Maestría en Finanzas

Defy the Game: Automated Market Making using Deep Reinforcement Learning

Autoría: Parrotta, Agustín
Año académico: 2023

UNIVERSIDAD TORCUATO DI TELLA

Trabajo Final de Graduación

Maestría en Finanzas UTDT

Formato elegido: trabajo académico

Defy the Game: Automated Market Making using Deep Reinforcement Learning

Año Académico 2022

Alumno: Agustin Parrotta

Tutor: Pablo Roccatagliata

Mayo 2023

Abstract

Automated market makers have gained popularity in the financial market for their ability to provide liquidity without needing a centralized intermediary (market maker). However, they suffer from the problems of slippage and impermanent loss, which can lead to losses for both liquidity providers and takers. This work implements a pseudo-arbitrage rule to solve the impermanent loss issues related to arbitrage opportunities. The mechanism implements a trusted external oracle to get the market conditions, put them on the automated market maker, and match the bonding curve to them. Next, the application of a Double Deep Q-Learning reinforcement learning algorithm is proposed to reduce these issues in automated market makers. The algorithm adjusts the curvature of the bonding curve function to adapt to market conditions quickly. This work describes the model, the simulation environment used to learn and test the proposed approach, and the metrics used to evaluate its performance. Finally, it explains the results of the experiments and analysis of their implications. The approach shows promise in reducing slippage and impermanent loss and recommending improvements and future works.

Keywords

Automated Market Maker, Liquidity Provider, Liquidity Taker, Divergence Loss, Slippage, Computational Finance, Unsupervised Learning, Deep Reinforcement Learning, Markov Decision Process, Q-Learning.

Table of Contents

Table of Figures

1. Introduction

In recent years, along with the emergence, evolution, and global adoption of the blockchain ecosystem and cryptocurrencies, Decentralized Finance (DeFi) applications have been gaining popularity. Currently there are about 6.68 million users, with a growth of 34% compared to January 2022 and more than 400% compared to January 2021. Regarding the market size, the Total Value Locked (TVL) is USD 77.29 billion, peaking at the end of 2021, with a value of USD 165.47 billion (Raynor de Best, 2023).

DeFi's rise preserves the general principles of scalability, security, and decentralization, which are supported by smart contracts on public blockchains, allowing the creation of an entire financial system in which different parties can operate under shared data and assumptions without trust issues caused by the institutional intervention (Lim, 2022).

1.1. Decentralized Exchanges

Decentralized Exchanges (DEXs) are an essential component of the DeFi ecosystem. A DEX is a cryptocurrency exchange type that operates on a decentralized platform, typically on a blockchain network. In a DEX, trades are executed through a smart contract that usually operates as an automated market maker. Unlike centralized exchanges (CEXs), DEXs do not rely on a central authority to maintain an order book. Instead, they allow users to trade crypto assets using a peer-to-peer mechanism without the need for intermediaries (Bartoletti et al., 2022).

In a CEX, trades are executed through an intermediary that acts as a trusted third party, controlling the exchange and its users. Clear examples of these exchanges are Binance, CoinBase, and Bitfinex. CEXs typically have a higher trading volume and liquidity than DEXs, as they allow for faster transaction speeds and can support a more significant number of trading pairs. However, centralized exchanges are also more susceptible to hacks and security breaches, as the centralization of control creates a single point of failure. Furthermore, CEX exchanges often require users to provide sensitive personal information, such as identity verification, which can compromise their privacy and security. There have been numerous events in the last years where the security of these CEXs was compromised, resulting in millions of dollars in losses and theft of information.

DEXs offer several advantages over CEXs, including increased security and privacy, as trades are executed through a trustless system that does not rely on intermediaries or require users to provide personal information. DEXs also enable users to control their funds, as they typically use non-custodial wallets that allow users to hold their private keys.

1.2. Automated Market Makers

Similar to centralized exchanges like Binance, FTX, and Coinbase, decentralized exchanges started employing the Limit Order Book to facilitate trades (Khakhar & Chen, 2022). These books maintain two lists of sorted information, one for buying (bid) and one for selling (ask), including the prices and amount that traders are willing to buy or sell (Biais et al., 1995). Liquidity for these books is accomplished by market participants adding orders to the book, such as a bid order to buy a certain number of shares at a certain price or an ask order to sell a specified number of shares at a certain price. The market participant who adds liquidity to the Limit Order Book is a Market Maker (MM).

Once an order is placed, another participant has the option to accept the entire offer or only a portion of the shares. The participant who accepts the bid or ask and removes liquidity from the Limit Order Book is referred to as the Market Taker. Due to the high fees associated with maintaining a Limit Order Book on the Ethereum chain, placing and updating orders on the book became excessively costly. As a result, a new solution was introduced: Automated Market Makers (AMMs), which have primarily replaced Limit Order Books in most DEXs (Khakhar & Chen, 2022). In an AMM-based DEX, individual buy and sell orders are not required because traders interact directly with protocol smart contracts so that trades execute automatically against a liquidity pool. The AMM system uses a mathematical formula known as a bonding curve or conservation function to determine the price of a given asset based on its supply and demand.

AMM systems have become popular in DEXs due to their ability to provide liquidity without intermediaries. They offer several benefits, including decentralization, automation, and continuous liquidity. In a traditional Order Book-based exchange, the market price of an asset is determined by the last matched buy and sell orders, which are driven by the supply and demand of the asset. However, in an AMM-based DEX, a liquidity pool acts as a single counterparty for each transaction. Asset pricing is algorithmically given by the bonding curve function, which only allows the price to move along predefined trajectories. This means that users can obtain immediate liquidity without finding an exchange counterparty, while liquidity providers (LPs) can profit from asset supply with exchange fees from users (traders). Additionally, since AMMs use a conservation function for price setting, maintaining an order book on a distributed ledger is unnecessary (Xu et al., 2022).

When a user wants to trade a cryptocurrency asset, the AMM system automatically matches the trade with the liquidity provided by other users. The AMM system uses the bonding curve to calculate the asset price based on the ratio of the assets in the pool and the trading fees charged by the system. As the number of users and trades increases, the asset's price adjusts accordingly, ensuring the market remains liquid and efficient.

