BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE IN ENGLISH

DIPLOMA THESIS

Advancing Market Integrity: A Proximal Policy Optimization Approach to Detect Spoofing and Layering in Algorithmic Trading

Supervisor
Lecturer Professor, PhD Diana-Lucia Miholca

Author
Iulia-Diana Groza

2024

ABSTRACT

Algorithmic trading has revolutionized financial markets by enabling rapid and complex trading strategies. However, it has also introduced new challenges in maintaining market integrity, particularly through manipulative practices like spoofing and layering. These tactics deceive market participants by placing and then quickly canceling large orders to create artificial price movements, undermining the fairness and transparency of trading activities.

This thesis presents a novel approach to detecting spoofing and layering using Proximal Policy Optimization (PPO), a state-of-the-art reinforcement learning algorithm. By leveraging historical Level 3 Limit Order Book (LOB) data from the LUNA flash crash in May 2022, we develop and implement a PPO-based model that can identify these manipulative behaviors in real-time.

The core contributions of this work include the development of a PPO policy network tailored for spoofing detection, the integration of this model into a web application called spoof.io, and a comprehensive evaluation of its performance. We detail the preprocessing and feature engineering steps, the creation of a market environment simulation, and the iterative process of training and hyperparameter tuning.

Our experimental results demonstrate the efficacy of the PPO model in detecting spoofing attempts, showing promising potential for enhancing market surveillance systems. This research not only advances the field of market manipulation detection but also provides a practical tool for real-time monitoring and maintaining market integrity.

Keywords: Algorithmic trading, Spoofing detection, Proximal Policy Optimization, Reinforcement learning, Market manipulation, Market integrity.

Contents

Chapter 1

Introduction

The advent of algorithmic trading has transformed financial markets, introducing significant efficiencies but also new challenges in maintaining market integrity. Among the most pressing issues are the manipulative practices of spoofing and layering, where traders deceive other market participants by placing and then quickly canceling large orders. These tactics can lead to artificial price movements and undermine the trust in market fairness.

This thesis explores the application of Proximal Policy Optimization (PPO), a reinforcement learning technique, to detect and mitigate spoofing and layering in algorithmic trading. By leveraging the capabilities of PPO, we aim to develop a robust detection system that can identify these manipulative behaviors in real-time, thereby enhancing market integrity.

The core of this study involves the development and implementation of a PPO-based model specifically tailored for the detection of spoofing and layering. This model is integrated into a web application, spoof.io, which provides a practical tool for monitoring trading activities and identifying potential manipulative behaviors. The application utilizes historical Level 3 Limit Order Book (LOB) data from a notable market event—the LUNA flash crash of May 2022—offering a realistic and challenging testbed for our approach.

In addition to the development of the PPO model, this thesis includes a comprehensive literature survey on existing methods for spoofing detection, highlighting the strengths and limitations of various approaches. We also detail the hyperparameter tuning process to optimize the performance of our PPO model and provide a thorough analysis of the experimental results.

Ultimately, this work aims to contribute to the field of market surveillance by providing an innovative solution for detecting sophisticated market manipulation strategies, thereby supporting fair and transparent trading practices.

Chapter 2

Algorithmic Trading and Market Integrity

The objective of this chapter is to present the terminology required for a better understanding of the main topic of the thesis, specifically providing a brief theoretical insight on algorithmic trading. Furthermore, we explore how *spoofing* and *layering* are performed, and how such manipulative tactics can deeply impact financial markets.

2.1 Introduction to Algorithmic Trading

Undoubtedly, the landscape of modern financial markets has undergone major transformations in recent decades due to the widespread adoption of algorithmic strategies. In general, *algorithmic trading* (or widely referred to as *automated trading* or *black-box trading*) represents the execution of trades at precise moments by leveraging the use of computer programming. Put another way, algorithmic trading improves the *liquidity* of markets by ruling out the involvement of human emotions and execution delays specific to *traditional market-making*. In their seminal work, Hendershott & Riordan (2013) provide a comprehensive examination of algorithmic trading and its implications for market liquidity, by studying the impact of algorithmic trading strategies on market dynamics [HR13].

In the context of market manipulation, we aim to focus specifically on *high-frequency trading* (HFT), which is characterised by speedy execution, when it comes to buying or selling securities. HFT is applied not only in stock markets, but in exchanging stock options and futures as well [Dur10].

Thanks to recent advancements in hardware development and the impact of *Field Programmable Gate Arrays* (FPGAs) on ultra-low latency, even a couple of *nanoseconds* could make the difference between *profit* and *loss*.

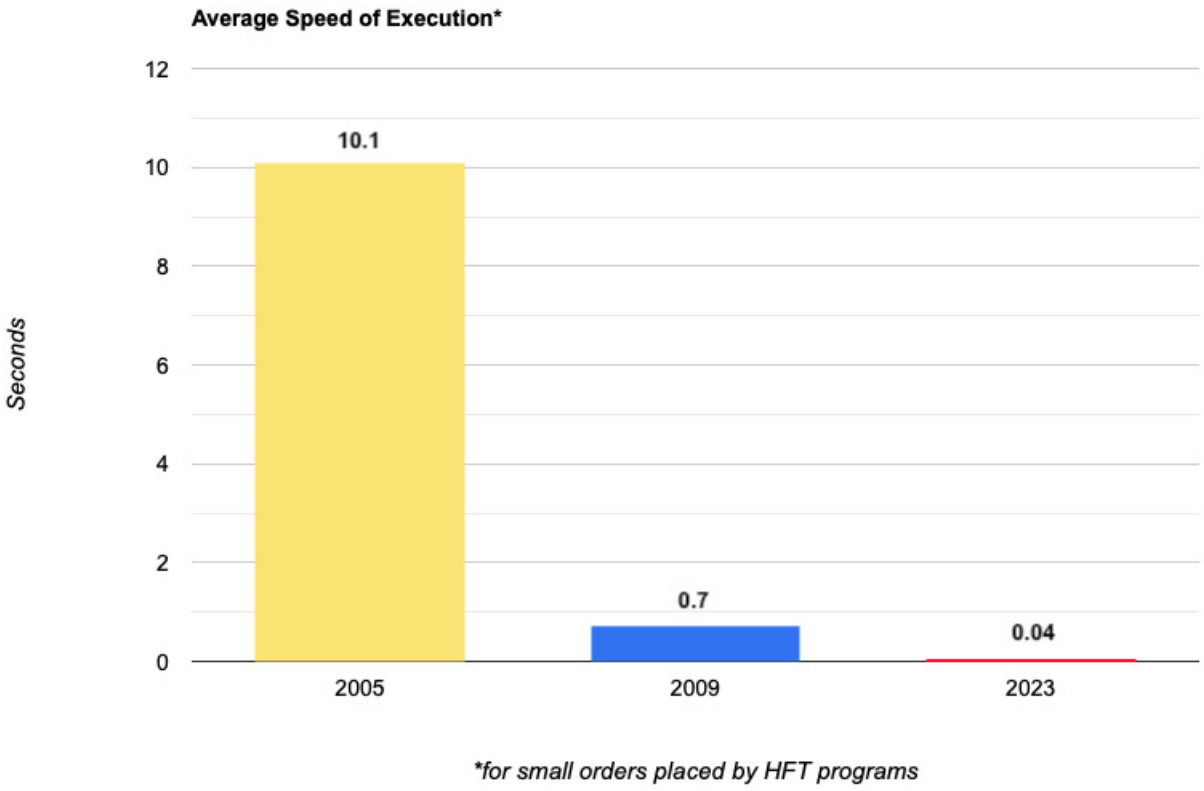


Figure 2.1: The average speed of execution for small orders in HFT has come to the realm of nanoseconds thanks to the development of ultra-low latency systems.

Given how ultra-low latency adversely affects market dynamics, we now introduce the fundamental concept in HFT: the *order*. Put succinctly, a trade cannot occur without a previously placed order to initiate it.

In his article "Limit Order Books", Martin D. offers an ample and insightful definition for orders, as well as for fundamental terms that are frequently used in the financial literature [MDGH13]:

Definition 2.1.1 (Order) *An order $x = (p_x, \omega_x, t_x)$, submitted at time t_x, with price p_x, and size $w_x > 0$ (respectively, $w_x < 0$), is a commitment to sell (respectively, buy) up to $|\omega_x|$ units of the traded asset at a price no less than (respectively, no greater than) p_x.*

Definition 2.1.2 (Bid Price) *The bid price at time t is the highest stated price among active buy orders at time t,*

$$b(t) := \max_{x \in B(t)} p_x. \tag{2.1}$$

Definition 2.1.3 (Ask Price) *The ask price at time t is the lowest stated price among active sell orders at time t,*

$$a(t) := \min_{x \in A(t)} p_x. \tag{2.2}$$

Definition 2.1.4 (Mid Price) *The mid price at time t is*

$$m(t) := [a(t) + b(t)]/2. \tag{2.3}$$

Definition 2.1.5 (Bid-Ask Spread) *The bid-ask spread at time t is*

$$s(t) := a(t) - b(t). \tag{2.4}$$

It is important for an investor to know how to leverage the placement of two major order types in stock trading: *market orders* and *limit orders*.

A *market order* involves *buying* or *selling* a security immediately, the price of the transaction being strongly linked to the time of its execution (different from submission). This implies that the price at submission time might (at most times) deviate from the price at execution time; price remains unchanged only when the *bid/ask* price is exactly at the last traded price. Therefore, with immediate execution, market orders are more aggressive - volatility increases drastically, especially for investments with fewer shares on the market or smaller trade volumes.

Pushing the market in the opposite direction, *limit orders* (sometimes known as *pending orders*) are characterised by total price control, coming with a specific set of instructions on the execution of the trade. Investors specify the maximum *bid* price or the minimum *ask* price for their stocks. The brokerage will execute the order only if the price of the financial instrument aligns with the specified bounds; otherwise, the order will be left *unexecuted*. Interestingly, the conditions imposed in limit orders can even lead to partial orders (only a part of the shares will be traded), as the price of the investment can modify *mid-order*. In the next sub-chapter, we will explore how the less-aggressive nature of limit orders is exploited to produce "artificial" shifts and influence the stock markets.

Since market manipulation most commonly occurs via limit orders, as previously indicated, our subsequent discussion will elaborate on basic order dynamics: how limit orders are stored, processed and amended. Most exchanges make use of *Central Limit Order Books* (CLOBs), or simply known as *Limit Order Books* (LOBs), to execute limit orders. LOBs are transparent, real-time, anonymous, and low-cost-in-execution systems that utilise *order books* and *matching engines* to map bid/ask orders of investors based on *price-time* priority. Considering a financial instrument which investors place orders for, the best market consists of mapping the highest bid offer to the lowest ask offer.

Martin D. offers the following definition for an LOB [MDGH13]:

Definition 2.1.6 (Limit Order Book) *An LOB $L(t)$ is the set of all active orders in a market at time t.*

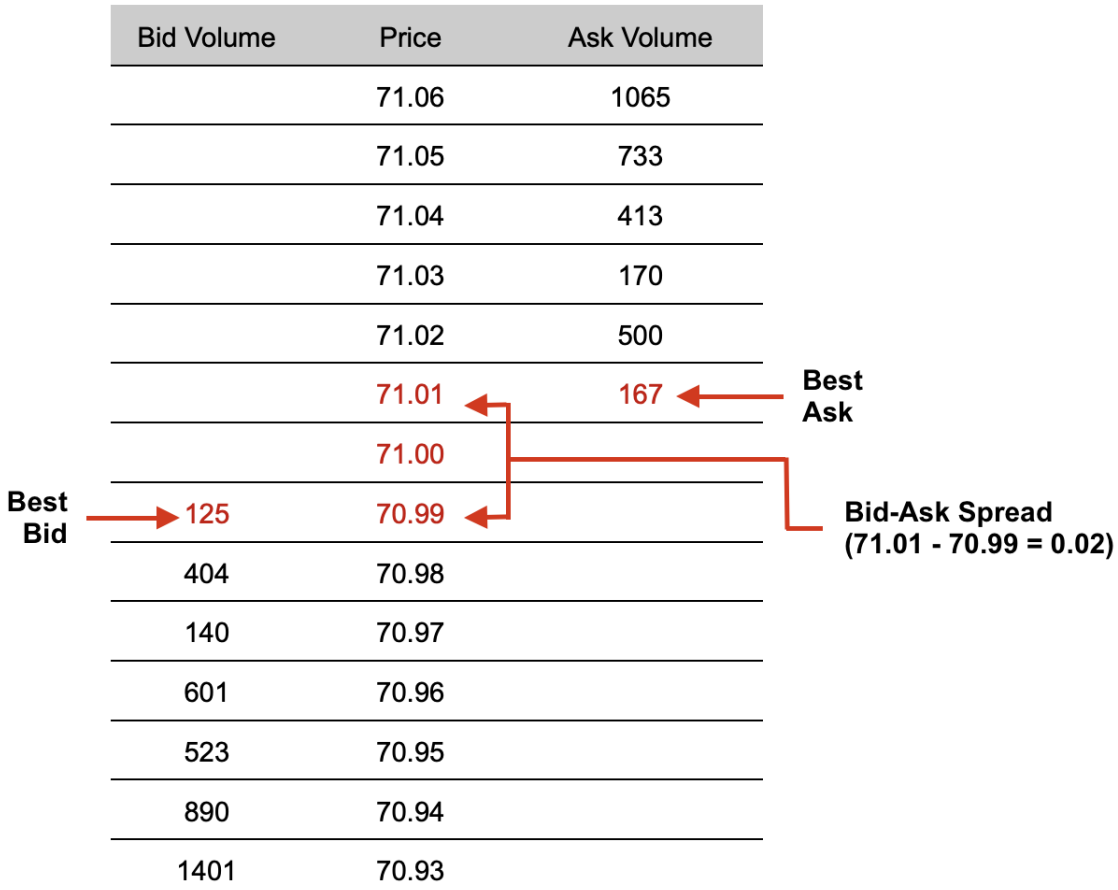


Figure 2.2: Graphical example of a 15-level deep LOB on a particular financial instrument. It depicts how traders judge liquidity and how they arrive at the bid-ask spread.

We continue with defining three crucial concepts for understanding CLOBs: *liquidity*, *price discovery*, and *depth of market*.

Liquidity refers to the degree to which a security can be easily traded in the market at a price reflecting its intrinsic value. Put another way, liquidity is the efficiency of converting an asset into ready cash without affecting its market price - the most liquid security is cash itself.

The most standard way of measuring liquidity is using the *current ratio* formula:

Definition 2.1.7 (Current Ratio) *The current ratio is a liquidity ratio measuring the capability of an investor of having enough resources to meet their short-term obligations:*

$$current\_ratio = \frac{current\_assets}{current\_liabilities}. \tag{2.5}$$

Price discovery denotes the (explicit or deduced) process in which buyers and sellers establish the fair price of a financial instrument in the market, at a given time.

It implies conducting market research analysis, by evaluating supply and demand, environmental, geopolitical and socioeconomic factors.

Depth of Market (DOM) refers to the volume of orders pending to be transacted for a particular security at different price levels - it is the overall level (or breadth) of open orders.

A large order can significantly impact DOM and liquidity. If a large buy order enters the market, it can exhaust the available sell orders at lower prices, increasing the price as it fulfills higher-priced sell orders. Conversely, a large sell order can fulfill all the buy orders at higher prices, driving the price down as it starts matching with lower-priced buy orders. This impact on price through the supply and demand balance is a direct outcome of the market's liquidity and depth.

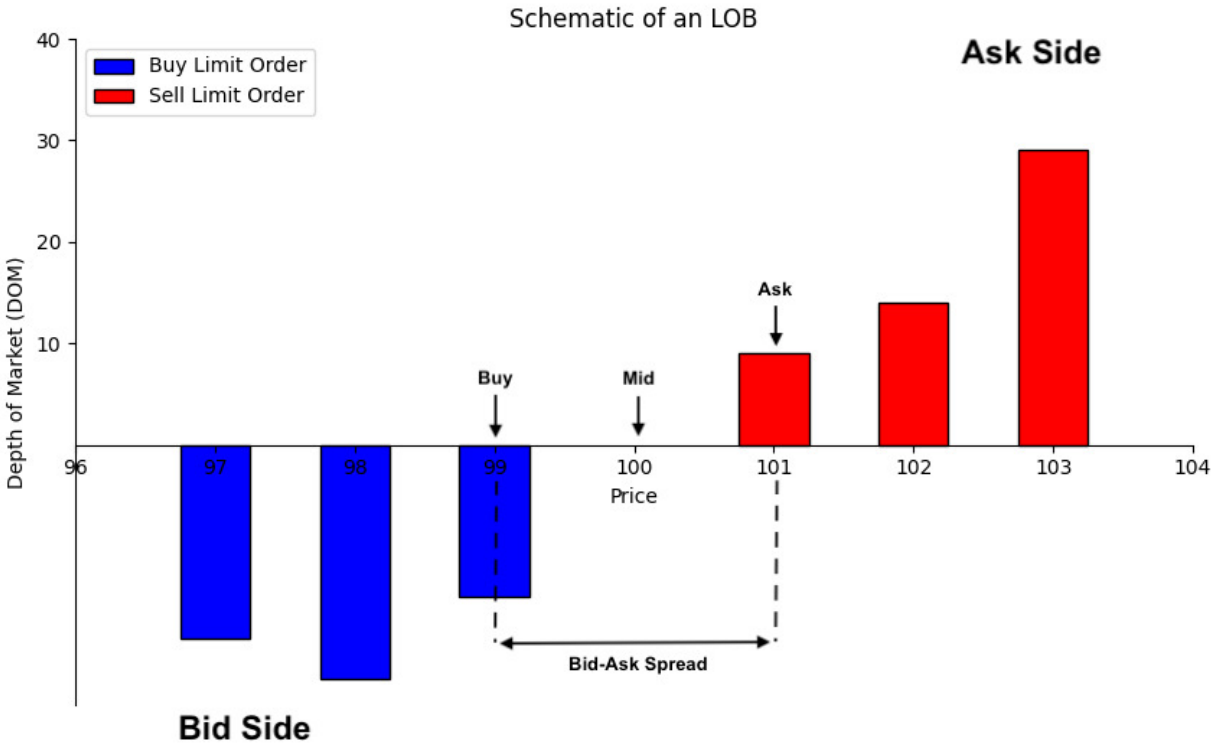


Figure 2.3: Schematic of an LOB

While we already covered the fundamentals of order placement, it is essential to acknowledge the mechanisms for order modification and cancellation in an LOB. In terms of *modification*, traders are generally allowed to update the limit price or the quantity of an existing order via the trading platform of the exchange. Modifications are generally permitted until the execution has begun, depending on the trading platform or exchange. For instance, if a buy limit order is partially filled,

the quantity or price may be adjusted for the remaining unfilled portion. As a consequence, after an order has been modified, the investor should be aware about the impact of such operation on the LOB - the priority of the order in the LOB might be affected. For example, if the investor increases (respectively, decreases) the buy (respectively, ask) price of the security, the order might shift its position in the order book, as it becomes more competitive. Changing the quantity of an order could also impact both the priority in the LOB and the likelihood of execution.

