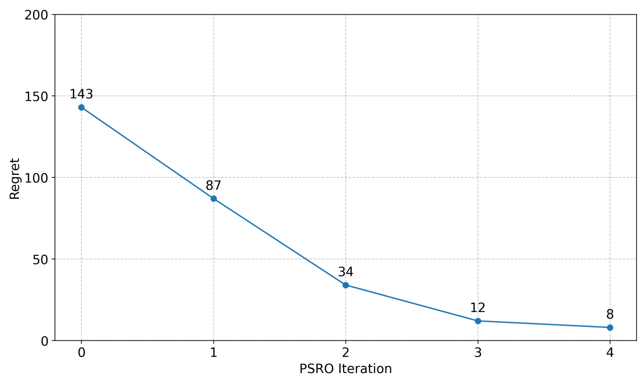

Figure 4: PSRO regret plot for the two TRON agents.

Once the first TRON agent is introduced, the equilibria computed by PSRO comprise mixtures over TRONs alone, assigning zero probability to the ZI strategy. As can be seen Fig. 4, the gains from training new TRON agents produces diminishing returns, approaching negligible increases after just a few iterations. That the TRON agents stop showing improved performance so quickly may be due to the limited amount of interaction between the two learners, who represent only 2/25 of the agents in the market. Allowing the ZI agents to recalibrate their parameters after each PSRO iteration would possibly affect this convergence pattern, but has not been explored here.

6 Conclusions and Future Work

In this paper, we introduced PyMarketSim a flexible and efficient financial market simulation environment designed for training and evaluating trading agents using deep reinforcement learning techniques. Our framework provides a comprehensive set of tools for simulating continuous double auction markets, incorporating key elements such as private valuations, asymmetric information, and a high-performance LOB mechanism.

We demonstrated the versatility of PyMarketSim through experiments in both single-agent and multi-agent reinforcement learning settings. Our results showcase the potential for developing sophisticated trading strategies and exploring complex market dynamics. The introduction of TRON agents extends the traditional Zero Intelligence model, providing a more realistic and adaptive background trader for market simulations.

While PyMarketSim offers a solid foundation for financial market research, there are several exciting directions for future work:

(1) **Options Trading**: Extending the framework to support options trading [33] would significantly broaden its applicability. This could involve:
    - Implementing various option types (e.g., European, American) and their associated pricing mechanisms.
    - Exploring the use of Neural Differential Equations [23] to recover Black-Scholes [4] pricing within the PyMarketSim environment, potentially leading to new insights in option pricing dynamics.

(2) **Enhanced Market Complexity**: Inspired by recent advances in generative AI, such as Generative Agents [29], we could create a more intricate and realistic market environment by:
    - Simulating a virtual world with social media and news reports that influence market dynamics.
    - Implementing agent interactions beyond direct trading, such as information sharing or strategy imitation.
    - Modeling the impact of macroeconomic events and policy changes on market behavior.

(3) **Multi-Asset Markets**: Expanding PyMarketSim to handle multiple interconnected assets would allow for the study of portfolio management strategies and cross-asset dependencies.

(4) **Market Microstructure Analysis**: Developing tools for in-depth analysis of market microstructure effects, such as order flow imbalance or liquidity provision, could provide valuable insights for both researchers and practitioners.

These enhancements would not only increase the realism and complexity of the simulated markets but also open up new avenues for research in areas such as behavioral finance, market impact studies, and the intersection of AI and financial economics.

In conclusion, PyMarketSim provides a powerful platform for studying financial markets through the lens of artificial intelligence and agent-based modeling. As the field continues to evolve, we believe this tool will play a crucial role in advancing our understanding of market dynamics and in developing the next generation of intelligent trading systems. We invite the research community to build upon this framework, contributing to its growth and applicability across various domains of financial research and practice.