AMMs offer accessible liquidity provision and exchange for illiquid assets to benefit liquidity providers (LPs) and traders. However, they come with implicit economic risks such as high slippage and divergence loss, which can result in losses for exchange users and LPs, respectively. Additionally, AMM-based Decentralized Exchanges (DEXs) face numerous security and privacy issues. Over the past few years, new protocols have been introduced with incremental improvements to address these issues and cater to new use cases. Despite their innovative features, most AMM protocols share a similar structure consisting of composed mechanisms allowing multiple system functionalities. The main differences between these protocols are parameter choices and mechanism adaptations (Xu et al., 2022).

There are numerous AMM-based DEXs currently in operation, with new ones emerging regularly:

- Uniswap: Uniswap is a DEX that uses an automated market maker (AMM) to facilitate trades. It allows users to trade any ERC-20 token on the Ethereum blockchain. Uniswap is one of the most popular DEX, with a daily trading volume of around $1 billion (Uniswap, 2023).
- SushiSwap: SushiSwap is a fork of Uniswap that aims to offer more features and incentives to users. It also uses an AMM model and rewards liquidity providers in its native token, SUSHI.
- PancakeSwap: PancakeSwap is built on the Binance Smart Chain, which offers lower fees and faster transactions than Ethereum. It uses an AMM model and allows users to stake and farm its native token, CAKE.
- Curve Finance: Curve specializes in stablecoins trading, providing low-slippage trades for stablecoin pairs. It uses a Hybrid Function Market Maker protocol.
- Balancer: Balancer is a multi-token AMM that allows users to create custom pools with up to eight tokens with customizable weights. It also offers liquidity providers incentives in its native token, BAL.

1.3. AMM participants

Market participants in an Automated Market Maker (AMM) can be broadly divided into two categories:

- Liquidity taker
- Liquidity provider

1.3.1. Liquidity taker or trader

A trader is any user that exchanges assets by taking liquidity from the market supplied by liquidity makers. The price of an asset is determined by the bounding curve that defines the relationship between the quantities of each asset in the pool. AMM protocols execute trades through liquidity pools for each pair of tradable tokens reserved in their respective smart contracts. A trader seeking to exchange X tokens for Y tokens can deposit X tokens in the liquidity pool and receive Y tokens in an atomic swap, ensuring that the pool's aggregate liquidity remains unchanged, as defined by the bonding curve. This pricing function determines the exchange rate at which the tokens are swapped (Lim, 2022).

The traders anticipate that the market will reflect the actual value of assets, with little price change during trade execution (or slippage) and the ability to exchange assets on demand. When a trader makes a trade, they pay a fee to the liquidity pool proportional to their trade size. In return, the pool provides the trader with the desired asset at the current market price.

There is a type of trader known as an arbitrageur. These arbitrageurs identify asset pricing differentials between different exchanges and trade these differentials to profit. When the exchange rate of a token pair differs from other exchange quoted prices, AMM protocols allow arbitrageurs to execute arbitrage trades to bring the exchange rate back into line with general market conditions (Aoyagi, 2020).

1.3.2. Liquidity provider

Liquidity providers contribute assets to liquidity pools that are used to facilitate trading on the AMM. They establish an efficient market in which liquidity takers can trade assets. Liquidity providers contribute pairs of X and Y crypto-assets to the pool so traders can buy or sell X and Y crypto-assets. In exchange for their contribution, LPs receive a portion of the trading fees charged on the exchange as the incentive mechanism. The fees earned by LPs are proportional to the amount of liquidity they provide to the pool. LPs can withdraw their liquidity at any time, subject to a penalty fee in some cases. LPs help to ensure that there is sufficient liquidity in the pool to facilitate trades. In contrast, traders help to establish the market price of assets by engaging in buying and selling activities.

Figure 1 shows a general diagram of the AMM and its participants.

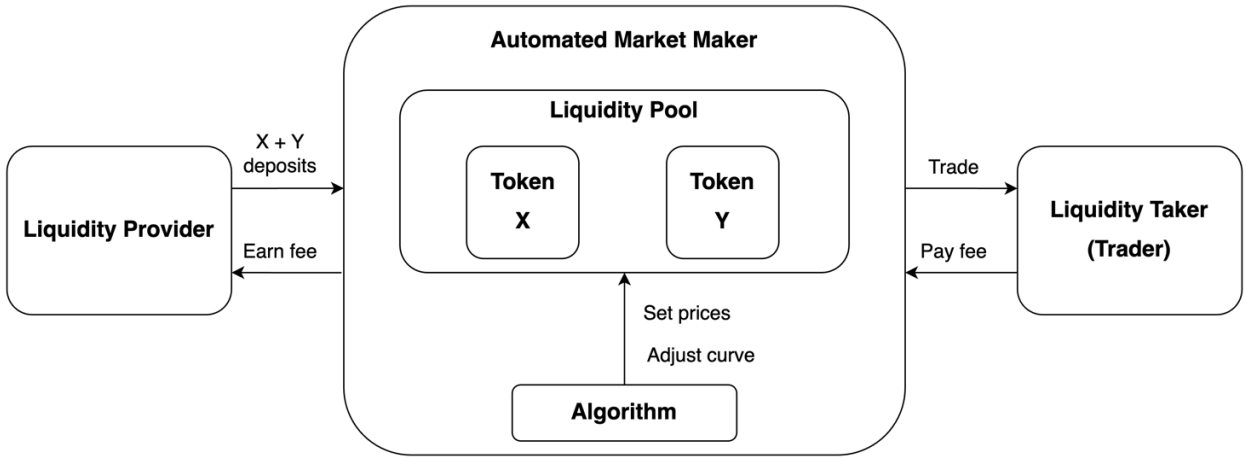

Figure 1. Diagram of an Automated Market Maker and its relationship with participants (own figure).

1.4. Common issues with AMMs

There are common issues among the existing AMMs that affect the stakeholders. One of them is the low capital efficiency, which is a function of the capital needed for efficient market making. Generally, it requires high capital to achieve the same performance as an order book-based exchange. This is because a significant portion of AMM liquidity is only available when the pricing curve starts to turn exponential. Consequently, rational traders will never utilize most liquidity due to the extreme price impact.

Another issue is the slippage loss. It refers to the implicit cost incurred by a liquidity taker when the actual execution price of a trade deviates from the expected price. This deviation can occur due to market volatility or when the trade sizes are relatively large compared to the size of the liquidity pool. Although it is

impossible to eliminate slippage, reducing this market inefficiency to the lowest possible level is to the benefit of liquidity takers.