Order *cancellation* in trading can occur for various strategic reasons. Cancellation for manipulative intents will be covered in a later chapter; however, traders might *legitimately* cancel limit orders because of shifting market conditions (e.g. news events affecting stock prices, market moving against their strategy). The impact of cancellations on LOBs includes the change in available liquidity, bid-ask spread and perceived market depth.

LOBs can be classified into different *levels*, based on the amount of information they provide:

1. **Level 1** - It encapsulates basic market data, such as:
    - *Bid price*;
    - *Bid size*;
    - *Ask price*;
    - *Last price*;
    - *Last size*.
2. **Level 2** - Additionally to Level 1 data, it doesn't provide just the highest bid and lowest offer, but also bids and offers at other prices:
    - *Highest bid prices*;
    - *Bid sizes*;
    - *Lowest ask prices*;
    - *Ask sizes*.
3. **Level 3** - It provides even deeper information than Level 2 data. Level 3 data refers to non-aggregated bids and asks placed by individual market makers. A Level 3 data feed would include every individual bid and ask, including *time series*.

LOB snapshots can be procured from various stock & crypto exchanges or trading platforms, via *WebSocket feeds* (for real-time data), or *REST APIs* (for historical data). The implementation proposed in this thesis will make use of *Level 3 LOB data*, to take full advantage of temporal information.

2.2 Challenges in Market Integrity: Spoofing and Layering

With the rise of algorithmic trading and the automation of financial markets, there is no doubt that the market becomes more and more exposed to major risks of fraud and exploitation. Affecting market integrity takes multiple forms, including market manipulation, insider trading, and short selling, all of them causing ample market destabilization due to their complex and ever-evolving nature. This thesis targets the challenges brought by market manipulation, particularly by two of its widely spread forms: spoofing and layering.

Market manipulation is defined as a set of "actions intended to cause an artificial movement in the market price, so as to make a profit or avoid a loss" [AAD+16]. Therefore, it is easy to justify why such cases of misconduct erode the confidence of investors and challenge the perception of market stability.

The Dodd-Frank Wall Street Reform and Consumer Protection Act of 2010 ("Dodd-Frank Act") defines *spoofing* in the realm of algorithmic trading as "bidding or offering with the intent to cancel the bid or offer before execution" [Dod10]. A trader places multiple large limit orders on a security with no intention of executing them. This is performed with the aim of artificially decreasing (respectively, increasing) the price of an asset, with the later intent of placing the actual buy (respectively, sell) order after cancellation, giving the false impression to other traders that the security is "on-demand".

Layering is a particular case of spoofing, which consists of placing multiple *non-bona fide* orders at *multiple price tiers*, in contrast to spoofing, where orders are entered only at the top of the order book.

Spoofing and layering often cause extreme volatility and rapid price movements. In such cases, they largely contribute to flash crashes. A *flash crash* is an event in electronic markets, where prices of a financial instrument *drop* dramatically, but then they immediately *rebound*. Flash crashes are *sudden* and short-lived: at the end of the day, it appears as if the crash had never happened.

While some methods are relatively straightforward (e.g. spreading *fake news*), spoofing and layering, take more subtle forms, and the methods used by malicious investors are constantly evolving. Additionally, in some cases, it is difficult to decide whether an investor has the intention of causing harm (e.g. a company buying its own shares to rise their price). This leads to the complexity of developing an efficient system that detects spoofing and layering in all of its shapes, and not misclassify peculiar cases of legal practices.

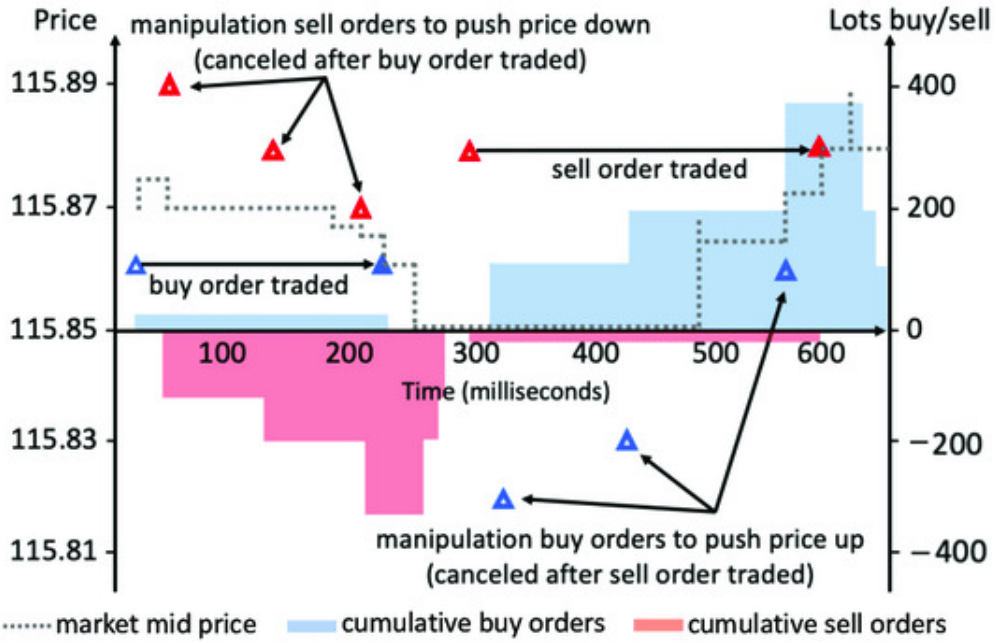


Figure 2.4: Illustrating an example of spoofing. Source: UK Financial Conduct Authority Final Notice 2013 [WHVW21]. "0.6 s, demonstrating how quickly and effectively such manipulation behavior can affect the market and profit from spoofed belief."

2.3 Case Studies of Notable Market Events

In his seminal work, *Spoofing, Market Manipulation, and the Limit-Order Book*, J. Montgomery highlights two notorious cases involving spoofing and layering allegations [Mon16], with those that pleaded guilty for the criminal actions being sentenced to up to *five years of imprisonment* and paying forfeits of as much as *US$300.000*.

One pivotal event revolves around Michael Coscia, who in November 2015 became the first person convicted of spoofing, under the *Dodd-Frank Act*. M. Coscia placed large "bait" orders for futures contracts on various commodities, through exchanges from the United States and the United Kingdom. This lead to a net profit for Mr. Coscia of *US$1.4 million* over a span of ten weeks as part of six attempts of spoofing.

In the most notable case of layering, Aleksandr Milrud lead a complex scheme for trading US securities. Mr. Milrud recruited and shared profits with online traders from China and Korea, who placed and then quickly cancelled HFT buy and sell orders at multiple price tiers, yielding net profits of as much as *US$600,000* in a day. This accumulated in net monthly profits between *US$1 million* and *US$50 millions*.

As part of our study, we aim to analyse LOB data from flash crashes, due to the high frequency of spoofing and layering attempts during such events. Specifically, we conduct our research on the flash crash of *LUNA*, from May 2022. Over a span of

just three days, *Terra*, the third-largest cryptocurrency ecosystem following Bitcoin and Ethereum, experienced a dramatic collapse, erasing *US$50 billion* in market valuation [LMS23].

2.4 Ethical and Legal Considerations

As highlighted in the legal cases presented in the previous chapter 2.3, the realm of financial trading is not only governed by economic principles, but also operates within a framework of ethical and legal standards. It is essential to implement and continually revise regulations to diminish the systematic losses produced by information asymmetry sourced from market manipulation.

In *"Principles of Financial Regulation"*, John Armour covers multiple aspects regarding the regulation of market manipulation, both in the European Union and the United States [AAD+16]. The integrity of financial markets hinges on the equitable treatment of all market participants. Unethical practices, such as market manipulation through spoofing and layering, undermine this fairness and can lead to significant distortions in market dynamics. Moreover, the lack of transparency in certain trading activities poses a threat to the trust essential in financial markets. This is particularly critical for retail investors, who might lack the resources to identify and navigate deceptive market practices effectively.

As mentioned in subchapter 2.2, legal frameworks play a vital role in regulating market activities and deterring unethical behavior. The legislation in both EU and US is designed to prevent market manipulation and promote a transparent trading environment. As noted in a report by Steel Eye [Steon], besides the *Dodd-Frank Act* [Dod10], market manipulation in US rules under multiple laws, including:

- *Commodity Exchange Act (CEA) (Section 4c(a)(5)(C))* [CEA13];
- *Securities Exchange Act of 1934 10(b)* [SEA34];
- *FINRA Rule 2020* [FIN08].

Regarding the UK and EU legislation, the *Market Abuse Regulation* (MAR) [MAR14] encompasses regulations related to insider trading, the illegal disclosure of privileged information, and the manipulation of markets.

Chapter 3

Reinforcement Learning and Optimization Methods

This chapter aims to cover fundamental *Reinforcement Learning* (RL) concepts that can be applied in high-frequency trading, specifically in market manipulation detection. We will dive deeper into how *Proximal Policy Optimization* (PPO) is applied to potentially solve market integrity issues.

3.1 Reinforcement Learning Mechanisms

It is no secret that in recent years, the interest in *Machine Learning* (ML) renewed thanks to exponential advancements and its extension to a broad range of fields. In less than a decade, ML has become an integrating part of almost every aspect of our lives: education, business & marketing automation, our social and personal life, and ML will only increase its potency in the future, especially with the recent rise of *Generative AI* (GenAI) and the race for developing the first instance of *Artificial General Intelligence* (AGI). As of the time of writing, creating AGI is the primary objective of pioneering AI research companies such as *OpenAI* [Ope24], *Google DeepMind*, and *Anthropic*.

The three main ML *paradigms* are: *Supervised Learning* (SL), *Unsupervised Learning* (UL), and *Reinforcement Learning* (RL). This chapter solely focuses on fundamental concepts of RL, as the only methods that we will use in the development of our detector are based on reinforcement. We will now proceed with covering the most essential terms and concepts in RL.

The term *"reinforcement"* originates from research conducted on animal behaviour, in the context of adaptive learning and experimental psychology - in the occurrence of an event and the proper relation to a response, the goal is to increase the probability of that specific response occurring again in the same situation [Kim61]. The

same principles are transposed in *Reinforcement Learning* (RL), an area of Machine Learning, described as "the science of decision making" through *optimal control*: an agent must take a suitable action to maximize cumulative rewards for achieving the desired result. In contrast to supervised learning, where the key result is given by labeling training data, the reinforcement agent is bound to adapt and learn from its experience, in the absence of a training dataset.

When it comes to how the behaviour of the agent is constructed through rewards, there are two types of reinforcement: *positive* and *negative*. In *positive reinforcement*, the occurrence of an event, being performed due to a specific behaviour, increases the frequency of that particular behaviour. Reinforcement with a positive effect allows maximised performance, and sustains long-term changes. An overuse of positive reinforcement produces an overload of states, thus diminishing the results.

In contrast, *negative reinforcement* highlights the idea of refining the behaviour of an agent through stopping or avoiding negative conditions. Reinforcement with negative effect builds up the bare minimum structure for an agent to work, providing defiance to a minimum standard of performance.

The environment in which an agent is placed is usually modeled as a *Markov decision process* (MDP), due to the use of *dynamic programming* (DP) techniques [vOW12]. The main difference between DP and RL is that the latter does not assume information about the mathematical model used in the MDP, but they rather target large MDPs where exact models become infeasible [EL22].

Definition 3.1.1 (Markov Decision Process) *An MDP is a discrete-time stochastic control process for decision making in systems that are partly random, and partly under the control of the decision maker.*

We can now translate the definition of an MDP in the context of an RL model. It is generally represented as a *4-tuple* (S, A, P_a, R_a), where:

- S is a set of *environment* and *agent states*;
- A is a set of actions of the agent, called the *action space* (alternatively, A_s is the set of actions available from state s);
- $P_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$;
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.

The state and action spaces may be finite or infinite. Processes with countably infinite state and action spaces can be simplified into ones with finite state and action spaces [Wro84].

We are now able to introduce the essence in RL, when it comes to the strategy utilised by the agent in pursuit of its goals: the *policy*.

Definition 3.1.2 (Policy) *A policy function π is a (potentially probabilistic) mapping from the perceived state space (S) of the environment to its action space (A):*

$$\pi : A \times S \rightarrow [0, 1]$$
$$\pi(a, s) = \Pr(A_t = a \mid S_t = s)$$

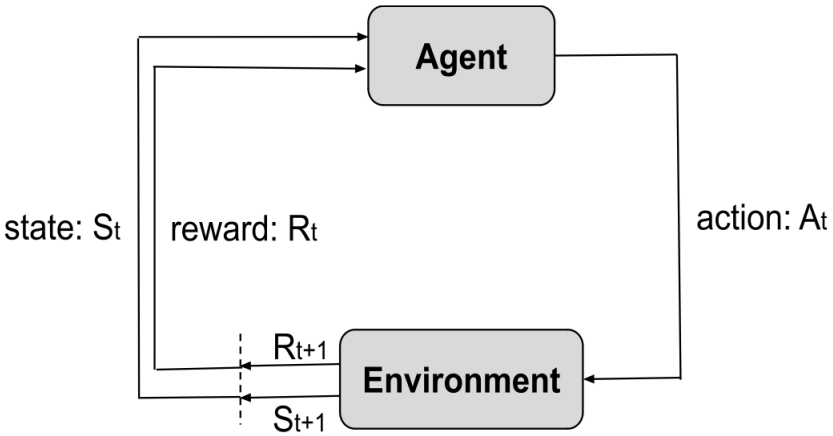


Figure 3.1: The Canonical Agent-Environment Feedback Loop. Basic algorithm for an RL approach: at each time t, the agent receives the current state S_t and reward R_t. It then picks and sends to the environment an action A_t from the set of available actions. The environment moves to a new state S_{t+1} and the associated reward R_{t+1} is computed. The aim of the agent is to learn a policy that maximizes the expected cumulative reward.

Consequently, the network that transforms input frames into output actions is called *"policy network"*. This allows us to finally consolidate the theoretical basis that underlies the methodology of our thesis: one of the simplest ways to train a policy network is given by control algorithms entitled *"policy gradients"*. All MDPs have at least one *optimal* policy.

Definition 3.1.3 (Parameterised Policy Objective) *The objective is to maximise the expected reward, following the imposed parameterised policy:*

$$J(\theta) = \mathbb{E}_{\pi}[r(\tau)] \tag{3.1}$$

A solution method, targeting the maximization problem, that is widely recognized in the ML literature is *Gradient Ascent* (respectively, *Descent)*. In gradient ascent, an update rule is used for iterating through the parameters:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{3.2}$$

The challenge brought by policy gradients is how we determine the gradient of the objective defined at equation 3.1. It is known that integrals are slightly inefficient in computational setting, so a workaround needs to be defined. This consideration lies at the base of the *Policy Gradient Theorem*.

Definition 3.1.4 (Policy Gradient Theorem) *The derivative of the expected reward is given by the expectation of the product between the reward and the gradient of the log of the policy π_θ:*

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}[r(\tau)\nabla_\theta \log \pi_\theta(\tau)] \tag{3.3}$$

One of the biggest challenges in the realm of reinforcement learning has been given by the *Exploration vs. Exploitation Dilemma*.

Exploration refers to the agent's action of trying new strategies that lead to better long-term rewards. It's akin to venturing into the unknown; the agent experiments with different actions and observes their outcomes. Exploration is crucial in the early stages of learning or in dynamically changing environments where previous knowledge may become outdated.

Exploitation, on the other hand, involves leveraging the knowledge the agent has already acquired to make decisions that yield the highest immediate reward. When exploiting, the agent selects the best-known action based on existing information, favoring short-term gains.

The trade-off comes into play, since both actions cannot be performed simultaneously and need to be balanced. If an agent explores too much, it may miss out on known rewards. Conversely, if it exploits too often, it may overlook better options that it hasn't discovered yet.

Balanced strategies have been developed in an attempt to solve the dilemma, one of the most known being the *ϵ-greedy method*, where the agent explores randomly with probability ϵ and exploits with probability $1 - \epsilon$. As learning progresses, the value of ϵ can be reduced, shifting the balance from exploration to exploitation as the agent gains more knowledge.

With this in mind, we are now able to contour a couple of general RL tenets that are pivotal to Proximal Policy Optimization (PPO), which will be explained thoroughly in the next subchapter.

Temporal Difference (TD) Learning is an essential concept in reinforcement learning that combines ideas from *Monte Carlo* methods and *Dynamic Programming* [vOW12]. It allows an agent to learn directly from raw experience without a model of the environment's dynamics. As an agent interacts with the environment, TD Learning updates the value of states in a way that the expected future rewards are estimated more accurately over time.

Moreover, *Q-learning*, a widely known TD Learning algorithm, is particularly notable for its ability to compare the expected utility of the available actions without requiring a model of the environment [WD92]. This value-based method has been instrumental in developing Proximal Policy Optimization (PPO), which further refines the policy optimization process. At the heart of TD learning and Q-learning stand *value functions*.

Considering the Canonical Agent-Environment Feedback Loop scheme from figure 3.1, we can define *value functions* as a measure of the expected long-term reward attainable in certain conditions defined by the states and action space. There are two types of value functions in RL: *state-value* and *action-value*. By the end of this subchapter, we will explore the relation between the two and how they lead to informed and optimal decision-making, required by PPO.

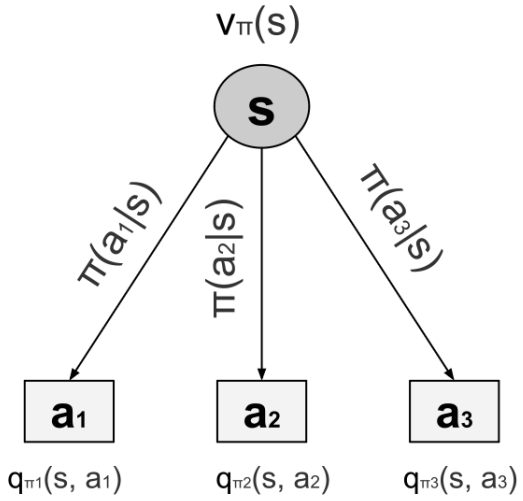


Figure 3.2: Value functions in the context of the action space $A_s = \{a_1, a_2, a_3\}$, generated by state s. Each action a_i is taken with probability $\pi(a_i|s)$.