References

[1] Selim Amrouni, Aymeric Moulin, Jared Vann, Svitlana Vyetrenko, Tucker Balch, and Manuela Veloso. 2021. ABIDES-Gym: Gym environments for multi-agent discrete event simulation and application to financial markets. In *2nd ACM International Conference on AI in Finance.* 30:1–30:9.
[2] Leo Ardon, Nelson Vadori, Thomas Spooner, Mengda Xu, Jared Vann, and Sumitra Ganesh. 2021. Towards a fully RL-based market simulator. In *2nd ACM International Conference on AI in Finance.* 7:1–7:9.
[3] Ariyan Bighashdel, Yongzhao Wang, Stephen McAleer, Rahul Savani, and Frans A. Oliehoek. 2024. Policy space response oracles: A survey. In *33rd International Joint Conference on Artificial Intelligence.* 7951–7961.
[4] Fischer Black and Myron Scholes. 1973. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy* 81 (1973), 637–654.
[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye

Wanderman-Milne, and Qiao Zhang. 2018. JAX: Composable transformations of Python+NumPy programs. http://github.com/google/jax

[6] David Byrd, Maria Hybinette, and Tucker Hybinette Balch. 2020. ABIDES: Towards high-fidelity multi-agent market simulation. In *34th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 11–22.

[7] Tanmoy Chakraborty and Michael Kearns. 2011. Market making and mean reversion. In *12th ACM Conference on Electronic Commerce* (San Jose). 307–314.

[8] Dave Cliff. 1997. Minimal-intelligence agents for bargaining behaviors in market-based environments. *Hewlett-Packard Labs Technical Reports* (1997).

[9] Dave Cliff. 2009. ZIP60: Further explorations in the evolutionary design of trader agents and online auction-market mechanisms. *IEEE Transactions on Evolutionary Computation* 13 (2009), 3–18.

[10] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *35th International Conference on Machine Learning*. 1407–1416.

[11] J. Doyne Farmer, Paolo Patelli, and Ilija I. Zovko. 2005. The predictive power of zero intelligence in financial markets. *Proceedings of the National Academy of Sciences* 102 (2005), 2254–2259.

[12] Sascha Yves Frey, Kang Li, Peer Nagy, Silvia Sapora, Chris Lu, Stefan Zohren, Jakob Foerster, and Anisoara Calinescu. 2023. JAX-LOB: A GPU-Accelerated limit order book simulator to unlock large scale reinforcement learning for trading. In *4th ACM International Conference on AI in Finance*. 583–591.

[13] Steven Gjerstad and John Dickhaut. 1998. Price formation in double auctions. *Games and Economic Behavior* 22, 1 (1998), 1–29.

[14] Dhananjay K. Gode and Shyam Sunder. 1993. Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality. *Journal of Political Economy* 101, 1 (1993), 119–137.

[15] Ronald L. Goettler, Christine A. Parlour, and Uday Rajan. 2009. Informed traders and limit order markets. *Journal of Financial Economics* 93 (2009), 67–87.

[16] Anri Gu, Yongzhao Wang, Chris Mascioli, Mithun Chakraborty, Rahul Savani, Theodore Turocy, and Michael P. Wellman. 2024. The Effect of Liquidity on the Spoofability of Financial Markets. In *5th ACM International Conference on AI in Finance*.

[17] Johannes Heinrich, Marc Lanctot, and David Silver. 2015. Fictitious self-play in extensive-form games. In *32nd International Conference on Machine Learning*. PMLR, 805–813.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780.

[19] Konark Jain, Nick Firoozye, Jonathan Kochems, and Philip Treleaven. 2024. *Limit order book simulations: A review.* Technical Report. University College London. Available at SSRN, 4745587.

[20] Joseph Jerome, Leandro Sánchez-Betancourt, Rahul Savani, and Martin Herdegen. 2022. Model-based gym environments for limit order book trading. (2022). arXiv:2209.07823

[21] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. 2018. Recurrent experience replay in distributed reinforcement learning. In *6th International Conference on Learning Representations*.

[22] Michaël Karpe, Jin Fang, Zhongyao Ma, and Chen Wang. 2020. Multi-agent reinforcement learning in a realistic limit order book market simulation. In *1st ACM International Conference on AI in Finance*. 30:1–7.