Finally, the impermanent loss is a cost that liquidity providers incur. It occurs when the price of the two assets in the liquidity pool changes. If the price of one asset increases relative to the other, traders will buy more of the first asset and sell more of the second asset, causing the liquidity pool to become unbalanced. As a result, the liquidity provider's share of the pool shifts towards the less valuable asset than the value of the initially deposited assets, leading to a temporary loss of value. This loss is impermanent because it is only realized if the liquidity provider decides to withdraw their liquidity from the pool when the asset prices are not in balance. Impermanent loss is a significant concern for liquidity providers because it can result in them earning less than if they had held the two assets separately, even if the liquidity pool has earned fees from trades.

1.5. Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning, which has gained increasing relevance in recent years in various studies applied to the field of finance, particularly in the context of optimal market making (Carlsson & Regnell, 2022). The algorithm involves an agent learning how to make decisions through interactions with an environment. The goal of a reinforcement learning algorithm is to find the optimal actions that the agent can take in order to maximize a long-term cumulative reward signal. Unlike other machine learning techniques, reinforcement learning interacts in a trial-and-error way with its environment. Given a state of the world, the agent takes actions in the environment, which response with a reward signal that depends on the agent's actions. The agent's goal is to learn a policy that maps observations of the environment to actions that maximize the cumulative reward over time. Reinforcement learning is especially useful in situations where the optimal solution is not known, and the agent must learn through exploration. In finance, RL has been applied to a variety of tasks, including portfolio optimization, algorithmic trading, risk management, and fraud detection. While considering an economic problem, traditional approaches may not consistently achieve optimal performance.

One of the critical components is the reinforcement signal, known as a reward. This reward is a numerical value that the agent receives from the environment, representing the action's quality. The reinforcement signal can be positive or negative, depending on whether the action the agent takes leads to a desired outcome. The agent's objective is to maximize the cumulative reward it receives over time. To achieve this, reinforcement learning algorithms typically use a value function or a policy function. The value function estimates the expected long-term reward for a given state and action, while the policy function maps states to actions. The agent uses these functions to select the action most likely to lead to the highest reward.

However, while reinforcement learning methods it has seen success in recent years, its scalability is limited, and it struggles with high-dimensional problems. This is where Deep Reinforcement Learning (DRL) comes in, which combines reinforcement learning with Deep Learning (DL). DL utilizes the robust function approximation and representation learning properties of deep neural networks (DNNs) to handle complex and nonlinear patterns in data. By incorporating these methods, DRL can efficiently overcome the limitations of traditional RL approaches (Singh et al., 2022).

Despite its successes, many challenges must be addressed. One of the main ones is the issue of exploration versus exploitation. The agent must balance the desire to explore new actions with the need to exploit the actions that leads to high rewards. Another challenge is the issue of generalization. The agent must be able to generalize their knowledge to new situations rather than simply memorizing the actions that led to high rewards in specific situations. Reinforcement learning is a powerful approach to machine learning that has shown great promise in a wide range of applications. By learning from experience, reinforcement learning algorithms can adapt to changing environments and learn how to make decisions that lead to the highest reward. As the field continues to develop, reinforcement learning is likely to become a more crucial tool for solving complex problems.

1.6. Objective

The objective of this work is to solve impermanent loss and slippage issues within a generic AMM framework. First, it adopts from the existing literature the application of a pseudo-arbitrage rule to resolve the impermanent loss issues associated with arbitrage opportunities. The mechanism implements a trusted external oracle to get the market conditions on the AMM and match the bonding curve to them. Next, it implements a Double Deep Q-Network (D-DQN) reinforcement learning algorithm to price swaps orders, aiming to minimize the remaining impermanent loss and slippage concerns within the AMM.

This work starts by providing an overview of related studies in the AMM and RL fields, establishing a comprehensive understanding of the existing knowledge gaps. Then, it exposes the methodology adopted and the proposed D-DQB developed system, with the main findings and results. Finally, it discusses and concludes the work.

2. Literature Review

Although there is limited research that applies RL models to AMMs, some studies have emerged in the past two years. However, a number of studies have explored the different challenges in AMM and reinforcement learning separately.

2.1. AMM models

Xu et al. (2022) described the economics of an AMM, formalized the system's state-space representations, proposed a general framework, and compared it to the existing protocols, including fees system and implicit costs such as divergence and slippage losses. Wang (2020) compared mathematical models for AMM, such as the logarithmic market scoring rule (LMSR), liquidity-sensitive LMSR (LS-LMSR), and constant product/mean/sum, and proposed a constant ellipse-based cost function.

Bartoletti et al. (2022) formalized the fundamental properties of the AMMs, characterizing both structural and economic aspects. He also devised a general solution to the arbitrage problem.

Aoyagi (2020) analyzed the coexistence between a centralized limit-order book and an AMM and proposed an equilibrium valuation point for more accurate pricing in the AMM.

Engel and Herlihy (2021) investigated how an AMM's formula affects the interests of liquidity providers, providing an analysis of how equilibrium valuation price and divergence, and slippage losses can be minimized. They also formally defined these costs and demonstrated that while they could be moved around, they could never be eliminated.

Using input from a market price oracle to change the mathematical relationship between the assets, Krishnamachari et al. (2021) proposed a new method to build an AMM with the idea of dynamic curves. As a result, the pool price continuously and automatically adjusts to be the same as the market price.

2.2. Deep Reinforcement Learning on MM

Most applications of Reinforcement Learning and Deep Reinforcement Learning algorithms for market-making optimization are on CEXs. These studies try to get optimal bidding and pricing strategies in limit-order books.

Sadighian (2020) developed a policy-based model-free Reinforcement Learning algorithm trained from order book data using an event-based environment, where an event is defined as a change in price greater or less than a given threshold, as opposed to by tick or time-based events. Bakshaev (2020) applied a continuous action space soft actor-critic Reinforcement Learning model using Open AI. Spooner (2020)

developed an Adversarial Reinforcement Learning to produce market-making agents that are robust to adversarial and adaptively-chosen market conditions. These cases are not helpful when applying them in AMM since they operate algorithmically in a deterministic way.

2.3. Deep Reinforcement Learning on AMM

The research on this specific area is relatively sparse. Most studies apply Reinforcement Learning for automatic trading from an investment management perspective. For example, Lucarelli & Borrotti (2019) compared Double and Dueling DQN to trade Bitcoin, using the Sharpe ratio and profit as reward functions.