Definition 3.1.5 (State-Value Function) *The state-value function denotes the expected cumulative reward, starting from state s, following policy π:*

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t r_t \mid s_t = s \right] \tag{3.4}$$

where γ is the discount factor that determines how long the return depends on future rewards.

Alternatively, the total cumulative reward at timestep t can be written using the goal G as shown below [Gor17]:

$$V^\pi(s) = E_\pi\{G_t \mid s_t = s\} \tag{3.5}$$

Definition 3.1.6 (Action-Value Function) *The action-value function denotes the expected cumulative reward, starting from state s, following policy π, taking action a:*

$$Q_\pi(s,a) = E_\pi \left[\sum_{t=0}^{T-1} \gamma^t r_t \mid s_t = s, a_t = a \right] \tag{3.6}$$

In terms of goal G, the action-value function becomes [Gor17]:

$$Q^\pi(s,a) = E_\pi\{G_t \mid s_t = s, a_t = a\} \tag{3.7}$$

The relationships between $V_\pi(s)$ and $Q_\pi(s,a)$ (in terms of each other), in a stochastic policy π, are given by the following Bellman equations [Gor17]:

$$V^\pi(s) = \sum_{a \in A} \pi(a \mid s) * Q^\pi(s,a)$$
$$Q^\pi(s,a) = \sum_{s' \in S} P(s' \mid s,a) \left[R(s,a,s') + \gamma V^\pi(s') \right]$$

Definition 3.1.7 (Advantage Function) *The advantage function, $A^\pi(s,a)$, quantifies how much better it is to take a specific action a in state s over randomly selecting an action according to the policy's probability distribution. Mathematically, it's defined as $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$.*

The important role the advantage function plays in PPO is that it provides a relative measure of the value of actions, guiding the policy update process.

Another staple in modern RL frameworks is given by *Actor-Critic Methods*. These are, in fact, *policy gradients* applied in *TD learning*. Konda & Tsitsiklis describe the *"actor"* component of the model as responsible for selecting actions based on a policy that is directly parameterized, while the *"critic"* assesses the actions taken by the actor by computing a value function [KT00].

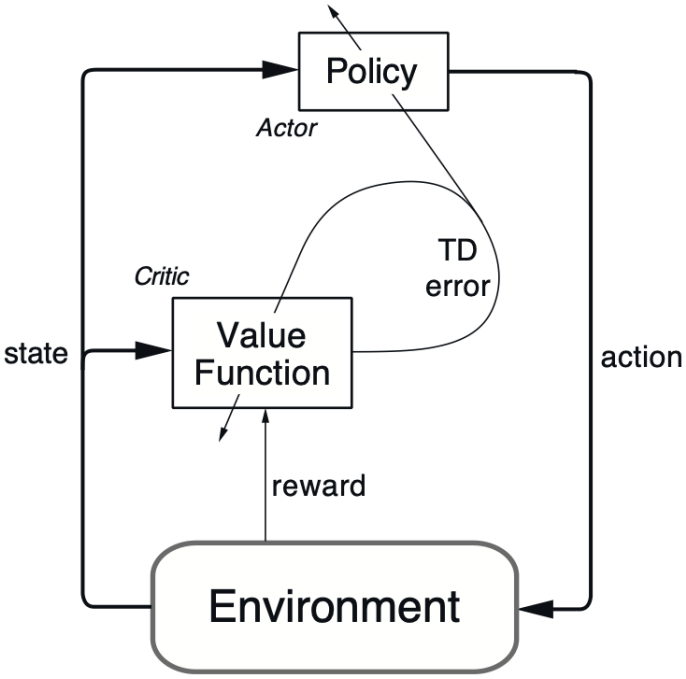


Figure 3.3: The actor-critic architecture. Source: *Reinforcement Learning: An Introduction (Sutton & Barto, 2018)* [SB18].

Having established the foundational concepts of Reinforcement Learning, such as value functions and Temporal Difference Learning, we are now well-positioned to transition to a more focused examination of *Proximal Policy Optimization* (PPO) in the upcoming subchapter.

3.2 Proximal Policy Optimization

Proven successful on a wide variety of tasks from robotic control to surpassing grandmasters at multiple strategy games, *Proximal Policy Optimization* (PPO) is a *deep reinforcement learning* algorithm designed by OpenAI in 2017. Since then, it became the default RL algorithm used by the AI research company, due to its simplicity and outstanding performance [Ope17].

The road to success in RL is often marked by major challenges. One potential issue arising is that training data is itself dependent on the current policy, since agents generate it by interacting with the environment, rather than relying on a static dataset, as it is the case for supervised learning. This implies that data distributions of observations and rewards are constantly updating as the agent learns, leading to major instability in the whole training process. Another problem frequently encountered is that RL approaches suffer from a very high sensitivity to *hyperparameter tuning*.

To address these issues, the OpenAI team developed the Proximal Policy Opti-

mization algorithm. The core purpose behind PPO was to strike a balance between ease of implementation, sample efficiency and ease of tuning.

PPO is a *policy gradient method*. Therefore, unlike Deep-Q Network, it does *not* rely on *experience replay* (where transition experiences are stored in a *replay buffer* [RMT17]); instead the agent learns *online*. We now introduce the *Policy Gradient Loss*, (strongly related to theorem 3.3), as defined by Schulman et al. in their seminal work [SWD+17].

Definition 3.2.1 (Policy Gradient Loss) *The Policy Gradient Loss is an expectation that maximizes the log-probability of beneficial actions weighted by their advantage estimates, thereby reinforcing effective behaviors:*

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right], \tag{3.8}$$

where $\hat{A}_t$ is the noisy estimate for the advantage function at timestep t, and π_θ a stochastic policy.

One potential issue arises when gradient descent is repeatedly applied to the same batch of collected experience: hyperparameters may become suboptimally tuned, extending well beyond the range appropriate for data collection, resulting in inaccurate estimates of $\hat{A}_t$. This issue can be circumvented by ensuring policy updates do not significantly deviate from the previous policy.

This idea was widely introduced by Schulman et al. two years earlier [SLA+15], in the development of *Trust Region Policy Optimization* (TRPO). TRPO serves as the foundational algorithm upon which PPO is constructed.

As a result, in TRPO, a *KL constraint* is added to the *"surrogate" objective*, which will block any major deviation in policy updates [SWD+17]:

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t \right] \tag{3.9}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t \left[\mathrm{KL}[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)] \right] \leq \delta. \tag{3.10}$$

We will refer to the surrogate objective as $L^{\text{CPI}}(\theta)$, *CPI* standing for the *conservative policy iteration*. Additionally, we denote the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. For instance, an action more likely to occur in the current policy than in the old one will have $r_t(\theta) > 1$. L^{CPI} is defined as:

$$L^{\text{CPI}}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right]. \tag{3.11}$$

The KL constraint guarantees monotonic improvement and an efficient optimization of control policies in TRPO [SLA+15]. However, the constraint may cause

additional overhead in the optimization process, potentially leading to undesirable training behavior.

Fixing this issue is exactly the essential target PPO managed to achieve. The main goal of the first-order algorithm is to maintain the monotonic improvement of TRPO, while replacing the hard constraint over the surrogate objective (formula 3.10) with a penalty [SWD+17]. This leads to the main objective of PPO, the *Clipped Surrogate Objective*:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right], \tag{3.12}$$

where ϵ is a hyperparameter, the first parameter of the *minimum* function is $L^{\text{CPI}}(\theta)$, while the second term aims to adjust the surrogate objective by clipping the probability ratio.

This minimalist, yet efficient approach ensures that r_t remains bounded in the interval $[1-\epsilon, 1+\epsilon]$. The value of the advantage estimate may be positive or negative, thus $\hat{A}_t$ affecting the effect of the main operator:

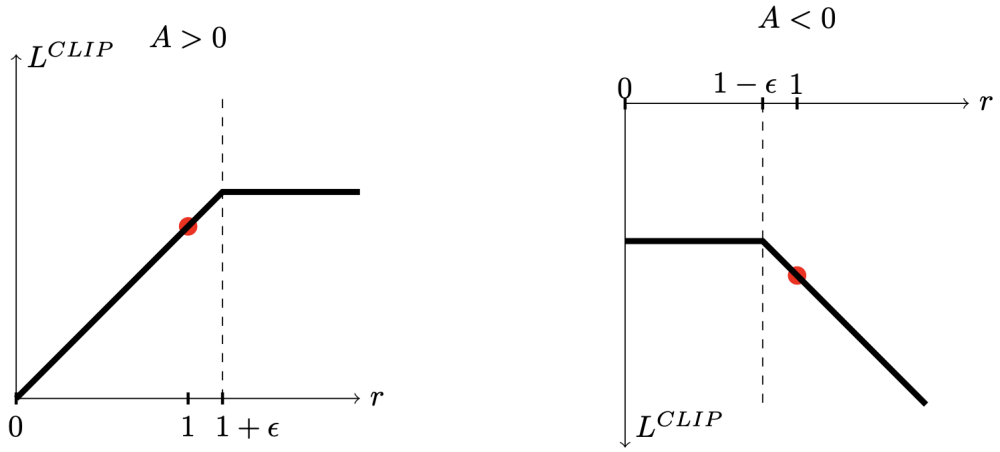


Figure 3.4: One timestep of L^{CLIP} with an advantage estimate of positive (left) and negative (right) value. The first plot showcases a "good" action, more probable after the gradient step. The latter displays the behavior of the agent in the case of undoing the last update policy due to a "bad" action. Source: *Proximal Policy Optimization Algorithms, Schulman et al., 2017* [SWD+17].

We lastly introduce the final training objective in PPO:

$$L_t^{\text{CLIP+VF+S}}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{3.13}$$

where c_1 and c_2 are coefficients, S denotes an entropy bonus, and L_t^{VF} is a squared-error loss.

The final objective formula lies at the foundation of the algorithm implementation proposed by Schuman et al. The PPO algorithm makes use of fixed-length

trajectory segments, and optimizes the surrogate loss using *minibatch SGD* (or for a better performance, *Adam*) [SWD+17]:

Algorithm 1 PPO, Actor-Critic Style

for iteration = 1, 2, ... **do**
  for actor = 1, 2, ..., N **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L with respect to θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Source: *Proximal Policy Optimization Algorithms, Schulman et al., 2017* [SWD+17]

3.3 Employing Proximal Policy Optimization Techniques in Uncovering Market Manipulation

In 2022, Chip Huyen presented in one of her most acclaimed pieces of work, *Designing Machine Learning Systems*, a 2020 survey analysing the large landscape of use cases of enterprise ML (in both internal and external scopes) [Huy22]. We notice that *27%* of enterprise ML applications focus on *"Detecting fraud"*. This demonstrates the feasibility of utilizing ML techniques in our work in detecting forms of financial market manipulation, such as *spoofing & layering*.

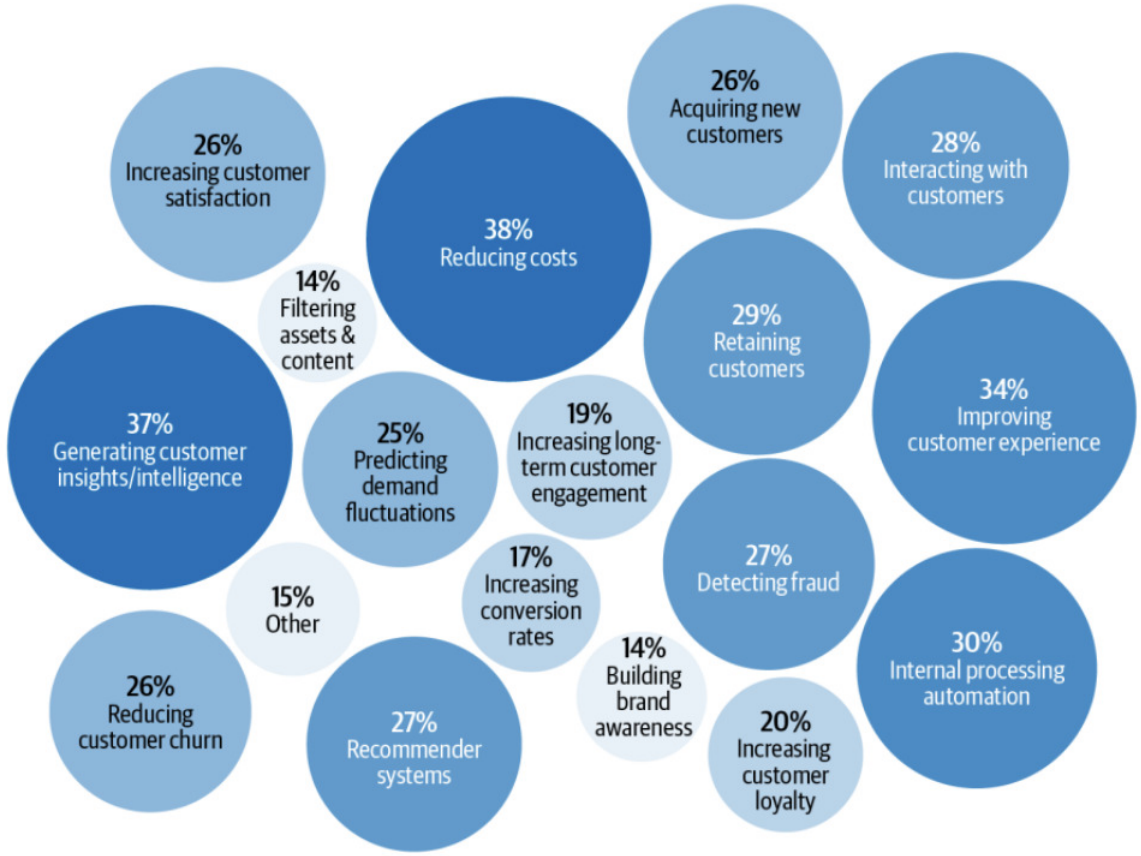


Figure 3.5: 2020 state of enterprise ML. Source: *Designing Machine Learning Systems, C. Huyen* [Huy22].

RL is applied in multiple domains, such as game theory, simulation-based optimization, swarm intelligence, statistics, and multi-agent systems. One obvious characteristic that gives reinforcement learning a major advantage over supervised learning is that, by definition, supervised learning cannot produce an output that is better than human output, since data labelled by humans is involved in the process. This makes RL one of the best candidates in the realm of sports, games, autonomous vehicles, and, most relevant to our thesis, finance. Spoofing and layering are incredibly sophisticated market manipulation tactics, and investors engaging in such

practices constantly improve their tactics to avoid detection, making the instances even harder to be captured by the human eye.

PPO, with its strategic advantage in learning complex policy representations and against the variance in market data, stands out as a promising candidate for improving market surveillance. The algorithm's adaptability and efficiency in policy updates empower it to discern subtle patterns of fraudulent behavior, offering a substantial leap forward in the ongoing effort to uphold market integrity. In the upcoming chapters, we will explore how PPO principles can be applied to maximize efficiency in detecting financial market abuse through spoofing and layering, during the development of *spoof.io*, the application designed as part of this thesis.

Chapter 4

Literature Survey on Spoofing Detection

This chapter aims to take a deep dive into related work on detecting spoofing and layering attempts, or other forms of market manipulation. Before presenting our approach and results, we will analyse and compare solutions based on *Gated Recurrent Units* (GRU), *Feedforward Neural Networks*, *latent multivariate autoregressive* processes, *Adaptive Hidden Markov Models* with anomaly states, and even approaches that model order limit order books using *Statistical Physics* tools.

4.1 A Supervised Learning Approach

In the realm of preventing and detecting market manipulation, supervised learning has been a favoured approach, largely due to its capability to discern patterns from labeled instances of illicit behaviour. Therefore, trading activities can be simply classified as legitimate or manipulative. We will now present in detail two significant contributions in the surveillance of financial markets, that address slightly different problems in market integrity, and utilise labeled data in two different manners. What is more important, is that these approaches will allow us to uncover the trade-offs associated with selecting LOB data of different granularities: Level 1 and Level 2, respectively.

In 2016, Leangarun et al. presented in their seminal work, "Stock price manipulation detection using a computational neural network model", *supervised deep learning* detection methods that focus both on spoofing and *pump-and-dump* attempts [LTT16]. In pump-and-dump, investors create a buying frenzy by misinformation to "pump" the price of the stock, and then eventually "dump" their shares, by selling them at the artificially inflated price.

Their initial approach involved labeled Level 1 LOB data (which is clearly more

accessible to investors) procured from NASDAQ companies, such as Intel, Microsoft, and Amazon. While this decision was proved to be quite successful regarding pump-and-dumping (model achieved 88.25% accuracy [LTT16]), the model failed to detect spoofing effectively. This lead to constructing a model that involved 1-minute LOB snapshots from Level 2 data. One major difference between L1 and L2 data is that Level 2 LOBs contain information about *order cancellations*. We previously discussed in chapter 2 how the volume and time of cancelling previously placed limit orders is a major indicator of spoofing instances. Thus, the results in the case of indicating spoofing positions from L2 data had a much higher accuracy than those from the L1-data model.

The model was based on a *feedforward neural network* (FNN) consisting of 25 nodes in the input layer, 3 nodes on the hidden layer, and one node in the output layer.

Regarding the input, the L2 LOB snapshots were processed to be separated into the OHLCV (open, high, low, close and volume) data - L1 information - of five time steps, and the manipulated class information - additional L2 data. The input layer nodes of the feedforward neural network stored the OHLCV information, while the manipulated class played a key role in analysing the circumstances of cancellation or deletion orders. This information was eventually fed into the price manipulation model. The forecast output was a binary variable that denoted whether the interval represented a spoofing instance or not.

There were three conditions imposed by the FNN in the process of searching for spoofing instances:

1. The price of the cancellation order is close enough to the bid or to the ask price.

2. The value of the cancellation volume is high enough.

3. The value of the last buy order volume is high enough.

For all three conditions, *threshold values* have been defined for calculating the absolute difference between the two terms of the inequalities.

When it comes to evaluating and experimenting with the model developed on L1 data, the testing phase implied *leave-one-out cross validation* (LOOCV). The reason for selecting this evaluation method is that spoofing attempts do not occur as often in real-life situations as in a simulated environment. For reasonable classification results, training data should generally have a 1:1 ratio between the number of points classified as spoofing attempts, and non-spoofing points. Therefore, as opposed to test data, training sets are usually built on data obtained from exceptional events, such as flash crashes, which do not occur on a daily basis.