[23] Patrick Kidger. 2022. On Neural Differential Equations. (2022). arXiv:2202.02435

[24] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. 2017. A unified game-theoretic approach to multiagent reinforcement learning. *Advances in Neural Information Processing Systems* 30 (2017).

[25] Blake LeBaron. 2000. Agent-Based Computational Finance: Suggested Readings and Early Research. *Journal of Economic Dynamics and Control* 24 (2000), 679–702.

[26] Iwao Maeda, David DeGraw, Michiharu Kitano, Hiroyasu Matsushima, Hiroki Sakaji, Kiyoshi Izumi, and Atsuo Kato. 2020. Deep reinforcement learning in agent based financial market simulation. *Journal of Risk and Financial Management* 13, 71 (2020), 1–17.

[27] H Brendan McMahan, Geoffrey J Gordon, and Avrim Blum. 2003. Planning in the presence of cost functions controlled by an adversary. In *20th International Conference on Machine Learning*. PMLR, 536–543.

[28] R. G. Palmer, W. Brian Arthur, John H. Holland, Blake LeBaron, and Paul Tayler. 1994. Artificial economic life: A simple model of a stockmarket. *Physica D: Nonlinear Phenomena* 75 (1994), 264–274.

[29] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[30] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. 2020. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In *37th International Conference on Machine Learning*. PMLR, 7652–7662.

[31] S. Phelps, M. Marcinkiewicz, S. Parsons, and P. McBurney. 2006. A novel method for automatic strategy acquisition in N-player non-zero-sum games. In *5th International Joint Conference on Autonomous Agents and Multi-Agent Systems*. Hakodate, 705–712.

[32] L. Julian Schvartzman and Michael P. Wellman. 2009. Stronger CDA strategies through empirical game-theoretic analysis and reinforcement learning. In *8th International Conference on Autonomous Agents and Multi-Agent Systems*. 249–256.

[33] Megan Shearer, Gabriel Rauterberg, and Michael P. Wellman. 2023. Learning to manipulate a financial benchmark. In *4th ACM International Conference on AI in Finance*. 592–600.

[34] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press, Cambridge, Massachusetts.

[35] Elaine Wah and Michael P. Wellman. 2013. Latency Arbitrage, Market Fragmentation, and Efficiency: A Two-Market Model. In *14th ACM Conference on Electronic Commerce*. 855–872.

[36] Elaine Wah, Mason Wright, and Michael P Wellman. 2017. Welfare effects of market making in continuous double auctions. *Journal of Artificial Intelligence Research* 59 (2017), 613–650.

[37] Xintong Wang, Christopher Hoang, Yevgeniy Vorobeychik, and Michael P. Wellman. 2021. Spoofing the limit order book: A strategic agent-based analysis. *Games* 12, 2 (2021), 46.

[38] Yongzhao Wang and Michael P. Wellman. 2023. Regularization for Strategy Exploration in Empirical Game-Theoretic Analysis. (2023). arXiv:2302.04928

[39] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1995–2003.

[40] Michael P. Wellman. 2011. *Trading Agents*. Morgan and Claypool.

[41] Michael P. Wellman, Karl Tuyls, and Amy Greenwald. 2024. Empirical Game-Theoretic Analysis: A Survey. (2024). arXiv:2403.04018

[42] Mason Wright and Michael P Wellman. 2018. Evaluating the stability of non-adaptive trading in continuous double auctions. In *17th International Conference on Autonomous Agents and Multiagent Systems*. 614–622.

[43] Peter R Wurman, William E Walsh, and Michael P Wellman. 1998. Flexible double auctions for electronic commerce: theory and implementation. *Decision Support Systems* 24, 1 (Nov. 1998), 17–27.

[44] Zhiyuan Yao, Zheng Li, Matthew Thomas, and Ionut Florescu. 2024. Reinforcement Learning in Agent-Based Market Simulation: Unveiling Realistic Stylized Facts and Behavior. *arXiv:2403.19781* (2024).