Lim (2022) utilized a deep hybrid Long Short-Term Memory (LSTM) and a Double Deep Q-Learning algorithm to reduce divergence loss for liquidity providers, reduce slippage for liquidity takers and improve market efficiency through better forecasts liquidity concentration ranges. In a different-scoped study, Churiwala and Krishnamachari (2022) proposed a Q-Learning model to optimize the fees collected by the liquidity providers on an AMM protocol, varying fee rates, and leverage coefficients with different market conditions.

However, to date, no model has comprehensively generically addressed the problem. Lim proposed a solution to reduce slippage and impermanent loss in the Uniswap v3 protocol, which uses CPMM. This paper builds upon Lim's work and extends it to the case of a Hybrid Flexible Market Maker (HFMM). Specifically, the proposed approach is an alternative Double Deep Q-Learning model that adjusts the leverage coefficient C.

3. Empirical Development

3.1. Background on AMM

3.1.1. Conservation Function

The bonding curve is defined by a conservation function, which is the mathematical expression determining the asset price in an AMM. This function follows an invariant property (Churiwala & Krishnamachari, 2022).

Constant Sum Market Maker

The Constant Sum Market Maker (CSMM) is a simple exchange function not commonly utilized due to its notable drawbacks. The CSMM maintains the sum of token reserves, denoted by (x, y), as a constant value, which is expressed as this equation:

$$(x_t - \Delta_x) + (y_t + \Delta_y) = k$$

Where:

- x_t and y_t are the reserves of X and Y tokens at a fixed time t, respectively,
- Δ_x and Δ_y are the changes of X and Y tokens on a trade, respectively,
- k is a constant defined by the AMM.

Generalizing to multiple tokens:

$$\sum_{i=1}^{n} i_t = k$$

Where:

- n is the number of tokens in the pool.

The equation implies that the exchange rate between tokens is constant. In contrast, it is vulnerable to arbitrage opportunities: when there is an imbalance or volatility in the market and given that the exchange rate does not change, any arbitrageur would drain one of the reserves.

Constant Product Market Maker

Due to its robustness under different market conditions, the canonical conservation function is the Constant Product Market Maker (CPMM). It assumes that the product of the amounts of each token in the pool is constant. For token reserves (x, y), consider:

$$(x_t - \Delta_x).(x_t + \Delta_y) = k$$

Generalizing to multiple tokens:

$$\prod_{i=1}^{n} i_t = k$$

In this case, exchange rate changes at every state, which causes a series of issues such as slippage or impermanent loss but are always able to provide liquidity.

Hybrid Function Market Maker

The Hybrid Function Market Maker (HFMM) combines CSMM and CPMM. While the CSMM maintains a constant sum of reserves, meaning there is no price slippage, the CPMM provides liquidity even in limited markets. The HFMM is designed to achieve no price slippage, even in illiquid markets, and always provide liquidity. By merging the exchange functions of CSMM and CPMM, the HFMM curves lie between them, with the curvature controlled by a leverage coefficient C. This coefficient determines the curvature of the HFMM, which can be adjusted to lean towards either the CPMM or the CSMM curve.

This function is inspired mainly by the whitepaper for Curve Finance (Egorov, 2021). The protocol's underlying exchange function can dynamically shift from a constant-product invariant to a constant-sum invariant. This shift is accomplished by interpolating between the two invariants, controlled by the C leverage coefficient.

When $C \to 0$, the exchange behaviors as a CPMM, and when $C \to \infty$, it does it as a CSMM. From the first version of Curve Finance (Egorov, 2019), the invariant equation becomes:

$$C.n^n.\sum_{i=1}^{n} i_t + D = C.D.n^n + \frac{D^{n+1}}{n^n.\prod_{i=1}^{n} i_t}$$

Where:

- C is the leverage coefficient,
- n is the number of tokens available at the pool,
- i_t is the available amount of token i at a fixed time t,
- D the constant sum invariant defined by $\sum_{i=1}^{n} i_t$.

Figure 2 shows the three types of bonding curves.

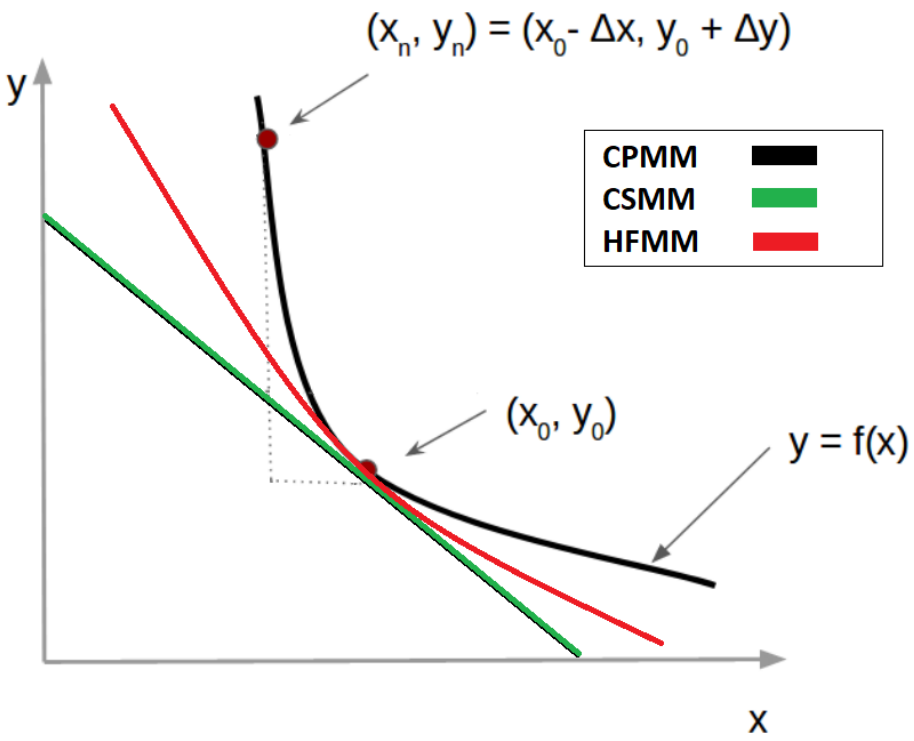

Figure 2. Types of bonding curves: Constant Product Market Maker, Constant Sum Market Maker, and Hybrid Function Market Maker (own figure).