The detection model fed with L1 data obtained the following results:

Table 4.1: Results of the FNN model with OHLCV data as input. Source: [LTT16]

Average mean square error	1.3992
False-positive error	1.3479
False-negative error	1.4505

It was observed that price volatility is not a key factor in detecting spoofing attempts (due to the inability of comparing *Average True Range* (ATR) values) [LTT16].

In *"Protecting Retail Investors from Order Book Spoofing using a GRU-based Detection Model"* (2021), Tuccella, Nadler and Serban developed an *early detection* model for spoofing, based on *Gated Recurrent Units* (GRU), using Level 2 data from several cryptocurrency exchanges [TNS21]. The outstanding contribution of their work is that detection is performed 2 seconds prior to the end of the manipulation attempt, thus rather acting as a prevention tool. This is a reasonable time span for investors to take action prematurely and avoid trades on a manipulated market.

The Level 2 LOB data was extracted from exchanges like Bitfinex and Kraken, on multiple pairs of cryptocurrencies, over the time span of several months (from November 2019 to May 2020), totaling to over 335 GB of data. For computational efficiency, the gathered raw data was compressed, and the LOB depth was restricted to 25 levels.

Regarding the criteria for classifying points as part of spoofing attempts, Tuccella et al. make use of the same three conditions that were imposed by Leangarun et al. in their work (previously described in this chapter 4.1). Process of price manipulation is finished once all three criteria are met. Let T_0 be that time of flagging a point as a spoofing instance. The time series $T = (T_0 - 12, T_0 - 11, ..., T_0 - 2)$ is used in training the classifier early detection model.

The advantage of opting for a GRU-based approach is that GRU is similar in terms of performance to a *Long Short-Term Memory* (LSTM) strategy, but faster in training. The time dependencies handled by the GRU are then fed into a *feedforward neural network*, using an *Adam* optimiser and hyperparameters tuned with a random search approach. The output is a scalar denoting the probability of the time series representing a price manipulation instance.

Since the focus of the tool is on *early detection*, in the evaluation process, both the accuracy of the model and a fast "response time" during the price manipulation attempt are used as performance indicators. The average accuracy of the model is *75%*, proving that reasonable results can be achieved even in real-time detection.

4.2 Leveraging Unsupervised Anomaly Detection Methods

In the context of supervised learning, labeling market manipulation instances is not very feasible and simple, as it requires manual validation. Moreover, price manipulation practices and strategies are constantly evolving and getting more sophisticated. This leads to unsupervised methods based on anomaly detection becoming a more reasonable option in the context of real-time fraud detection. This chapter explores the solution described in "Adaptive Hidden Markov Model With Anomaly States for Price Manipulation Detection" (2015), and developed by Cao et al. [CLC+15].

Their approach centers around an *Adaptive Hidden Markov Model with Anomaly States* (AHMMAS), which outperformed the previously developed solutions in terms of recognition of various intraday price manipulation methods. As input, Level 2 stock data was procured from NASDAQ and the London Stock Exchange. The model was tested with both simulated, and real market data. The most valuable aspect of analysing the contents of this paper is that this will allow us to have an ensemble perspective over other standard algorithms used in the literature, such as: *Gaussian Mixture Models* (GMM), *k-Nearest Neighbors* (kNN), and *One-Class Support Vector Machines* (OCSVM).

In the context of applying anomaly detection in financial market problems, the goal is to detect *unusual patterns* in the bid-ask price time series. This is where the temporal information contained by L2 data becomes helpful. It was observed that price manipulation is mostly associated with short, high-frequency oscillations around standard equity price levels. Keeping track of such anomalous movements along with the original bid-ask values is the key to building an effective detector [CLC+15].

The feature extraction model works under three patterns of *waveforms*, described in figure 4.1: *Sawtooth*, *Square*, and *Pulse*. As part of the signal decomposition process in the extraction of features, *wavelet transform* was applied. The feature extractor contains a *wavelet filter* and a *gradient calculator*. The four features extracted from the set of bid-ask prices are:

- The original price: P_t;
- The short-term oscillation: $\hat{P}_t$;
- The first-order derivatives of the original price: dP_t/dt;
- The first-order derivatives of the short-term oscillation: $d\hat{P}_t/dt$.

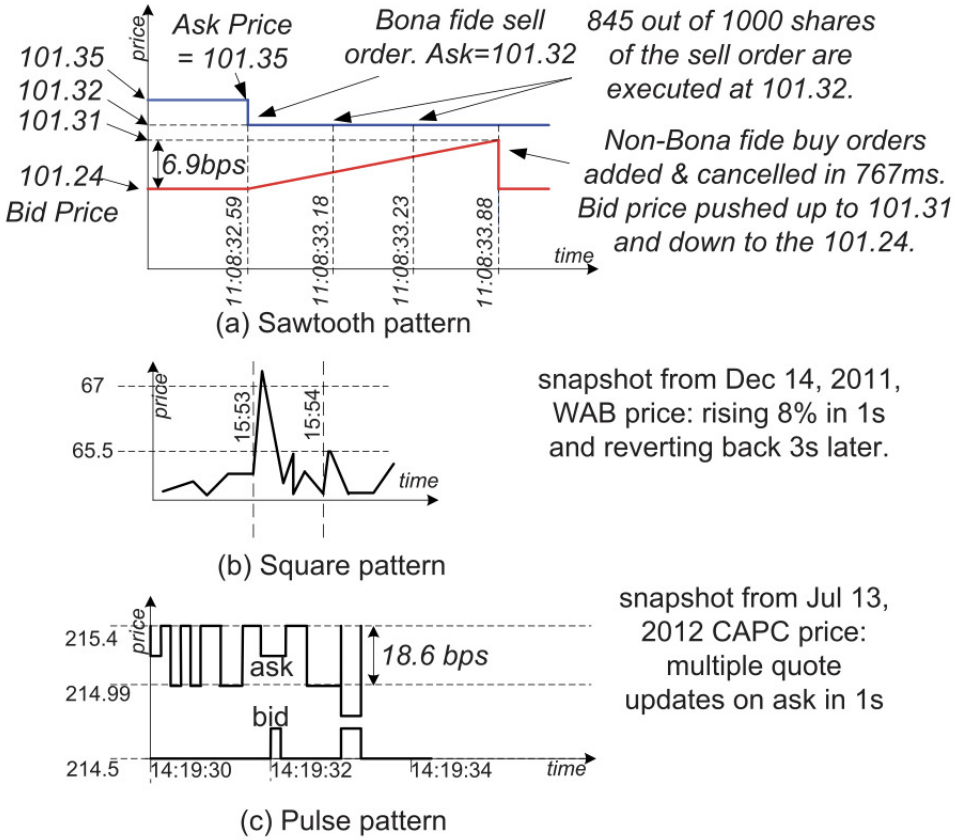


Figure 4.1: Feature extraction module: (a) *Sawtooth* pattern. (b) *Square* pattern. (c) *Pulse* pattern. Source: [CLC+15].

Three issues are highlighted about applying a traditional HMM in detecting manipulation in financial markets. The first one revolves around the complexity of using HMM with multiple features, since it is usually built around one-dimensional sequential data. Additionally, HMMs are usually unable to detect the anomaly type as well, thus making it impossible to differ from the three manipulation patterns. Lastly, the *probability density functions* (pdfs) of equity prices are not constant due to the dynamic nature of the time series.

Consequently, the authors introduced an improved HMM with anomaly states. The pdf of each of the four features is learned independently by the *Gaussian Mixture Model* (GMM). Building the hidden anomaly states starts by setting *anomaly thresholds* for each pdf, similarly to solutions based on *one-class support vector machines* (OCSVM). The data for which the pdf is below 99%, is considered abnormal, this condition allowing to capture the most 0.5%, and the least 0.5% frequent values for each feature.

The thresholds allow partitioning the data into multiple parts that serve as the foundation of the hidden states used in the HMM. The hidden states of all four features are fused into the 32 hidden states of the new adaptive HMMAS. Finally, the 32 states are further simplified as three final manipulation states. Depending on

the features that prove to be anomalous, each of the hidden states falls into one of these three categories.

The adaptive nature of the HMMAS is given by the training being performed in a previous time range, constructed as a *sliding window* with length equivalent to data from one day (detection is done on intraday market data). Therefore, the model becomes an AHMMAS. The window is constantly slid forward to adapt to the non-stationary time series and to mantain the closest points as reference for training. The deviation between the current data sequence and the previous training dataset is identified by *t-test* [CLC+15]. Computational efficiency and performance are balanced using a t-test module with significance level of 1%.

For evaluating the results of the AHMMAS, synthetically generated data was injected into real market data, due to the lack of frequent market manipulation instances in real data. Normal statistical features (such as mean, variance, and volatility) were maintained. The performance of the model is measured using the *receiver operating characteristic* (ROC). The following table contains values for the *Area Under the ROC Curve* (AUC) for the AHMMAS run on 7 different stocks, compared with OCSVM, kNN, and GMM. The AHMMAS clearly outperformed all benchmark models.

Table 4.2: AUC of Four Detection Models on Seven Real Stock Data Sets. Source: [CLC+15]

AUC	AHMMAS	OCSVM	kNN	GMM
AAPL	0.8142	0.6603	0.7926	0.6695
ARM	0.8270	0.5830	0.7982	0.7918
BARC	0.8710	0.6125	0.7627	0.6466
GOOG	0.8025	0.6593	0.5612	0.6163
INTC	0.8971	0.6970	0.6280	0.5200
MSFT	0.7336	0.6419	0.6250	0.6802
VOD	0.8775	0.7044	0.7278	0.7495

4.3 Modeling the Order Book Dynamics Using Statistical Physics

We will now explore the distinctive, yet powerful applications of statistical physics in uncovering price manipulation, through modeling the dynamics of order books. The methodology adapts principles from *econophysics* (including *Brownian motion*, *fluctuation-dissipation* relations, and *statistical aggregation*) to analyse the microstructural behavior of financial markets, providing a novel perspective on surveillance and regulatory measures. The current subchapter reviews several studies that have

contributed to this field, presenting their approaches and findings.

In "The key role of liquidity fluctuations in determining large price changes" (2005), Fabrizio Lillo and J. Doyne Farmer examine the significant role of *liquidity fluctuations* in explaining large price changes in financial markets, focusing on the London Stock Exchange (LSE) [LDF05].

Liquidity gaps, quantified as blocks of adjacent price levels lacking quotes, significantly contribute to price volatility. These gaps, particularly the first three observed in the dataset of 16 high-volume stocks, display a probability distribution characterized by *"fat tails"*, suggesting that extreme values are more common than a Gaussian distribution would predict. It was observed that these gaps follow *power-law distributions* with indices that measure the tail's heaviness. Such distributions reflect the substantial variability in liquidity, which can result in large price swings.

Lillo & Farmer further prove that liquidity gaps are persistent over time through *long-memory* processes. This persistence is quantified using *autocorrelation functions* and *Detrended Fluctuation Analysis* (DFA), revealing that gaps exhibit a memory effect where past states influence future states beyond a short-term scope. The study confirms that the size of the gaps is not only a momentary snapshot of market conditions, but part of a consistent pattern that affects future liquidity and price changes.

Moreover, the gaps are interconnected across different levels of the order book and between the buy and sell sides. The synchronous behavior of gaps indicates that liquidity issues on one side of the market can simultaneously affect the other.

In "Financial Brownian Particle in the Layered Order-Book Fluid and Fluctuation-Dissipation Relations" (2014), Yura et al. introduce an innovative model to describe financial market dynamics, drawing a parallel with the physical theory of *Brownian motion*. This approach conceptualizes the financial markets dynamics as a *colloidal particle* (representing the price) moving through a fluid composed of smaller particles (buy and sell orders). Comprehensive market data was utilised in validating the theoretical model, focusing on correlations within the order book of the foreign exchange market [YTST14].

Empirically, Yura et al. found that the inner layer of the order book, which is closer to the current market price, exhibits a stronger and short-term memory correlation with the price changes. In contrast, the outer layer shows a weaker but longer-lasting memory effect, as noted in [LDF05] as well. This differentiation between the inner and outer layers once again proves the influence of liquidity dynamics on price changes across different timescales.

The study also explores the *fluctuation-dissipation relation* (FDR) in the context of this financial Brownian motion. Under certain conditions, the classic FDR, which links the fluctuations in a system to its response to perturbations, can be applied to financial markets.

One of the key results involves the statistical analysis of the *"drag"* within the market, caused by the interaction between different order layers. This drag impacts the market's responsiveness to new information, thereby affecting price volatility. The analysis confirms that the market dynamics exhibit characteristics similar to those of particles in a fluid, where the "temperature" of the fluid could be analogous to market volatility.

Lastly, we examine the contribution of Li, Polukarov and Ventre in their seminal work, "Detecting Financial Market Manipulation with Statistical Physics Tools" (2023). Their solution for identifying spoofing & layering attempts during the LUNA flash crash from May 2022, focuses on modeling the dynamics within the LOB by treating market orders as particles. This analogy allows the computation of a momentum measure, $m = s \cdot v$, where s is the size of the order and v is its velocity, which is defined as the rate of price change per unit of time. The key contribution of the paper is that they leverage modeled *Level 3* LOB data in performing the detection. Their approach proves to be effective, outperforming the standard anomaly detection *Z-score* model [LPV23].

The core of their model is the aggregation of this momentum over a defined period to capture the systemic behavior of the market. They define the velocity of the midprice $v_M(t) = \frac{z_M(t) - z_M(t-\Delta t)}{\Delta t}$, where $z_M(t)$ is the market midprice at time t, and Δt is the sampling interval.

This model is then employed to analyse the order flow and detect anomalies in real trading data from the LUNA/USD market, especially during its flash crash. It focuses on the "active area" of the LOB, where most trading activities occur, and extends to a "passive area" to detect less obvious manipulative activities.

In terms of evaluation, the method is compared against a conventional Z-score-based anomaly detection method, across data of the LUNA and Bitcoin cryptocurrencies. The model successfully identifies manipulative trading behaviors by tracking the cumulative sum of net momentum, calculated as:

$$M = \sum_{t \in (T-\Delta t, T]} \sum_{\gamma = b_M - \alpha}^{a_M + \alpha} N_\gamma(t) \sum m \tag{4.1}$$

where $N_\gamma(t)$ is the number of limit orders at depth γ at time t, and α defines the depth of the active area around the bid-ask spread [LPV23].

During the LUNA flash crash, the model detected significant spoofing activities by observing the deviations in the momentum measures, which were not captured by the Z-score model. Limit orders played a crucial role in driving the prices down, where the attempt of market orders to drive prices up was unsuccessful, indicating potential manipulative activities. The comparison in accuracy between their model

and the Z-score method shows a clearer detection of anomalous events with the former, particularly in identifying the exact times and types of orders involved in spoofing.

4.4 Comparative Analysis of Detection Methods

The final section of this chapter aims to perform a thorough comparative analysis of the price manipulation detection methods discussed in our literature survey. We will explore multiple trade-offs, including computational efficiency, performance, flexibility, and granularity of market data. Both the advantages and disadvantages of each method will be highlighted. This will serve as a solid reasoning foundation during the development of our novel approach, which utilises Proximal Policy Optimization.

Before diving into the analysis, we will establish the criteria for evaluating each discussed method, which are of high interest for the development of our tool:

- **Granularity of input data**: The complexity of the order book information that was used during the development and the testing phases (Level 1, Level 2, respectively Level 3).
- **Accuracy**: The ability of the method to correctly identify instances of market manipulation without generating false positives. Detection of more sophisticated methods of market manipulation should obtain a reasonable accuracy, regardless of the granularity of the input data.
- **Computational Efficiency**: The amount of computational resources required and the speed of the detection process, which are crucial for real-time analysis.
- **Flexibility and Reliability**: The method's ability to perform under different market conditions and across various types of assets.
- **Ease of Implementation**: The complexity involved in implementing and maintaining the method in operational environments.

Supervised learning models, particularly those employing deep learning architectures like GRUs and FNNs, have shown high accuracy in detecting known patterns of market manipulation. The GRU-based model developed by Tuccella et al. [TNS21] achieved an accuracy of 75%, which is commendable for real-time detection. Their work proved the advantage of GRU over LSTM, the former being faster during the training phase. However, these models require extensive labeled datasets for training, which can be a significant limitation in markets where manipulation tactics evolve rapidly.

On the other hand, the FNN model discussed by Leangarun et al. [LTT16] demonstrated the impact of data granularity by showing improved results with Level 2 data over Level 1 data, highlighting the importance of detailed order book information (especially on cancellation orders and their depth) in enhancing accuracy of manipulation detection. Lastly, both papers define solid criteria for labeling spoofing & layering instances (presented during the first subchapter 4.1), that will serve as a good framework for defining the surrogate objective, and the cumulative reward of our agent.

The Adaptive Hidden Markov Model with Anomaly States (AHMMAS) introduced by Cao et al. [CLC+15] represents a significant advancement in unsupervised detection methods. It outperformed traditional models by effectively recognizing anomalous trading patterns without prior labeling of L2 snapshots. Although this model offers powerful detection capabilities and adapts to new data through its sliding window mechanism, its complexity and computational demands could pose challenges for real-time market surveillance.

The works of Lillo & Farmer [LDF05], and Yura et al. [YTST14] provide foundational insights into how liquidity fluctuations and price movements can be analyzed using statistical physics. Their methods are particularly adept at capturing complex interactions within the order book that traditional models might overlook. This demonstrates the complex interplay between market orders, limit orders, and cancellations, a remark that is essential in our study conducted on spoofing & layering.

Li, Polukarov, and Ventre's approach [LPV23] successfully applied these principles to detect spoofing & layering during the LUNA cryptocurrency flash crash. Their model, which considers orders as particles whose interactions can be quantified and analyzed, showcased superior performance in identifying manipulative behaviors that were not detected by more traditional Z-score-based anomaly detection methods. Their work proved that the use of more granular data, of Level 3, can be both successful and computationally efficient at the same time. Thanks to their exceptional results and insightful statistics, our solution will make use of the exact same L3 real market data on the LUNA/USD pair, gathered during the LUNA flash crash from May 2022 (11/05/2022, 16:00-20:00).

When comparing all methods, it's evident that while supervised and unsupervised learning strategies provide accurate tools for detecting known and evolving manipulative patterns, statistical physics approaches bring a novel and profound analytical depth. An ideal solution would combine the simplicity and computational power of supervised learning in real-time detection, the absence of a requirement for labeled data in unsupervised learning strategies, and the accuracy given by state-of-the-art econophysics methods. We consider that a PPO-based approach will effectively harness most of these advantages.

Chapter 5

spoof.io: Web Tool for Spoofing Detection on LUNA

In this chapter, we present the development of the practical application that integrates a spoofing detector using a Proximal Policy Optimization (PPO) model. The web application, entitled ***spoof.io***, is a proof-of-concept visualisation tool that simulates historical Level 3 LOB data from the LUNA flash crash from May 2022, and outputs orders detected as spoofing attempts in real time. In the first part of the chapter, we will dive into the development and evaluation of our PPO Policy Network model. Finally, we will detail the architecture, data flow, and pipeline of *spoof.io*.

5.1 Proximal Policy Optimization Model Development

The development of our Proximal Policy Optimization (PPO) model involves a systematic approach to designing, implementing, and refining a neural network capable of detecting spoofing attempts in financial markets. This process consists of multiple stages, including preprocessing, feature engineering, the simulation of the market environment, the construction of the PPO policy network. Finally, we train and test our model, performing hypertuning with the aim of finding the optimal configuration for our model's efficiency.

5.1.1 Data Collection and Preprocessing

As discussed in the theoretical background of this thesis, spoofing highly occurs in events marked by market instability, such as flash crashes. Therefore, when collecting our data, there are several important aspects that we took into consideration. Analysing Level 3 (the most granular level) LOB data as part of a PPO-based detector is extremely important, as it marks our contribution, and temporal data is

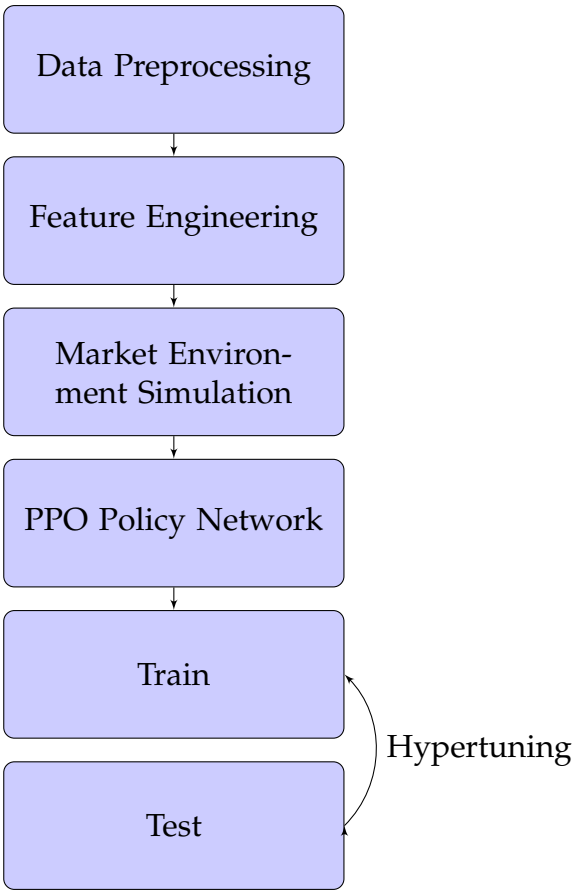


Figure 5.1: Key Steps in Proximal Policy Model Development

effectively leveraged. Additionally, procuring data from recent flash crashes is essential - there are higher chances of more subtle and diverse spoofing methods to occur in more recent data. Finally, since we employ unsupervised methods and do not work with labeled data, to ensure the correctness of our model evaluation, we must make use of historical data for which statistical reports on spoofing attempts were already performed, verified, and are publicly available.

All these considerations lead to our decision of working on the wLUNA/USD data that was utilised by Li et al. in their seminal work [LPV23]. The dataset was provided by the authors of the paper, Haochen Li, Maria Polukarova, and Carmine Ventre. It was initially fetched from Coinbase via a WebSocket feed, and it consists of Level 3 LOB historical data on wLUNA/USD from the LUNA flash crash from May 2022 (11/05/2022, 16:00-20:00).

Orders placed at a price that matches or is less favourable than an existing order (typically the best bid or ask) are executed immediately, resulting in a $match$ record. Orders that are not executed, or only partially executed, become limit orders in the LOB, creating an $open$ record. These limit orders will remain in the LOB until they are cancelled, generating a $canceled$ record. The minimum price precision is 0.01 USD, and the smallest trading volume unit (order size) is 0.001 LUNA. The times-

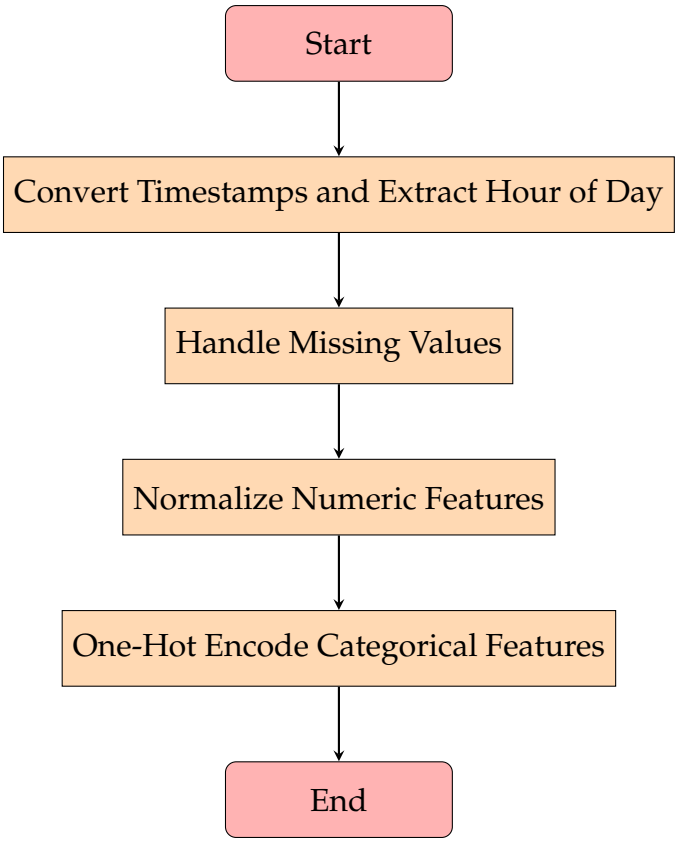


Figure 5.2: Data Preprocessing Pipeline

tamps are recorded with a precision of one microsecond, and are marked when an order activity (or event) occurs, making these event-driven timestamps discrete and non-consecutive [LPV23].

The raw data consists of six JSON files: three for the full channel data, respectively three for the ticker data. The event is split across three temporal windows (16:00 - 17:00, 18:00 - 19:00, respectively 19:00 - 20:00), each of the three files corresponding to one temporal window.

Table 5.1: Summary of raw dataset JSON files and their record counts

#	**File Name**	**Number of Records**
1	FullChannel_GDAX_20220511_17hr.json	768,272
2	FullChannel_GDAX_20220511_19hr.json	456,282
3	FullChannel_GDAX_20220511_20hr.json	550,124
	Total Full Channel Records	**1,774,678**
4	Ticker_GDAX_20220511_17hr.json	37,761
5	Ticker_GDAX_20220511_19hr.json	27,864
6	Ticker_GDAX_20220511_20hr.json	37,180
	Total Ticker Records	**102,805**

The first step in the preprocessing pipeline is to handle the temporal aspects of

the data. We convert the timestamp column into a datetime format and extract the hour of the day from each timestamp, as trading behaviours often exhibit diurnal patterns. By including the hour of the day as a feature, we allow the model to learn and leverage these temporal patterns, which can be indicative of spoofing activities that may occur at specific times. Formally, if t is the timestamp, the extracted hour h can be represented as:

$$h = hour(t) \tag{5.1}$$

Next, we address the issue of missing values, which is a common challenge in financial datasets. For numeric features, we employ median imputation to fill in missing values. The median is chosen for its ability to eliminate outliers, ensuring that the central tendency of the data is preserved. This step prevents the disruption of the model's learning process due to incomplete data. For categorical features, missing values are replaced with a placeholder $missing$, ensuring that no data points are excluded.

Normalization is the next step in the preprocessing pipeline. By applying MinMaxScaler, we scale the numeric features to a range between 0 and 1. In the case of neural networks, this normalization approach ensures that features with larger magnitudes do not disproportionately influence the model. Consistent scaling facilitates faster convergence and improves the overall stability of the learning process. The normalization for a feature x is given by:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{5.2}$$

where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values of the feature x, respectively.

Categorical features are processed through one-hot encoding, transforming them into binary vectors, therefore, each category is treated as a distinct entity without imposing any ordinal relationships. For instance, order types such as $received$, $open$, $done$, and $match$ are each represented as separate binary features.

$remaining\_size\_change$ captures the dynamics of order size changes over time. By calculating the difference in remaining sizes across consecutive records grouped by order ID, we can identify patterns indicative of spoofing, such as frequent and abrupt changes in order sizes. This feature provides the model with additional insights into the aggressive placement and cancellation of orders, which are hallmark behaviours of spoofing activities. Formally, for an order i with size s at time t, the change in remaining size Δs_i can be represented as:

$$\Delta s_i = s_i(t) - s_i(t-1) \tag{5.3}$$

For the full channel data, we maintain identifiers such as $order\_id$ and $time$ to preserve traceability. The ticker data complements the order book data by providing additional context on price movements and trading volumes, further enriching the dataset for model training.

5.1.2 Feature Engineering

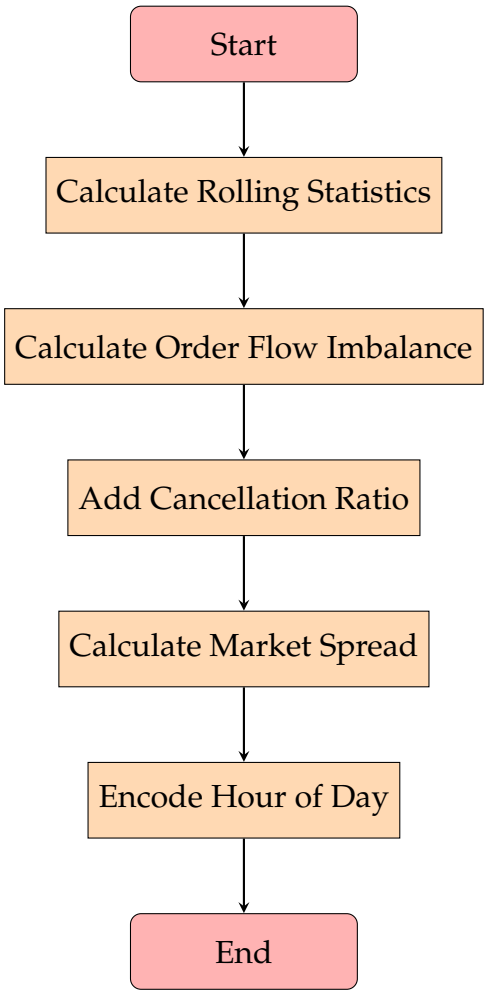


Figure 5.3: Feature Engineering Pipeline

The feature engineering process is designed to extract and engineer features from the preprocessed data, enhancing its suitability for our PPO model. This process involves creating additional attributes that provide more insight into the trading behaviors and market conditions. The features will be integrated either in the anomaly detection phase during the market simulation, or directly as hyperparameters in the architecture of the PPO policy network.

During this process, we mark our contribution by designing and processing unique features that effectively harness temporal data and financial statistics to build a PPO approach for detecting market manipulation. Our objective is to fully

leverage the theoretical foundations presented in Chapter 2 and the spoofing preconditions outlined in Section 4.1, as introduced and effectively applied by Leangarun et al. [LTT16]. Through this, we aim to create a novel application of PPO in financial markets integrity, by translating theoretical insights from the literature into a practical application, thus proving our efficiency and correctness, while still introducing innovative and heuristic elements into our approach.

We begin by calculating rolling statistics such as mean, standard deviation, and variance for windows on multiple sizes (5, 10, and 15, respectively) on important columns like $price$ and $size$. These statistics provide us with a detailed view of the central tendency and dispersion of the data over different time periods. For instance, by examining the mean (μ), standard deviation (σ), and variance (σ^2), we can identify abnormal fluctuations in price and order sizes, which are indicative of spoofing. A sudden increase in the variance of order sizes, for example, may suggest the presence of spoof orders intended to create a false impression of market depth.

Next, we compute the order flow imbalance (OFI), which measures the discrepancy between buy and sell orders over a rolling window. This is achieved by assigning a signed size to each order (positive for buy orders and negative for sell orders) and summing these sizes over the window. The OFI directly reflects market pressure, caused by the imbalance between buy and sell orders. Mathematically, the OFI can be expressed as:

$$OFI(t) = \sum_{i=t-w}^{t} size_i \cdot side_i \tag{5.4}$$

where w is the window size, size_i is the size of order i, and side_i is +1 for buy orders and -1 for sell orders. High positive or negative OFI values indicate significant market pressure, which can signal spoofing activities.

We also add the cancellation ratio to the dataset, which is the ratio of cancelled orders to received orders. This ratio is a significant indicator of spoofing, as spoofers often place a large number of orders and cancel them quickly to create a misleading sense of market activity, as previously seen in Section 4.1. The cancellation ratio (CR) is calculated as:

$$CR = \frac{reason\_canceled}{type\_received\_adjusted} \tag{5.5}$$

where $type\_received\_adjusted$ ensures that the denominator is never zero by replacing zero values with one. A high cancellation ratio suggests a higher likelihood of spoofing.

The market spread is calculated as the difference between the best ask and best bid prices. The market spread (spread) provides insight into the market's liquidity

and the aggressiveness of trading activities. It is expressed as:

$$spread = best\_ask - best\_bid \tag{5.6}$$

A large spread often indicates uncertainty or manipulation in the market, which is commonly associated with spoofing.

Additionally, we one-hot encode the $hour\_of\_day$ feature to capture diurnal patterns in trading activity. This encoding ensures each hour is represented as a separate binary feature, allowing the model to identify time-specific spoofing behaviors. One-hot encoding transforms the hour of day (h) into a binary vector where only the component corresponding to h is 1, and all others are 0.

This leads to the final, enhanced CSV datasets for the full channel, and ticker, respectively, that are now ready to be fully harnessed in the model pipeline.

Table 5.2: Features of the Processed Data

Dataset	**Features**
Full Channel Data	order_id, time, price, size, remaining_size, remaining_size_change, type_change, type_done, type_match, type_open, type_received, side_buy, side_sell, reason_canceled, reason_filled, reason_missing, hour_of_day, price_5_mean, price_5_std, price_5_var, price_10_mean, price_10_std, price_10_var, price_15_mean, price_15_std, price_15_var, size_5_mean, size_5_std, size_5_var, size_10_mean, size_10_std, size_10_var, size_15_mean, size_15_std, size_15_var, signed_size, order_flow_imbalance, cancel_to_received_ratio, hour_15, hour_16, hour_17, hour_18, hour_19
Ticker Data	sequence, price, open_24h, volume_24h, high_24h, volume_30d, best_bid, best_ask, trade_id, last_size, hour_of_day, side_buy, side_sell, spread, last_size_5_mean, last_size_5_std, last_size_5_var, last_size_10_mean, last_size_10_std, last_size_10_var, last_size_15_mean, last_size_15_std, last_size_15_var, hour_15, hour_16, hour_17, hour_18, hour_19

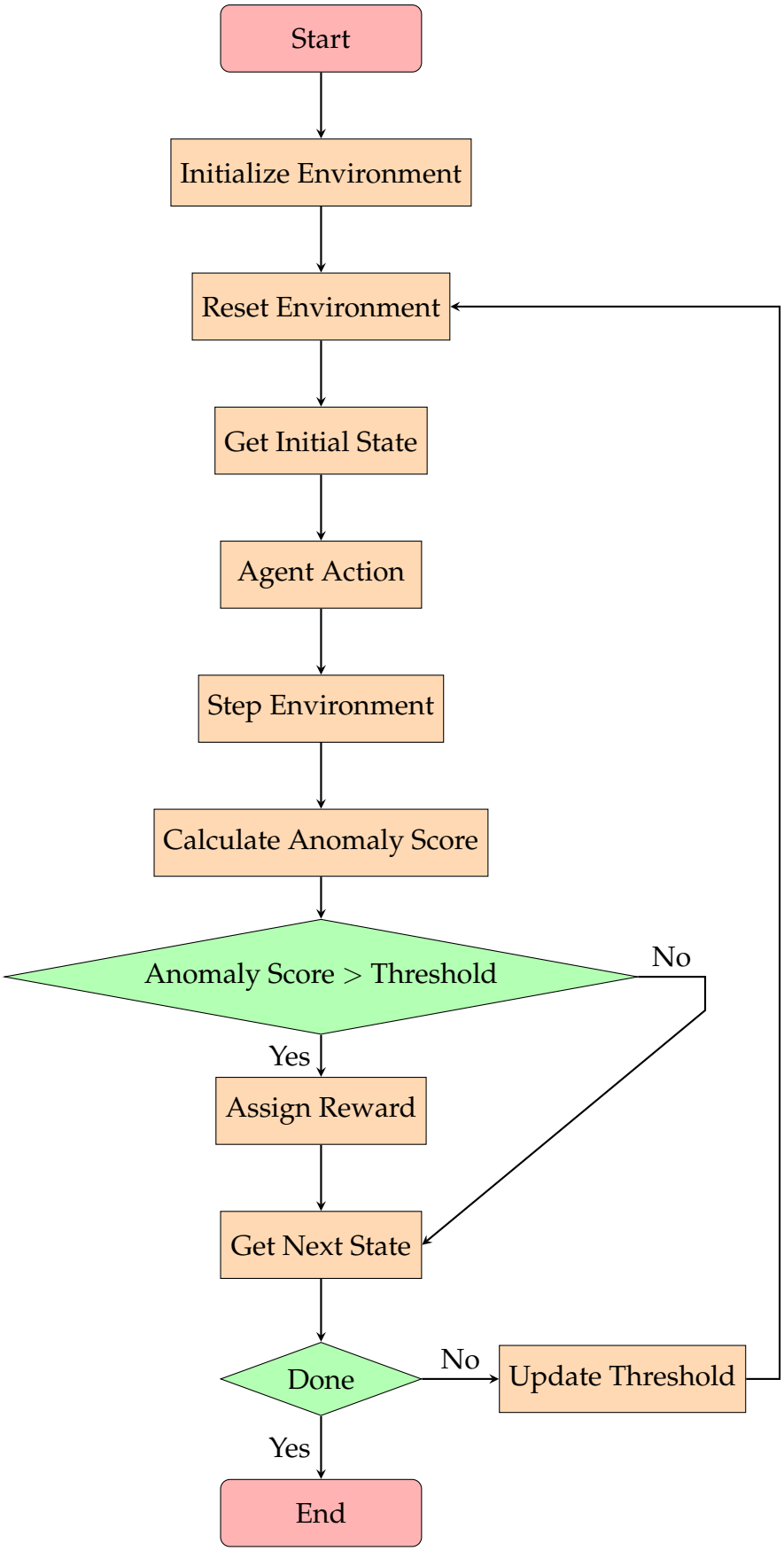


Figure 5.4: Market Simulation Environment Pipeline

5.1.3 Development of the Market Simulation Environment

The market simulation environment provides a controlled setting where the Proximal Policy Optimization (PPO) model can interact with historical market data, learn patterns, and improve its detection capabilities.