3.1.2. Price and Bonding Curve

It is noted on every AMM function that every associated curve is convex and monotonically decreasing. This ensures that the price of the token x is monotonically decreasing as a function of its availability in the pool, as should be expected of a typical supply curve (Krishnamachari et al., 2021).

For token reserves (x, y) and given an AMM function, the price of token x, measured in terms of y, is:

$$p_x(x, y) = -\frac{dy}{dx}$$

For example, for the CPMM, it gets $p_x(x, y) = \frac{k}{x^2}$, and it is noted that the price of token x varies with its supply in the liquidity pool.

Then, the value of the pool is:

$$V_p(x, y) = p_x.x + y$$

3.1.3. Slippage

Slippage refers to the financial loss suffered by a trader due to a discrepancy between the pool price at which a trade is initiated and the actual price achieved during the trade. Given a trader who wants to buy Δ_x units of token X, then the liquidity pool goes from (x_0, y_0) to $(x_t, y_t) = (x_0 - \Delta_x, y_0 + \Delta_y)$. If the pool price at t=0 was p_0, then the buyer would have to pay $\Delta_{y_0} = p_0\Delta_x$. The difference between Δ_y and Δ_{y_0} is defined as the slippage loss $S(x_0, y_0, \Delta_x)$. Similarly, on a sell of X token, the slippage results: $S(x_0, y_0, -\Delta_x)$.

On a CSMM, the slippage loss is always 0 because the price is constant at all points on the curve. However, for any strictly convex curve, such as CPMM or HFMM, the slippage loss is always a positive quantity, given that the tangent line will be below the curve. This implies that traders will always incur a penalty. Moreover, as the size of the trade increases, the total slippage will also increase, which can serve as a disincentive for traders to conduct large trades with the liquidity provider.

Figure 3 presents a bonding curve from a CPMM, showing slippage on a sample trade.

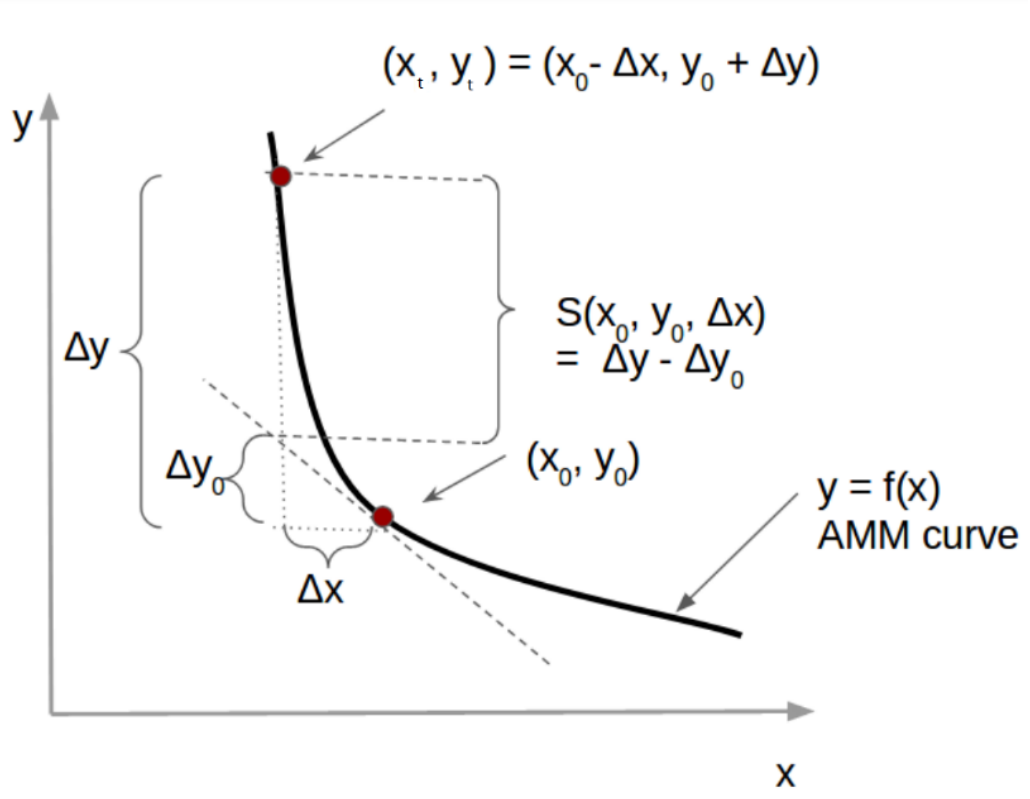

Figure 3. Bounding curve defined from a CPMM. It is shown the slippage on a trade (adopted from Krishnamachari et al., 2021).

3.1.4. Impermanent Loss

Generally, when a trade is made, the price changes and moves from (x_0, y_0) to (x_t, y_t), resulting in a new price. This causes the value of a liquidity pool could decrease after the trade. This decrease is defined as impermanent loss, as follows:

$$\delta = \frac{V_{p_t}(x_t, y_t) - V_{p_t}(x_0, y_0)}{V_{p_t}(x_0, y_0)}$$

For a CPMM, this results in the following:

$$\delta = \frac{2\sqrt{\rho} - 1 - \rho}{1 + \rho}$$

For a CSMM, as the price does not change, $\delta = 0$

3.1.5. Dynamic Curves

In the original CPMM, the curve has a fixed form and is determined by the initial total liquidity, suppose $x_0.y_0 = k$. The curve can only change if the liquidity provider add/remove tokens of the pool, but not from trading activity.

Suppose an AMM learns from an external Oracle service that its assets market valuation has moved from the current stable value, from (x_0, y_0) to (x_0', y_0'). This leaves the liquidity providers exposed to arbitrage trades and, in consequence, to substantial impermanent loss, moving (x_0, y_0) to (x_0', y_0') (Engel and Herlihy, 2021).

The AMM can eliminate this opportunity by moving the curve, as a pseudo arbitrage. Suppose $x_0 > x_0'$ and $y_0' > y_0$, the transformed AMM curve becomes $(x, f(x - (x_0 - x_0') - (y_0' - y_0))$. The current state (x_0, y_0) still lies on the new curve, but now with a different slope, e.g., price that matches the new valuation.

The main advantage of this is that the AMM is no longer exposed to impermanent loss from arbitrage trades. The disadvantage is that the equation does not satisfy the boundary conditions:

- $\lim_{x\to\infty} f(x) = 0$ and
- $\lim_{x\to 0} f(x) = \infty$.