The `MarketEnvironment` class initializes by loading and preparing the necessary market data. This data includes both full channel and ticker data, which are segmented based on whether the environment is used for training or testing. This prevents overfitting and ensures generalisability; further details regarding data segmentation are given in Section 5.2.6.

The environment maintains a `current_index` to track the current position in the dataset, ensuring that the agent processes data sequentially. The `reset()` method reinitializes the environment to its starting state, setting the `current_index` to a valid position within the data bounds and updating the spoofing threshold. This reset mechanism ensures that each training episode starts afresh, providing the model with a consistent and repeatable starting point.

The `step()` method processes the agent's actions, advancing the market simulation by one step and returning the new state, reward, and other relevant metrics. The reward structure is designed to reinforce correct detections of spoofing and penalize incorrect actions. This is mathematically represented as:

$$reward = \begin{cases} 1 & \text{if } (action = 1 \text{ and } is\_spoofing) \text{ or } (action = 0 \text{ and } !is\_spoofing) \\ -1 & \text{otherwise} \end{cases} \tag{5.7}$$

where $is\_spoofing$ is determined by comparing the anomaly score to the spoofing threshold.

The `get_state()` method generates the current state by concatenating historical features from both full channel and ticker datasets. This approach ensures that the state includes a comprehensive view of market conditions over a specified history window, defined by `Config.HISTORY_WINDOW_SIZE`. The numerical features are extracted and flattened into a single array, providing the model with a detailed snapshot of recent market activity.

Finally, since our dataset is not labeled, we need a method to assess the correctness of the agent's decisions and correlate it with the reward system. For this, we perform anomaly detection on orders. The anomaly score of each order is calculated based on predefined feature weights and market data features at the given index. The specific weights and other parameters used in this calculation will be

discussed in detail in Section 5.2.5. The score is computed as:

$$anomaly\_score = \sum_i w_i \cdot \log(1 + |feature_i|) \tag{5.8}$$

where w_i are the weights from `Config.FEATURE_WEIGHTS` and $feature_i$ are the corresponding feature values. This scoring mechanism allows the model to quantify the likelihood of spoofing based on the aggregated influence of multiple features.

To adapt to changing market conditions, the spoofing threshold is dynamically updated based on recent anomaly scores. This ensures that the model remains responsive to shifts in market behavior. The threshold is updated to the 75% of the recent anomaly scores, represented as:

$$spoofing\_threshold = percentile(\{anomaly\_score_i\}_{i=n-50}^{n}, 75) \tag{5.9}$$

where n is the current index in the dataset.

Algorithm 2 Market Simulation Environment

Input: Full channel data, ticker data
Output: State, reward, anomaly score, spoofing threshold
procedure INITIALIZE
    Load and split data
    Initialize parameters and update threshold
end procedure
procedure RESET
    Reset state and update threshold
    return initial state
end procedure
procedure STEP(action)
    if index exceeds data length **then**
        return None, None, 0, True, 0, threshold
    end if
    Calculate anomaly score and determine spoofing
    reward $\leftarrow$ 1 if action matches spoofing, else -1
    Increment index and get next state
    return next state, data, reward, done, anomaly score, threshold
end procedure
procedure GET STATE
    Concatenate historical features from both datasets
    return feature array
end procedure
procedure CALCULATE ANOMALY SCORE(index)
    Compute score based on feature weights
    return anomaly score
end procedure
procedure UPDATE THRESHOLD
    Update threshold to 75th percentile of recent scores
end procedure

5.1.4 Implementation and Training of the Proximal Policy Optimization Policy Network

The PPO Policy Network is implemented using PyTorch and features a straightforward feed-forward neural network architecture. The network consists of an input layer, two hidden layers with 256 neurons each, and an output layer. Each hidden layer employs ReLU activation functions to introduce non-linearity, allowing the

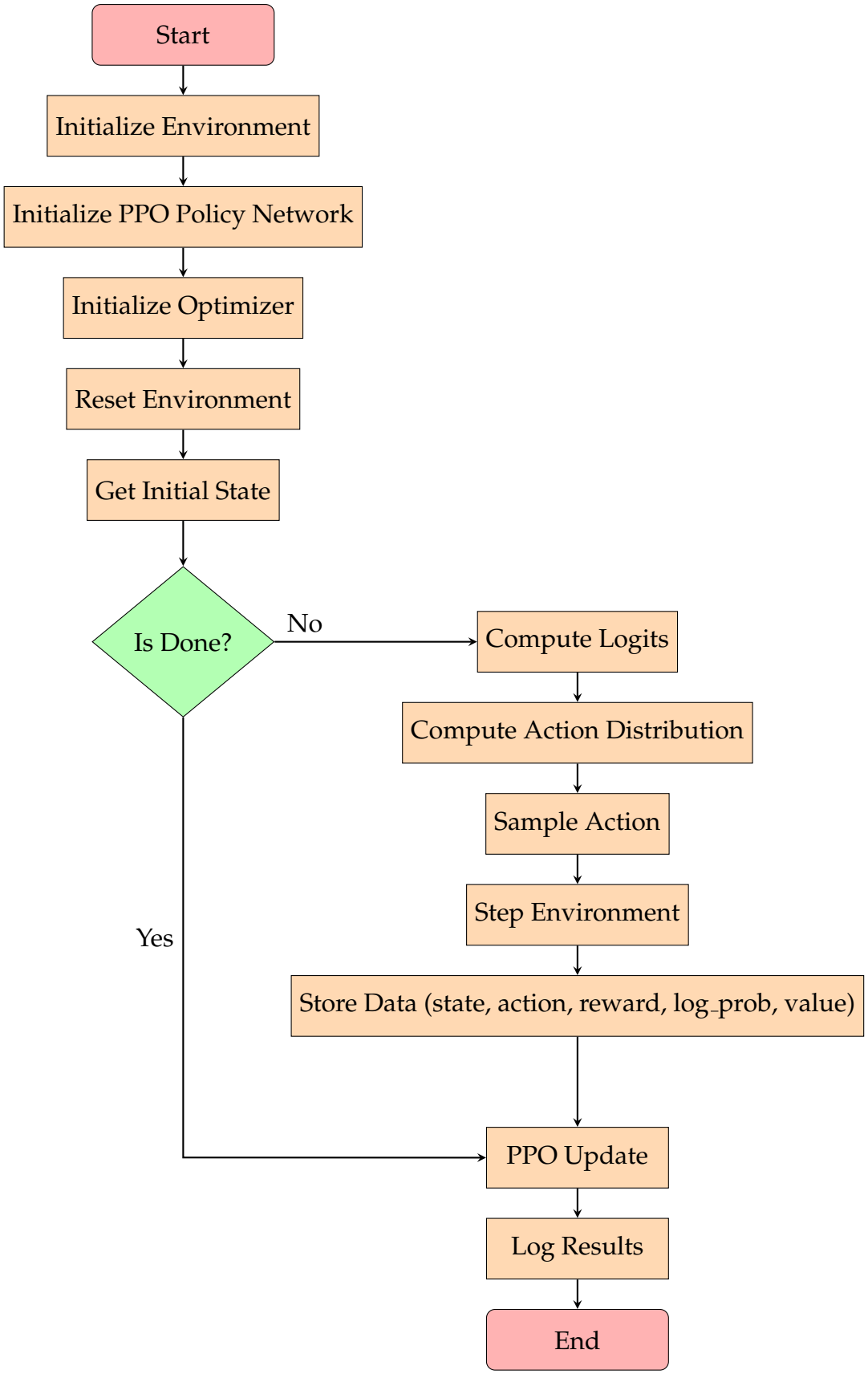


Figure 5.5: PPO Policy Network Training Pipeline

network to capture complex patterns within the trading data.

The neural network is defined as follows:

Algorithm 3 PPOPolicyNetwork

```
procedure PPOPolicyNetwork(num_features, num_actions)
    layers ← Sequential(
        Linear(num_features, 256),
        ReLU(),
        Linear(256, 256),
        ReLU(),
        Linear(256, num_actions)
    )
end procedure
function forward(x)
    return layers(x)
end function
```

This architecture is designed to process the input features derived from market data and produce logits for the action probabilities. The logits indicate the likelihood of each possible action, where the actions are binary: 0 (no spoofing) and 1 (spoofing). This design is particularly effective for our problem as it allows the network to learn and distinguish between normal and suspicious trading behaviors.

To optimize the policy network, we compute discounted rewards. Discounted rewards take into account future rewards to emphasize immediate actions that could indicate spoofing. The formula used is:

$$R_t = r_t + \gamma R_{t+1} \tag{5.10}$$

where r_t is the reward at time step t and γ is the discount factor. This recursive calculation accumulates rewards backwards through time, providing a measure of the long-term benefit of each action. The implementation is as follows:

Algorithm 4 get_discounted_rewards

```
procedure GET_DISCOUNTED_REWARDS(rewards, gamma)
    discounted_rewards ← []
    R ← 0
    for r in reversed(rewards) do
        R ← r + gamma * R
        discounted_rewards.insert(0, R)
    end for
    return discounted_rewards
end procedure
```

This approach ensures that the network not only focuses on immediate rewards, but also considers the future impact of its actions, an absolutely essential factor in identifying patterns of spoofing that may unfold over multiple time steps.

The Generalized Advantage Estimation (GAE) is used to compute more stable advantage estimates. The advantage A_t at time step t is calculated using:

$$A_t = \delta_t + \gamma\lambda A_{t+1} \tag{5.11}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, and $V(s_t)$ represents the value function. This method smooths out the advantage estimates, making policy updates more stable and reliable. The implementation is as follows:

Algorithm 5 compute_advantages

```
procedure COMPUTE_ADVANTAGES(rewards, values, gamma, lam)
    advantages ← []
    last_adv ← 0
    for t in reversed(range(len(rewards))) do
        delta ← rewards[t] + gamma * values[t + 1] - values[t]
        last_adv ← delta + gamma * lam * last_adv
        advantages.insert(0, last_adv)
    end for
    return advantages
end procedure
```

GAE helps mitigate the high variance typically associated with policy gradient methods by incorporating a balance of bias and variance; the detection of spoofing needs to be effective against the noisy and volatile nature of financial markets.

The PPO update mechanism is designed to adjust the network weights in a controlled manner. It calculates the loss function and performs gradient descent to up-

date the policy. The loss function includes a clipped surrogate objective, a value function loss, and an entropy bonus. The clipped surrogate objective is defined as:

$$L^{\text{CLIP}} = \mathbb{E}\left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot A_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right) \cdot A_t\right)\right] \tag{5.12}$$

where $\pi_\theta(a_t|s_t)$ represents the new policy, and $\pi_{\theta_{\text{old}}}(a_t|s_t)$ is the old policy. This objective helps maintain a balance between the new and old policies, preventing drastic updates. The implementation is as follows:

Algorithm 6 ppo_update

```
1: procedure PPO_UPDATE(network, optimizer, states, actions, old_log_probs, advantages, returns, clip_param)
2:     logits ← network(states)
3:     dist ← Categorical(logits=logits)
4:     new_log_probs ← dist.log_prob(actions)
5:     entropy ← dist.entropy().mean()
6:     ratios ← torch.exp(new_log_probs - old_log_probs.detach())
7:     surr1 ← ratios * advantages
8:     surr2 ← torch.clamp(ratios, 1 - clip_param, 1 + clip_param) * advantages
9:     actor_loss ← -torch.min(surr1, surr2).mean()
10:    critic_loss ← 0.5 * (returns - dist.probs.mean()).pow(2).mean()
11:    loss ← actor_loss + critic_loss - Config.PPO_CONFIG['ent_coef'] * entropy
12:    optimizer.zero_grad()
13:    loss.backward()
14:    torch.nn.utils.clip_grad_norm_(network.parameters(), Config.PPO_CONFIG['max_grad_norm'])
15:    optimizer.step()
16:    return loss.item()
17: end procedure
```

The PPO update ensures that the policy improvement is conservative and stable when dealing with the highly dynamic and stochastic environment of financial markets. The inclusion of an entropy term in the loss function encourages exploration, preventing the policy from becoming too deterministic and potentially missing out on novel strategies to detect spoofing.

To integrate the PPO Policy Network with our market simulation environment, we initialize the environment and the network. The environment provides state data, while the network processes these states to predict actions and update its policy. The network is trained to recognize patterns indicative of spoofing.

5.1.5 Hyperparameter Tuning

The anomaly score is computed using a weighted sum of selected features. The weights for these features are defined in the configuration file (`Config.FEATURE_WEIGHTS`). These weights determine the importance of each feature in contributing to the overall anomaly score. The formula for calculating the anomaly score is described in Equation 5.8.

The following table lists the feature weights used in our model:

Table 5.3: Feature Weights for Anomaly Score Calculation

Feature	**Weight**
order_flow_imbalance	0.15
cancel_to_received_ratio	0.15
price_5_std	0.05
price_10_std	0.05
price_15_std	0.05
size_5_var	0.05
size_10_var	0.05
size_15_var	0.05
spread	0.10
last_size_5_var	0.05
last_size_10_var	0.05
hour_of_day	0.15
hour_15	0.025
hour_16	0.025
hour_17	0.025
hour_18	0.025
hour_19	0.025

These weights were selected based on domain knowledge and empirical experimentation. During hyperparameter tuning, we adjust these weights to find the optimal combination that maximizes the model's performance in detecting spoofing.

The hyperparameters are as follows:

- **Learning Rate (α)**: This controls how much to change the model in response to the estimated error each time the model weights are updated. We experimented with learning rates of 1×10^{-4}, 5×10^{-4}, and 1×10^{-3}.
- **Batch Size**: This defines the number of samples that will be propagated through the network. We evaluated batch sizes of 32, 64, and 128.
- **Number of Epochs**: This parameter specifies the number of complete passes through the training dataset. We tested with 10, 20, and 30 epochs.

- **Spoofing Threshold**: This is a specific parameter for our problem, which defines the threshold above which an action is classified as spoofing. We tested thresholds of 0.7, 0.8, and 0.9.

The PPO configuration parameters used for the initial setup are as follows:

```
PPO_CONFIG = {
    'learning_rate': 2.5e-4,
    'n_steps': 2048,
    'batch_size': 64,
    'n_epochs': 10,
    'gamma': 0.99,
    'gae_lambda': 0.95,
    'clip_range': 0.2,
    'ent_coef': 0.01,
    'vf_coef': 0.5,
    'max_grad_norm': 0.5
}
```

Definition 5.1.1 (Learning Rate) *The learning rate is one of the most critical hyperparameters as it controls the step size at each iteration while moving toward a minimum of the loss function. If the learning rate is too high, the model might converge too quickly to a suboptimal solution. If the learning rate is too low, the training process may take an inordinately long time.*

We experimented with different learning rates to find a balance between convergence speed and the quality of the solution.

$$\alpha \in \{1 \times 10^{-4}, 5 \times 10^{-4}, 1 \times 10^{-3}\}$$

Definition 5.1.2 (Batch Size) *Batch size influences the stability and speed of the training process. Smaller batch sizes typically provide a regularizing effect and lower generalization error, while larger batch sizes can speed up the training process.*

We tested different batch sizes to find an optimal size that balances these aspects effectively.

$$\text{Batch Size} \in \{32, 64, 128\}$$

Definition 5.1.3 (Number of Epochs) *The number of epochs determines how many times the entire training dataset passes through the network. More epochs can lead to better learning but may also cause overfitting if not controlled properly.*

We adjusted the number of epochs to ensure that the model learned effectively without overfitting.

$$\text{Epochs} \in \{10, 20, 30\}$$

Definition 5.1.4 (Spoofing Threshold) *The spoofing threshold is unique to our application, defining the anomaly score threshold above which actions are classified as spoofing. Adjusting this threshold helps to balance the sensitivity and specificity of our spoofing detection.*

$$\textit{Spoofing Threshold} \in \{0.7, 0.8, 0.9\}$$

The tuning process involved running multiple training sessions with different combinations of these hyperparameters. For each combination, the model's performance was evaluated based on its ability to correctly identify spoofing attempts. The following algorithm outlines the tuning process:

Algorithm 7 Hyperparameter Tuning Process

```
procedure TuneHyperparameters
    for learning_rate in {1e-4, 5e-4, 1e-3} do
        for batch_size in {32, 64, 128} do
            for epochs in {10, 20, 30} do
                for spoofing_threshold in {0.7, 0.8, 0.9} do
                    Set PPO_CONFIG['learning_rate'] = learning_rate
                    Set PPO_CONFIG['batch_size'] = batch_size
                    Set PPO_CONFIG['n_epochs'] = epochs
                    Set Config.DEFAULT_SPOOFING_THRESHOLD = spoofing_threshold
                    Train the PPO Policy Network with the current configuration
                    Evaluate performance on the validation set
                    Record the performance metrics
                end for
            end for
        end for
    end for
    Select the configuration with the best performance metrics
end procedure
```

5.1.6 Evaluation Metrics

In order to assess the efficacy of our PPO model in detecting spoofing in financial markets, we implemented a series of evaluation metrics. The following points outline the methodology and results of our evaluation.

1. **Data Splitting**: We followed a classic reinforcement learning (RL) approach for data splitting, using 70% of the data for training and 30% for testing. This split ensures that the model has sufficient data for learning while preserving a significant portion for unbiased evaluation.

2. **Contiguous Data Segments**: Since our analysis involves time-series data, maintaining the continuity of data is crucial. Thus, we ensured that the training and testing segments are contiguous. This approach avoids any cross-contamination between training and testing datasets, which is essential for maintaining the temporal integrity of the data.

3. **Evaluation Metrics**:

- **Loss**: During training, the loss function, which combines the actor and critic losses with an entropy term for regularization, is a key metric. It guides the optimization process, ensuring that the policy improves iteratively. The lower the loss, the better the model is performing.