This means now the AMM contains more units of token x and a shortage of token y to cover all possible. Although the imbalance is usually minor, given that small tick changes typically drive price movements in an efficient market, it can accumulate over time and become problematic. Therefore, the AMM will need to address this shortfall by making minor adjustments to liquidity provisions to rebalance the pool. As part of liquidity provision, liquidity providers must deposit additional tokens of either x or y, as the AMM states. Incentives will be provided for all tokens deposited, including the additional X or Y tokens. This result generalizes to the dynamic version of any strictly convex curve (Krishnamachari et al., 2021).

3.2. Background on Reinforcement Learning

3.2.1. Key Elements

Although there is a wide range of algorithms in reinforcement learning, they all have the same essential components (Ravichandiran, 2020; Dixon et al., 2020):

- **Agent**. An agent, also called a learner, is a software program that learns to make decisions in a settled environment. For instance, in a financial trading environment, the agent could be a trader making decisions on order executions.

- **Environment**. The environment is everything outside of the agent. In a more general sense, it is the universe except for the agent, but technically it is simplified and modeled. In the financial trading example, the environment could be the market, including orders, tokens, prices, and the rest of the traders.
- **State**. A state is a representation of the position in the modeled environment that the agent can be in. For example, a state of the environment could be the current orders, tokens, and prices at a specific moment. The agent, as it evolves, will go through different states of the environment. The environment is denoted by s.
- **Action**. The agent interacts with the environment and moves from one state to another by performing an action. For example, at every step, the trader places an order on the market with a buy or sell price. The order, with its details, is the action. It is denoted by a.
- **Reward**. The agent receives a reward based on the action executed by the agent and the new state of the environment. This is nothing but a numerical value based on the performance of the action. For the provided example, if the trader (agent) places an order (action) and gets a positive profit, the agent receives a positive reward. In contrast, if the agent loses money on the trade, it receives a penalization of a negative reward. Modeling a reward function is one of the main components that drive a reinforcement learning algorithm, affecting every decision the agent makes and can lead to excellent results or poor performance in a sensible way.

Figure 4 shows the interaction between the agent and the environment. Given a state, the agent performs and action on the environment, which outputs a new state and the corresponding reward.

3.2.2. Algorithm

The basic steps for a reinforcement learning algorithm are:

1. At the step t, the agent gets the state of the environment s_t and executes and action a_t
2. By performing an action, the environment moves to state s_{t+1}
3. Based on this new state, the agent receives a reward r_{t+1}
4. Given this reward, the agent will interpretate if the executed action was good or not and will learn from this with the purpose of receive positive rewards on the next steps.

There are a lot of considerations in each of these steps, which will be explained and detailed in the following sections.

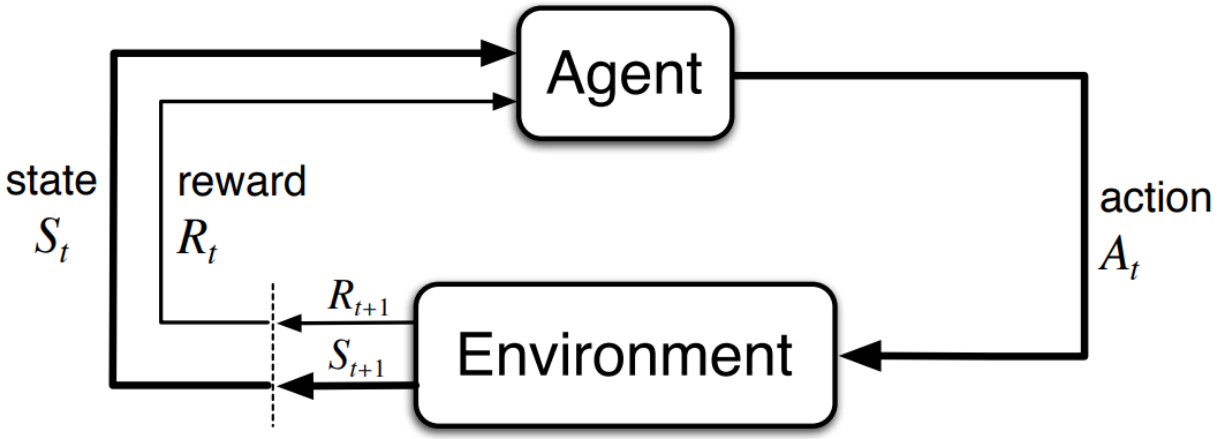

Figure 4. Agent-Environment interaction in a reinforcement learning model (adopted from Sutton and Burton, 2020)

3.2.3. Other Elements

Action Space

In each state, the agent can perform a determined range of actions on the environment. This set of actions is called action space. There are two types:

- Discrete action space
- Continuous action space

The first one consists of actions that are discrete. For the trading example, the action space could consist of a sell order and a buy order. On the other hand, continuous action space is suited when actions are continuous. For example, the amount of the placed order could be a continuous variable.

Policy

The policy is the core of the agent. It tells the agent which action to perform in each state. In its most basic form, the policy is a table that maps each state to an action. The table is initialized with random values, and over the series of iterations, it will be learned from the rewards, and values will be updated. The optimal policy is the policy that gets the agent the best reward. It can be classified as:

- Deterministic policy
- Stochastic policy

The first one maps each state to one particular action. It does not mean that it will not change over iterations, but it is, at each step, fully determined. Unlike a deterministic policy, a stochastic policy maps the state to a probability distribution over an action space.

Episode

The model evolves over the advancement of the states of the environment. The agent-environment interaction starting from one state until a final state is called an episode, and the agent will act over episodic tasks. This is the generic case in games, where the agent dies and is not able to continue executing new actions. Sometimes, the model cannot have a final state so that the agent will act over a continuous task. This case can be applied to the example of the trader, given that it acts on the market, which does not have an ending state.

3.2.4. Markov Decision Process

The foundation of reinforcement learning is the Markov Decision Process (MDP). It provides a mathematical model used to generate decision-making problems in situations where the outcome of an action is uncertain. In its discrete form, an MDP consists of a set of states and a set of actions that can be taken in each state, a reward function that determines the immediate reward obtained for each action, and a transition function that specifies the probability of moving from one state to another after taking an action.

The MDP framework assumes that the future depends only on the current state and not any previous states or actions. Reinforcement learning uses MDP as a framework to model the decision-making problem, and the agent's goal is to learn the optimal policy by maximizing the expected cumulative reward (Rao & Jelvis, 2022).

To formalize this process, the notation provided by Liu (2020) is introduced:

- Event: t
- State: s_t
- Action: a_t
- Next state from state s_t following action a_t: s_{t+1}, with probability:

$$P(s_{t+1}|s_t, a_t)$$

- Reward given to the agent while it transits to s_{t+1}: r_t, with probability:

$$P(s_{t+1}, r_t|s_t, a_t)$$

- Policy of the agent, which is probability of taking action a_t from state s_t: $\pi(s_t, a_t)$
- The objective is to maximize cumulative discounted return:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k . r_{t+k+1}$$

including the possibility of $T \to \infty$ or $\gamma = 1$ but not both. γ is the discounted factor and is applied so the agent can learn to sacrifice the short-term reward for a bigger long-term benefit.

The analysis reveals the presence of multiple integral components, including the agent, the environment, the reward signal, and the policy or value function. The agent is the entity that interacts with the environment and makes decisions based on the current state. The environment is the entity that the agent interacts with and provides feedback in the form of a reward signal. The reward signal is a scalar value that indicates the quality of the agent's action in the current state. The policy or value function is the function that the agent learns to make decisions based on the current state.

The probability for each possible pair of (r, s') to transition from state s after action a:

$$p(s', r|s, a) = P(s_{t+1} = s', R_{t+1} = r|s_t = s, a_t = a)$$

Then, the policy is defined:

$$\pi(a|s) = P(a_t = a|s_t = s)$$

Given a policy, the value of state or value function is defined as:

$$V_\pi(s) = E_\pi[G_t|s_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k . r_{t+k+1}|s_t = s]$$

Notice that this value of state is dependent on the policy. The value function denotes the expected return starting from that state and following the policy π.

Similarly, for a state-action pair, the Q value of a state-action pair or Q function is defined as:

$$Q_\pi(s, a) = E_\pi[G_t|s_t = s, a_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k . r_{t+k+1}|s_t = s, a_t = a]$$

In this case, the Q function represents the expected return starting from a given state-action pair and following the policy π.

3.2.5. The Bellman Equation

The Bellman equation simplifies the state value or state-action value calculation.

The value function has a recursive property (Liu, 2020):

$$V_\pi(s) = E_\pi[\sum_{k=0}^{\infty} \gamma^k . r_{t+k+1} | S_t = s]$$

$$= E_\pi[R_{t+1} + \sum_{k=0}^{\infty} \gamma^k . r_{t+k+2} | S_t = s]$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma . V_\pi(s')]$$

This equation presents a recursive method to compute the value function at time t in terms of its future values at time $t + 1$ by going backward in time. This equation type is known as Bellman Equation. It was proposed in 1950s by Richard Bellman in the context of his pioneering work on dynamic programming (Bellman, 1952).

Similarly, the Bellman equation of the Q-function is:

$$Q_\pi(s, a) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma . Q_\pi(s', a')]$$

3.2.6. The Optimal Bellman Equation

Is it known that the value function depends on the policy, i.e., the value of a given state varies based on the chosen policy? Thus, there can be many different value functions according to different policies. The optimal value function denoted as $V^*(s)$ is the one that yields the maximum value compared to all other value functions. In a more general sense, the optimal Bellman equation is the one that has the maximum value (Ravichandiran, 2020).

The problem is that it needs to be known which action gives the maximum value. The solution is, instead of using a policy to select the action, compute the value of the state using all possible actions and then select the maximum value as the value of the state.

Given this, the expectation over the policy can be removed from the equation, and add the max over the action to express the optimal Bellman equation:

$$V^*(s) = max_a \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma . V^*(s')]$$

Similarly, for the Q function, instead of using the policy π to select action a' in the next state s', it is given by choosing all possible actions in that state s' and computing the maximum Q value:

$$Q^*(s, a) = \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma . max_{a'} Q^*(s', a')]$$

The relationship with these two identities is given as follows:

$$V^*(s) = max_a Q^*(s, a)$$

Using the Bellman equations:

$$Q^*(s, a) = \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma . V^*(s')]$$

If this relation is reversed, starting from a value function $V(s)$, or a state-action function $q(s, a)$, an optimal policy is obtained:

$$\pi_V(s) = argmax_a \sum_{s'} p(s'|s, a) . V(s')$$

$$\pi_Q(s) = argmax_a Q(s, a)$$

3.2.7. Policy-based vs Value-based models

Given the previous equations, it is noted that there are two approaches to learn the policy:

- Policy-based methods: where the policy function is directly trained, using the p-function ($\pi_V(s)$). It is also called P-Learning.
- Value-based methods: where the policy function is learned indirectly. First, the agent learns which state is more valuable, and then take the action that leads to the more valuable states. It is also called Q-Learning.

In policy-based reinforcement learning, the agent learns a policy function that maps states to actions, e.g., selecting what action to take given a state (or a probability distribution over actions at that state). The policy function can be represented by a neural network or a lookup table. The policy can be deterministic, meaning that each output corresponds to one action given a state. It can also be a stochastic policy that outputs a probability distribution over actions. Consequently, it is not defined by hand the behavior of the policy, and so is the training that will define it. The most common policy-based algorithm is the policy gradient algorithm, which learns the optimal policy by updating the parameters of the policy function in the direction of the gradient of the expected cumulative reward with respect to the parameters.

On the other hand, in value-based reinforcement learning, the agent learns a value function that estimates the expected long-term reward for taking a particular action in a particular state. The most common value-based algorithm is Q-learning, which uses a table or function to store the Q-values for each state-action pair. The Q-value represents the expected cumulative reward of taking action in a particular state and then following the optimal policy from that point on.

This means that a value function is trained, which outputs the value of a state-action pair. Given this, the policy then takes action. Since the policy is not trained, its behavior must be specified. The most common policy is the Greedy Policy, which, given the value function, will take actions that always lead to the biggest reward.

3.2.8. The exploration-exploitation trade-off

In order to maximize its cumulative reward, a reinforcement learning agent must balance its decisions between exploiting its past experiences and exploring new options. Exploitation allows the agent to make use of its existing knowledge to make decisions, while exploration presents opportunities to discover new, potentially better choices. However, excessive exploration can be time-consuming and computationally expensive, while pure exploitation can result in suboptimal solutions. Therefore, finding the appropriate balance between the two is critical for designing successful RL algorithms (Liu, 2020).