- **Anomaly Scores**: We monitored the anomaly scores over time to detect potential spoofing activities. The anomaly score is calculated based on predefined feature weights and market data features. Compared against our defined spoofing threshold, a bigger anomaly score determines an order to be marked as spoofed.

- **Cumulative Rewards**: Cumulative rewards provide insight into the overall performance and learning efficiency of the PPO agent. Higher cumulative rewards indicate better policy performance in terms of detecting spoofing and taking appropriate actions.

- **Reward Distribution**: The distribution of rewards during training reveals the effectiveness of the agent's actions. In our case, rewards are binary, with positive rewards for correct spoofing detection and negative rewards for false positives. The distribution of rewards can be visualized as follows:

5.1.7 Experimental Results and Performance Analysis

In this dataset, roughly 98.6% open orders were eventually canceled. According to [LPV23], *"a total of 248,796 order submission and cancellation records were identified, with 233,134 occurring within the active area, 8,302 within the passive area, and 7,360 outside"*. This leads to 97% of the order submissions and cancellations to occur within the active area, with only 3% within the passive area.

5.2 Application Development

5.2.1 Feature Specification

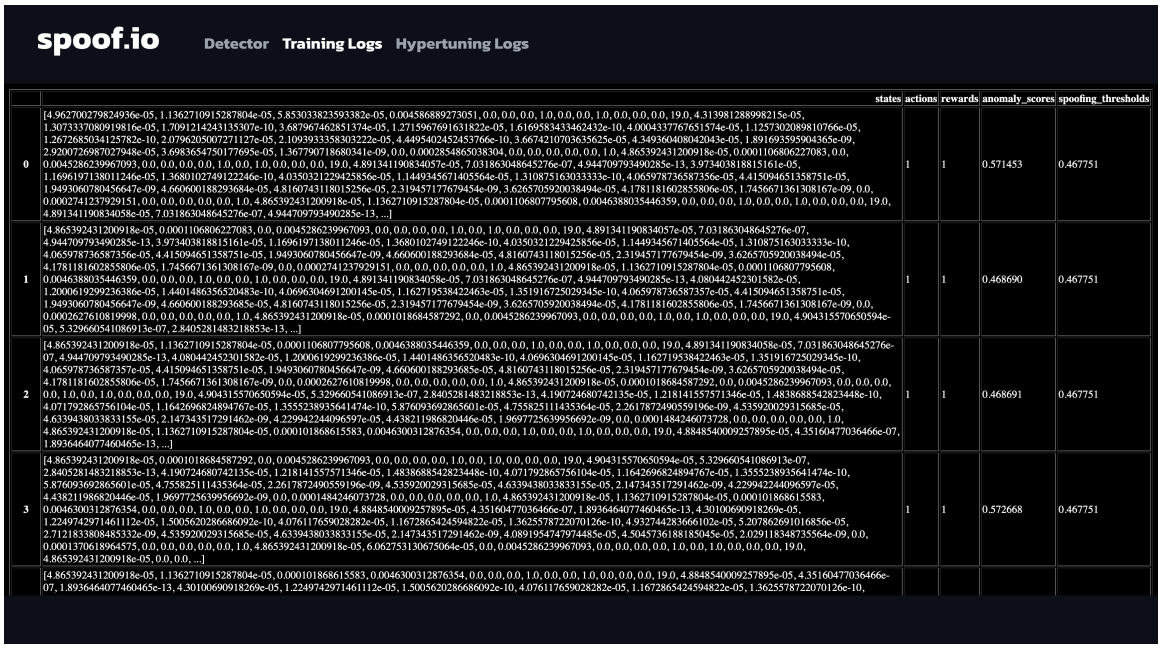
Figure 5.6: spoof.io User Interface

The web application, branded as *spoof.io*, is designed to offer a solution for monitoring and detecting spoofing attempts in the trading of LUNA cryptocurrency. The user interface is crafted to provide a real-time, intuitive experience for the users. The target market of the product consists of research analysts and LUNA investors that would like to identify spoofing attempts (even with the aim of taking legal action), or just to gain a better look of the current market situation in order to make informed decisions promptly before participating in the LUNA market.

To demonstrate the performance of our model, we need a great amount of spoofing attempts. Since market manipulation attempts do not occur on a daily basis, but only during exceptional market events (like flash crashes), for proof of concept purposes, we use simulated real historical Level 3 LOB data from the LUNA flash crash from May 2022. We simulate the last 30% of our data, which roughly amounts to two hours of simulation.

The navigation bar at the top provides easy access to training and hypertuning logs, ensuring users can review the model's performance and a comparative analysis of the training history at any time. This includes updates every time the model is trained or hypertuned, ensuring transparency and traceability. All evaluation metrics and logs can be seen in each section.

The interface of the *spoof.io* detector is divided into two primary sections: the live



	states	actions	rewards	anomaly_scores	spoofing_thresholds
0	[4.962700279824936e-05, 1.1362710915287804e-05, 5.853033823593382e-05, 0.0045868892730 51, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.313981288998215e-05, 1.3073337080919816e-05, 1.7091214243135307e-10, 3.6879674628513 74e-05, 1.2715967691631822e-05, 1.6169583433462432e-10, 4.0004337767651574e-05, 1.1257302089810766e-05, 1.2672685034127582e-10, 2.0796205007271127e-05, 2.1093933358303222e-05, 4.4495402452453766e-10, 3.6674210703635625e-05, 4.3493604080420 43e-05, 1.8916935959043365e-09, 2.9300726987027948e-05, 3.6983654750177695e-05, 1.3677907186803 41e-09, 0.0, 0.0, 0.0028548650383 04, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 0.0001106806227083, 0.0, 0.0045286239067093, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.891341190834057e-05, 7.031863048645276e-07, 4.944709793490285e-13, 3.973403818815161e-05, 1.1696197138011246e-05, 1.3680102749122246e-10, 4.0350321229425856e-05, 1.1449345671405564e-05, 1.3108751630333333e-10, 4.0659787365873 56e-05, 4.4150946513587515e-05, 1.9493060780456647e-09, 4.6606001882936 84e-05, 4.8160743118015256e-05, 2.3194571776794 54e-09, 3.6265705920038494e-05, 4.1781181602855806e-05, 1.7456671361308167e-09, 0.0, 0.0, 0.0002741237929151, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 1.1362710915287804e-05, 0.0001106807795608, 0.0046388035446359, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.891341190834058e-05, 7.031863048645276e-07, 4.944709793490285e-13, ...]	1	1	0.571453	0.467751
1	[4.865392431200918e-05, 0.0001106806227083, 0.0, 0.0045286239967093, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.891341190834057e-05, 7.031863048645276e-07, 4.944709793490285e-13, 3.973403818815161e-05, 1.1696197138011246e-05, 1.3680102749122246e-10, 4.0350321229425856e-05, 1.1449345671405564e-05, 1.3108751630333333e-10, 4.0659787365873 56e-05, 4.4150946513587515e-05, 1.9493060780456647e-09, 4.6606001882936 84e-05, 4.8160743118015256e-05, 2.3194571776794 54e-09, 3.6265705920038494e-05, 4.1781181602855806e-05, 1.7456671361308167e-09, 0.0, 0.0, 0.0002741237929151, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 1.1362710915287804e-05, 0.0001106807795608, 0.0046388035446359, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.891341190834058e-05, 7.031863048645276e-07, 4.944709793490285e-13, 4.080442452301582e-05, 1.2000619299236386e-05, 1.4401486356520483e-10, 4.0696304691200145e-05, 1.1627195384224 63e-05, 1.3519167250293 45e-10, 4.0659787365873 57e-05, 4.4150946513587515e-05, 1.9493060780456647e-09, 4.6606001882936 85e-05, 4.8160743118015256e-05, 2.3194571776794 54e-09, 3.6265705920038494e-05, 4.1781181602855806e-05, 1.7456671361308167e-09, 0.0, 0.0002627610819998, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 0.0001018684587292, 0.0, 0.0045286239967093, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.904315570650594e-05, 5.329660541086913e-07, 2.8405281483218853e-13, ...]	1	1	0.468690	0.467751
2	[4.865392431200918e-05, 1.1362710915287804e-05, 0.0001106807795608, 0.0046388035446359, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 19.0, 4.891341190834058e-05, 7.031863048645276e-07, 4.944709793490285e-13, 4.080442452301582e-05, 1.2000619299236386e-05, 1.4401486356520483e-10, 4.0696304691200145e-05, 1.1627195384224 63e-05, 1.3519167250293 45e-10, 4.0659787365873 57e-05, 4.4150946513587515e-05, 1.9493060780456647e-09, 4.6606001882936 85e-05, 4.8160743118015256e-05, 2.3194571776794 54e-09, 3.6265705920038494e-05, 4.1781181602855806e-05, 1.7456671361308167e-09, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 19.0, 4.904315570650594e-05, 5.329660541086913e-07, 2.8405281483218853e-13, 4.190724680742135e-05, 1.2181415575713 46e-05, 1.4838688542823448e-10, 4.0717928657561 04e-05, 1.1642696824894767e-05, 1.3555238935641474e-10, 4.6539243120091 8e-05, 1.1362710915287804e-05, 0.0001018684587292, 0.0, 0.0045286239967093, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 1.1362710915287804e-05, 0.0001018868615583, 0.0046300312876354, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.884854000925789 5e-05, 4.351604770364 66e-07, 1.8936464077460465e-13, ...]	1	1	0.468691	0.467751
3	[4.865392431200918e-05, 0.0001018684587292, 0.0, 0.0045286239967093, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.904315570650594e-05, 5.329660541086913e-07, 2.8405281483218853e-13, 4.190724680742135e-05, 1.2181415575713 46e-05, 1.4838688542823448e-10, 4.0717928657561 04e-05, 1.1642696824894767e-05, 1.3555238935641474e-10, 5.8760936928656 01e-05, 4.7558251114353 64e-05, 2.2617872490559 196e-09, 4.5359200293156 85e-05, 4.6339438033833 155e-05, 2.1473435172914 62e-09, 4.2299422440965 97e-05, 4.4382119868204 6e-05, 1.9697725639956 692e-09, 0.0, 0.0, 0.0001484246073728, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 1.1362710915287804e-05, 0.0001018684587292, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.884854000925789 5e-05, 4.351604770364 66e-07, 1.8936464077460465e-13, 4.301006909182 69e-05, 1.2249742971461112e-05, 1.5005620286686092e-10, 4.0761176590282 82e-05, 1.1672865424594 822e-05, 1.3625578722070 126e-10, 4.9327442836661 02e-05, 5.2078626910168 56e-05, 2.7121833808485332e-09, 4.5359200293156 85e-05, 4.6339438033833 155e-05, 2.1473435172914 62e-09, 4.0891954747974 485e-05, 4.5045736188185 045e-05, 2.0291183487355 64e-09, 0.0, 0.0001370618964575, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 4.865392431200918e-05, 6.062753130675 064e-05, 0.0, 0.0, 0.0045286239967093, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.865392431200918e-05, 0.0, 0.0, ...]	1	1	0.572668	0.467751
	[4.865392431200918e-05, 1.1362710915287804e-05, 0.0001018686615583, 0.0046300312876354, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 19.0, 4.884854000925789 5e-05, 4.351604770364 66e-07, 1.8936464077460465e-13, 4.301006909182 69e-05, 1.2249742971461112e-05, 1.5005620286686092e-10, 4.0761176590282 82e-05, 1.1672865424594 822e-05, 1.3625578722070 126e-10,				

Figure 5.7: Training Logs Tab

order feed and the spoof detection panel.

The live order feed displays simulated LUNA orders in real time, offering a dynamic and continuously updating view of market activities. This section includes the following fields:

- **Order ID:** A unique identifier for each order, allowing users to track specific transactions.
- **Type:** Indicates whether the order is received, open, or executed.
- **Size:** The volume of the order.
- **Price:** The price at which the order is placed.
- **Time:** The timestamp of the order, providing precise timing information.
- **Remaining Size:** The remaining volume of the order if it has not been fully executed.
- **Trade ID:** An identifier for trades, relevant for matched orders.
- **Reason:** Specifies the reason for the order status, such as canceled or filled.

This feed ensures that users have a granular view of the order book, enabling detailed monitoring and analysis of trading activities.

The spoof detection panel is dedicated to displaying identified spoofing attempts, providing critical insights into potential market manipulations. This section includes the following fields:

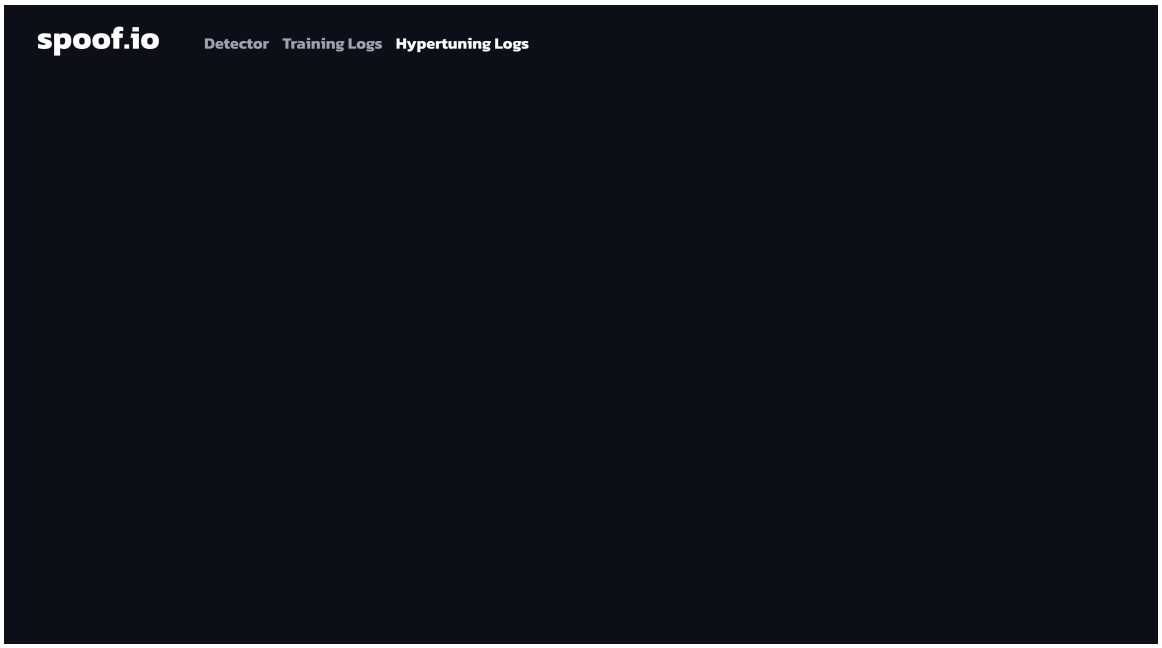


Figure 5.8: Hypertuning Logs Tab

- **Detected Spoofed Order ID:** The unique identifier of the order detected as spoofed.
- **Time:** The timestamp of when the spoofing attempt was detected.
- **Anomaly Score:** A percentage indicating the likelihood of the order being a spoof, derived from the model's anomaly detection algorithm.
- **Spoofing Threshold:** The threshold value used to classify the order as spoofed, providing context for the anomaly score.

The spoof detection panel is designed to alert users to suspicious activities, ensuring they can take necessary actions to mitigate risks associated with market manipulations. We employed a severity color grading feature for the detected spoofed instances, to facilitate the analysis process:

- **Informational Spoofs:** Color-graded with green - in terms of our model, the detection score is passing the spoofing threshold with **0-5%**.
- **Moderate Spoofs:** Colored in dark yellow - spoofing threshold is passed with **5-10%**.
- **Critical Spoofs:** Marked with red - anomaly score is with **more than 10%** higher than the spoofing threshold.

5.2.2 User Guide

Assuming all required packages are installed on the machine, *spoof.io* can be set up locally as well.

Ensure that the following software is installed in a virtual environment:

- Redis
- Daphne
- Node.js and npm
- Python and required dependencies:

```
$ pip install -r requirements.txt
```

Steps for local setup:

1. For session management and synchronisation, start the Redis server:

```
$ redis-server
```

2. To start the ASGI Django server, via Daphne, run the following command:

```
$ daphne -p 8000 backend.asgi:application
```

3. The React frontend node is initialised with:

```
$ npm start
```

Once the web application is loaded, the user lands on the main page containing the spoofing detector. A guide on how to analyse the live feed order entries and the detection outcome (including severity classification) is detailed in the *"Feature Specification"* section 5.2.1.

Even though the most recent orders (both LOB entries and detected spoofs) are stacked in the tables for real-time display, past data can be accessed by scrolling (respectively swiping on mobile devices) through the entries.

To improve the utility of our tool, for an easier identification of a spoofed order (or any kind of particularly useful LOB information), the content of any table field is automatically copied in the clipboard once the field is clicked (on bigger screens) or tapped (on smaller devices).

Additionally, for responsiveness considerations, the order data (including table header details) might be truncated using an ellipsis. This does not affect the UX of the product, as by hovering over any field, a tooltip is shown for displaying the full content of the field.

The training logs can be visualised accessed from the navigation bar at the top of the dashboard. The user can toggle between the detector view, the training logs, and the hypertuning logs, respectively.

Detected Spoofed Order ID	Time	Anomaly Score	Spoofing Threshold
e62ec586-ac37-42dd-a2bf...	2022-05-11T19:05:38.43673...	66.79%	66.70%
e62ec586-ac37-42dd-a2bf...	2022-05-11T19:05:38.43673...	66.79%	66.70%
e62ec586-ac37-42dd-a2bf...	2022-05-11T19:05:38.43673...	66.79%	66.70%
e62ec586-ac37-42dd-a2bf...	2022-05-11T19:05:38.43673...	66.79%	66.70%
a753a816-b363-45b9-8f7f...	2022-05-11T19:05:43.95037...	75.90%	64.23%
a753a816-b363-45b9-8f7f...	2022-05-11T19:05:43.95037...	75.90%	64.23%
3c4aa075-cc86-494d-af62...	2022-05-11T19:05:43.95037...	55.66%	53.26%
3c4aa075-cc86-494d-af62...	2022-05-11T19:05:43.95037...	55.66%	53.26%
267c7286-c7de-421b-b516-...	2022-05-11T19:05:43.95037...	63.82%	53.26%
267c7286-c7de-421b-b516-...	2022-05-11T19:05:43.95037...	63.82%	53.26%
a753a816-b363-45b9-8f7f...	2022-05-11T19:05:43.95037...	75.90%	64.23%
a753a816-b363-45b9-8f7f...	2022-05-11T19:05:43.95037...	75.90%	64.23%
3c4aa075-cc86-494d-af62...	2022-05-11T19:05:43.95037...	55.66%	53.26%
3c4aa075-cc86-494d-af62...	2022-05-11T19:05:43.95037...	55.66%	53.26%
267c7286-c7de-421b-b516-...	2022-05-11T19:05:43.95037...	63.82%	53.26%
267c7286-c7de-421b-b516-	2022-05-11T19:05:43.95037	63.82%	53.26%

Figure 5.9: Accessing spoof entries from the past two hours, by scrolling in the orders box logs.

Figure 5.10: Revealing full content of an order ID.

5.2.3 Backend Architecture and Data Flow

The backend architecture of the spoof.io web application is designed to facilitate real-time detection and visualization of spoofing attempts in algorithmic trading. The core of this architecture leverages Django REST Framework (DRF) for API development, along with several other key technologies to ensure efficient data processing and model integration.

Our backend is built on Django REST Framework. DRF simplifies the development of RESTful APIs and provides robust features for serialization, authentication, and view handling. Additionally, we utilise a Redis server for caching and as a message broker. Redis enhances the performance of our application by storing frequently accessed data in memory, reducing the need for repeated database queries. It also facilitates communication between different components of the sys-

Figure 5.11: spoof.io Navbar

tem through its pub/sub capabilities.

For handling WebSocket connections, we use Daphne, an HTTP, HTTP2, and WebSocket protocol server for ASGI and ASGI-HTTP, developed by Django. Daphne allows our application to maintain persistent WebSocket connections, enabling real-time data transmission between the backend and frontend. The application uses the ASGI (Asynchronous Server Gateway Interface) specification, which is designed to handle asynchronous and synchronous web protocols. ASGI provides the scalability required for real-time applications.

WebSocket connections are the key for real-time data transmission in our application. They allow the server to push updates to the client instantly, ensuring that the live order feed and spoofing detection notifications are up-to-date. The WebSocket connection is managed by the `OrderConsumer` class in `consumers.py`. When a client connects to the WebSocket endpoint (`ws/orders/`), an instance of `OrderConsumer` is created. This instance joins the `order_group` channel layer, enabling it to receive and send messages to all clients connected to this group. Orders are transmitted in real-time through WebSocket messages. The `send_order` function in `simulation.py` is responsible for sending order data to the WebSocket. This function converts order data into JSON format and sends it to the `order_group`, which then broadcasts the message to all connected clients.

Algorithm 8 WebSocket Connection for Real-Time Order Processing

```
initialize OrderConsumer
define connect()
    room_group_name ← 'order_group'
    await channel_layer.group_add(room_group_name, channel_name)
    await accept()
    create_task(start_simulation())
define disconnect(close_code)
    await channel_layer.group_discard(room_group_name, channel_name)
define order_message(event)
    message ← event['message']
    await send(text_data=message)
define start_simulation()
    await simulate_market_data()
define send_order(order, is_spoof=False)
    channel_layer ← get_channel_layer()
    order_dict ← convert_order_to_dict(order)
    if is_spoof
      order_dict.update({'is_spoofing': True, 'anomaly_score': order['anomaly_score'], 'spoofing_threshold': order['spoofing_threshold']})
    await channel_layer.group_send('order_group', {'type': 'order.message', 'message': json.dumps(order_dict)})
```

The data flow in our simulation involves several key steps to ensure that raw Limit Order Book (LOB) data is processed, analyzed, and displayed accurately. Raw LOB data entries are simulated in real time, replicating the exact timestamps using the `simulate_market_data` function in `simulation.py`. This function reads historical LOB data and replays it with the original time intervals between orders. Each order is sent via WebSocket in real-time to be displayed in the live order feed on the frontend. This is handled by the `send_order` function, which transmits the data to the `order_group`.

Orders are collected in batches of 15, which is necessary for feature engineering. Once a batch is complete, the simulation proceeds with preprocessing and feature extraction. This preprocessing involves converting raw data into a structured format, followed by the computation of statistical features such as mean, standard deviation, and variance.

The processed batch is then fed into the `MarketEnvironment`, a class designed

to simulate the trading environment, which interacts with the `PPOPolicyNetwork`, our trained Proximal Policy Optimization model. The `MarketEnvironment` initializes with the processed data and resets to prepare for the new batch.

The `test_model` function is responsible for running the PPO model against the market environment. It resets the environment and enters a loop where it processes each state in the batch. For each state, the model predicts the next action by sampling from a categorical distribution of logits produced by the `PPOPolicyNetwork`. The environment then steps through the action, returning the next state, transaction data, reward, done status, anomaly score, and spoofing threshold.

If an order is identified as spoofed, it is sent via WebSocket to be displayed in the spoofing attempts box on the frontend. The `send_order` function enriches the order data with the anomaly score and spoofing threshold before broadcasting it.

5.2.4 Frontend Interface and User Experience

The frontend of our application is developed using React with TypeScript and SCSS. The choice of React facilitates the creation of reusable components, enhancing development efficiency and ensuring consistency across the user interface. TypeScript adds an additional layer of type safety, reducing runtime errors and improving code quality. SCSS is utilized for styling, enabling nested rules, variables, and mixins, which streamline the CSS management process.

Responsiveness is a critical aspect of our frontend design, ensuring that the application is accessible and usable across various devices and screen sizes. The interface is designed to effortlessly adapt to screen widths ranging from mobile devices (320px) to large desktop monitors (1920px). This adaptability is achieved through a combination of flexible grid layouts, media queries, and responsive units. Our application is composed of several React components, including `Hero`, `Navbar`, `OrdersBox`, and `TableHeader`.

The `OrdersBox` component is designed to be highly reusable, serving both the live order feed and the detected spoofing attempts box. Depending on the type of orders to display (`regular` or `spoofing`), the component populates the table with the relevant data; the component leverages props to customize its behavior and appearance based on the data it receives.

The OrdersBox component dynamically updates through a WebSocket connection, providing real-time data feeds. The WebSocket is constructed to listen for incoming messages from the server, parsing and categorizing the orders into regular and spoofing orders based on the received data. The WebSocket connection is established using the `useEffect` hook in the `WebSocketProvider` component. This ensures that the connection is initiated when the component is mounted

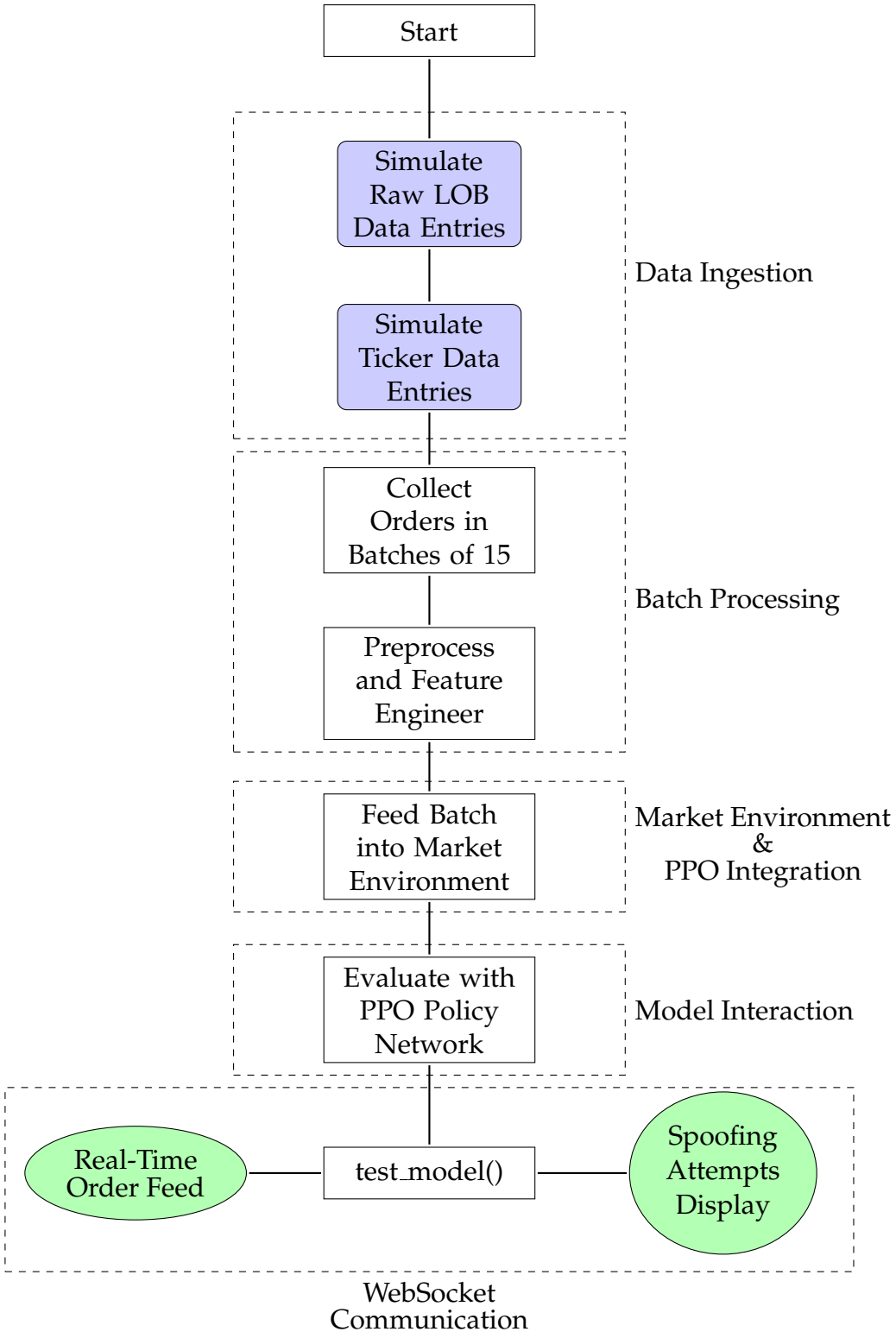


Figure 5.12: Server-side WebSocket Setup Process

and closed when it is unmounted, maintaining an efficient use of resources. This context is used to subscribe to live updates and dispatch new orders to the relevant components. The WebSocket connection ensures that the data displayed in the `OrdersBox` component is updated in real-time, ”stacking” the most recent orders

in a `column-reverse flex` display.

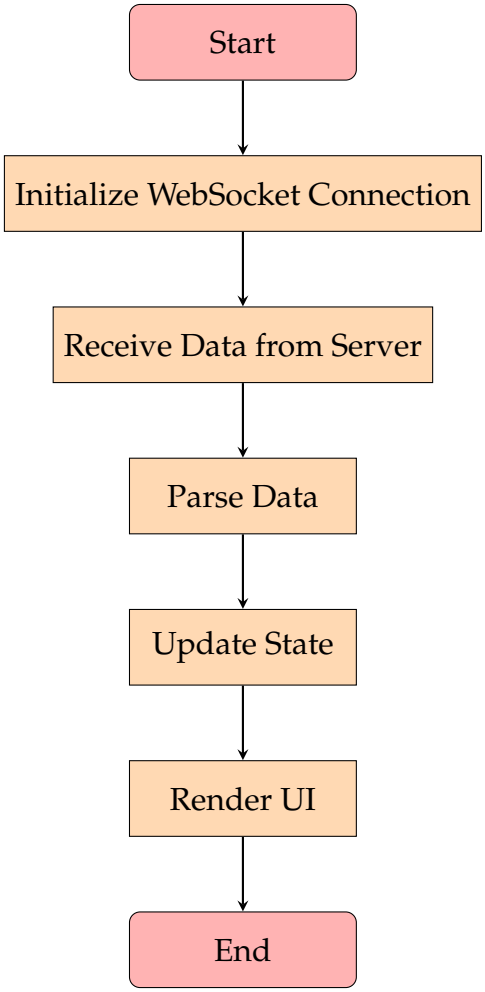


Figure 5.13: Client-side WebSocket Setup Process

The user experience is further enhanced through several UX elements:

- **Tooltips**: Implemented using the `react-tooltip` library, tooltips provide additional context and information when users hover over specific elements. This feature is particularly useful for displaying detailed order information without cluttering the interface.
- **Ellipsis for Text Overflow**: Long text strings are truncated with an ellipsis to prevent overflow and maintain a clean layout. This is achieved using CSS properties such as `"white-space: nowrap"`, `"overflow: hidden"`, and `"text-overflow: ellipsis"`.
- **Severity Color Grading**: Orders identified as spoofing attempts are color-coded based on the severity of the anomaly. This visual cue helps users quickly assess the risk level associated with each order. The color grading is implemented using conditional rendering in React, where the severity level dictates the CSS class applied to each order entry.

5.2.5 Testing

The client-side testing process was primarily conducted using Jest, and it encompassed the following major areas:

1. Unit Testing: Unit tests were written to verify the functionality of individual components and utility functions. For example, in the `OrdersBox.test.tsx` file, various test cases ensured that the `OrdersBox` component correctly rendered order entries and displayed the appropriate severity color grading based on the anomaly scores. Similarly, the `Navbar.test.tsx` and `TableHeader.test.tsx` files contained tests to validate the rendering and interaction of the navigation bar and table headers, respectively.

2. Integration Testing: Created tests to validate the WebSocket connection and message handling within the context provider. These tests simulated real-time data transmission and ensured that the `OrdersBox` component updated correctly when new orders were received.

3. Snapshot Testing: Snapshot tests were utilized to capture the rendered output of components and detect any unexpected changes in the UI. This approach was particularly useful for maintaining the visual consistency of components such as `Hero`, `HypertuningLogs`, and `TrainingLogs`. By comparing the current render output to the stored snapshots, any unintended modifications could be promptly identified and addressed.

4. WebSocket Testing: Given the critical role of WebSocket communication in the application, extensive testing was performed to ensure the reliability and performance of real-time data transmission. The `WebSocketContext.test.tsx` file includes tests that mocked the WebSocket server and simulated various scenarios, such as establishing a connection, receiving messages, and handling errors. These tests ensured that the WebSocket context correctly managed the state and propagated updates to the connected components.

5. Mocking and Simulations: To isolate components and test their behavior independently, mocking techniques were employed extensively. Mocked functions and objects were used to simulate API responses, WebSocket messages, and other dependencies.

5.2.6 Deployment

5.3 Future Considerations

Chapter 6

Conclusions

Bibliography

[AAD+16] John Armour, Dan Awrey, Paul Davies, Luca Enriques, Jeffrey N. Gordon, Colin Mayer, and Jennifer Payne. 181 Trading and Market Integrity. In *Principles of Financial Regulation*. Oxford University Press, 07 2016.

[CEA13] Commodity exchange act (cea). Section 4c(a)(5)(C), 2013. Accessed: 2024-03-26.

[CLC+15] Yi Cao, Yuhua Li, Sonya Coleman, Ammar Belatreche, and Thomas Martin McGinnity. Adaptive hidden markov model with anomaly states for price manipulation detection. *IEEE Transactions on Neural Networks and Learning Systems*, 26(2):318–330, 2015.

[Dod10] Dodd-frank wall street reform and consumer protection act. Pub. L. No. 111-203, 124 Stat. 1376, 2010. Available at U.S. Government Publishing Office.

[Dur10] Michael Durbin. *All About High-Frequency Trading*. McGraw-Hill, US, 2010.

[EL22] Shengbo Eben Li. *Reinforcement Learning for Sequential Decision and Optimal Control*. Springer, 2022.

[FIN08] Finra rule 2020, 2008. Accessed: 2024-03-26.

[Gor17] Matthew Gormley. Reinforcement learning introduction. Lecture Notes for 10-601: Machine Learning, Carnegie Mellon University, 2017. Accessed: 2024-04-01.

[HR13] Terrence Hendershott and Ryan Riordan. Algorithmic trading and the market for liquidity. *Journal of Financial and Quantitative Analysis*, 48(4):1001–1024, 2013.

[Huy22] Chip Huyen. *Designing machine learning systems*. " O'Reilly Media, Inc.", 2022.

[Kim61] Gregory A Kimble. Hilgard and marquis'" conditioning and learning.". 1961.

[KT00] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, volume 12, 2000.

[LDF05] FABRIZIO LILLO and J. DOYNE FARMER. The key role of liquidity fluctuations in determining large price changes. *Fluctuation and Noise Letters*, 05(02):L209–L216, 2005.

[LMS23] Jiageng Liu, Igor Makarov, and Antoinette Schoar. Anatomy of a run: The terra luna crash. Working Paper 31160, National Bureau of Economic Research, April 2023.

[LPV23] Haochen Li, Maria Polukarova, and Carmine Ventre. Detecting financial market manipulation with statistical physics tools, 2023.

[LTT16] Teema Leangarun, Poj Tangamchit, and Suttipong Thajchayapong. Stock price manipulation detection using a computational neural network model. In *2016 Eighth International Conference on Advanced Computational Intelligence (ICACI)*, pages 337–341, 2016.

[MAR14] Market abuse regulation (mar). European Union Regulation, 2014. Accessed: 2024-03-26.

[MDGH13] Stacy Williams Mark McDonald Daniel J. Fenn Martin D. Gould, Mason A. Porter and Sam D. Howison. Limit order books. *Quantitative Finance*, 13(11):1709–1742, 2013.

[Mon16] John D. Montgomery. Spoofing, market manipulation, and the limit-order book. May 2016. Available at SSRN: `https://ssrn.com/abstract=2780579` or `http://dx.doi.org/10.2139/ssrn.2780579`.

[Ope17] Openai baselines: Ppo, 2017. Accessed: 2024-04-01.

[Ope24] OpenAI. Openai charter, 2024. Accessed: 2024-03-26.

[RMT17] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network, 2017.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[SEA34] Securities exchange act of 1934. Section 10(b), 1934. Accessed: 2024-03-26.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR.

[Steon] Steel Eye. Spoofing: A growing market manipulation risk and focus for regulators, Year of Publication. Accessed: 2024-03-10.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[TNS21] Jean-Noel Tuccella, Philip Nadler, and Ovidiu Serban. Protecting retail investors from order book spoofing using a gru-based detection model, 2021.

[vOW12] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[WHVW21] Xintong Wang, Christopher Hoang, Yevgeniy Vorobeychik, and Michael Wellman. Spoofing the limit order book: A strategic agent-based analysis. *Games*, 12:46, 05 2021.

[Wro84] Andrew Wrobel. On markovian decision models with a finite skeleton. *Zeitschrift für Operations Research*, 28:17–27, 1984.

[YTST14] Yoshihiro Yura, Hideki Takayasu, Didier Sornette, and Misako Takayasu. Financial brownian particle in the layered order-book fluid and fluctuation-dissipation relations. *Phys. Rev. Lett.*, 112:098703, Mar 2014.