The exploration-exploitation dilemma is a common challenge in many decision-making processes that involve a feedback loop between data gathering and decision-making. An efficient approach to this problem is essential for optimizing performance.

The most basic method is called $\varepsilon - greedy$ action selection. In this method, whenever action is selected, there is a probability of ε that this selection is entirely random, regardless of what the current policy model outputs. The optimal action based on the current learning is selected with only the probability of $1 - \varepsilon$. This parameter of ε is usually decreased over time, starting at one at the beginning of the training, such that the agent can fully explore different actions. As the slowly decreases to 5% or even to 0, the agent can explore some variations based on the current policy.

3.2.9. Monte Carlo vs Temporal Difference Learning

Monte Carlo and Temporal Difference Learning (TD) are two different strategies for training the value or policy functions. Both methods use the experience to train the algorithms (Rao, 2022).

Basically, Monte Carlo uses an entire episode of experience before learning. On the other hand, Temporal Difference uses only a step to learn.

Following the Monte Carlo approach, given a policy π, generate a number of full episodes. For each state-action pair in these generated episodes, update the values according to discounted rewards from the full episode. This approach can be particularly valuable in situations where the understanding of the environment's dynamics is limited (Liu, 2020).

$$V(S_t) = V(S_t) + \alpha.[G_t - V(S_t)]$$

Where G_t is the discounted return from an episode.

This method carries a limitation: if the length of the episode is large, this could result in a significant variance.

Alternative, the Temporal Difference method update the $V(S_t)$ at each step:

$$V(S_t) = V(S_t) + \alpha.\delta$$

Where δ is the TD error, defined as:

$$\delta = R_{t+1} + \gamma.V(S_{t+1}) - V(S_t)$$

While in Monte Carlo the update factor is G_t, in Temporal Difference it is estimated using $R_{t+1} + \gamma.V(S_{t+1})$. According to Liu (2020), studies show that Temporal Difference improves performance compared to Monte Carlo.

3.2.10. Types of Reinforcement Learning algorithms

Another category for reinforcement learning algorithms is based on whether the agent has access to a model of the environment, which is a function that predicts state transitions and rewards (Kanwar, 2019).

Model-Based algorithms

Model-based algorithms enable the agent to engage in proactive planning by forecasting potential outcomes for various choices and consciously selecting the best option through this process. Afterward, the agent can extract insights from this proactive planning and incorporate them into a learned policy (Kanwar, 2019).

However, one major drawback is that the agent usually lacks access to an accurate model of the environment. If the agent seeks to use a model in such a scenario, it must derive it solely from experience, which poses various challenges. The most significant obstacle is that the agent may exploit biases in the model, resulting in an agent that performs well based on the learned model but exhibits suboptimal behavior in the real environment.

Model-Free algorithms

Reinforcement learning methods that are model-free operate without using an environment model (Kanwar, 2019). They do not attempt to learn the fundamental principles that regulate an agent's interaction with its environment. In contrast to model-based approaches, model-free methods prioritize being simpler to implement and tune. Model-free algorithms, such as policy iteration or value iteration, directly calculate the optimal policy or value function. This is substantially more computationally efficient. As a result, model-free algorithms have garnered considerably more attention than their model-based counterparts. As it was explained, there are two main approaches to representing and training agents with model-free algorithms:

- Policy-based methods, or policy optimization methods.
- Value-based methods, or Q-Learning.

Off-Policy vs On-Policy models

Another categorization of reinforcement learning algorithms relies upon terms of the policy. Off-policy methods use a different policy for acting (inference) and updating (training), while on-policy models use the same one for acting and updating (Kanwar, 2019).

Overall, the choice between on-policy and off-policy algorithms depends on the specific problem and the available resources. On-policy algorithms are preferred in situations where the policy changes frequently or the data collection process is expensive, and stability is a major concern. Off-policy algorithms are preferred when a significant amount of historical data is available, or the acting policy can be designed to cover the state-action space efficiently (Lapan, 2020).

3.2.11. The Q-Learning Algorithm

Q-Learning is an off-policy, value-based method that uses a Temporal Difference approach to train its action-value function. It uses an $\varepsilon - greedy$ policy for acting, while a greedy policy for updating values.

The process is as follows:

1. Initialize an empty table of $Q(s, a)$ for all pairs of (s, a)
2. Interact with the environment and obtain the state (s, a, r, s'). It is recommended to apply a exploration-exploitation technique in this step.
3. Update the Q value based on the Bellman equation:
4. $Q^*(s, a) = \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma . max_{a'} Q^*(s', a')]$
5. Check convergence condition and repeat from step 2.

It is noted that the optimal Q function is used, which may not be optimal in the firsts iterations as Q values are not known, and the table is initialized with zero values.

3.2.12. The Deep Q-Network (DQN)

Deep neural networks can also be applied to solve reinforcement learning algorithms. Deep Q-Learning or Deep Q-Networks (DQNs) was introduced by Mnih et al. (2015).

The tabular Q-Learning algorithm iterates over a complete set of states, while this becomes computationally expensive when the state space becomes more extensive or even continuous. The idea of Deep Q-Learning is to build a neural network $Q_\theta(s, a)$ to approximate $Q(s, a) \approx Q(s, a, \theta)$ (Liu, 2020). The simplest form of this algorithm is:

1. Initialize a neural network $Q_\theta(s, a)$ with parameter θ as random to approximate $Q(s, a)$. The dimension of the input and output of the network should be the same as the dimension of state space and size of action space.
2. Interact with the environment and obtain the state (s, a, r, s').
3. Calculate the loss given the Temporal Difference method.
4. Update $Q_\theta(s, a)$ paremeter θ using stochastic gradient descent algorithm.
5. Repeat from step 2 until converged.

Replay memory

However, this algorithm is not highly effective. If it selects actions randomly, the chances of hitting and earning rewards are quite low. Thus, the program learns very slowly. To address this issue, it is common to utilize a replay memory to collect the agent's experience and, based on that, train the network.

The replay memory or replay memory, denoted as D, saves the agent's previous experience, which is every tuple (s, a, r, s'). The replay memory stores these transitions in a memory buffer and samples them randomly to update the Q-values. This allows the network to learn from past experiences and reduce the correlation between the samples used for updates, which can improve the learning speed and stability of the algorithm.

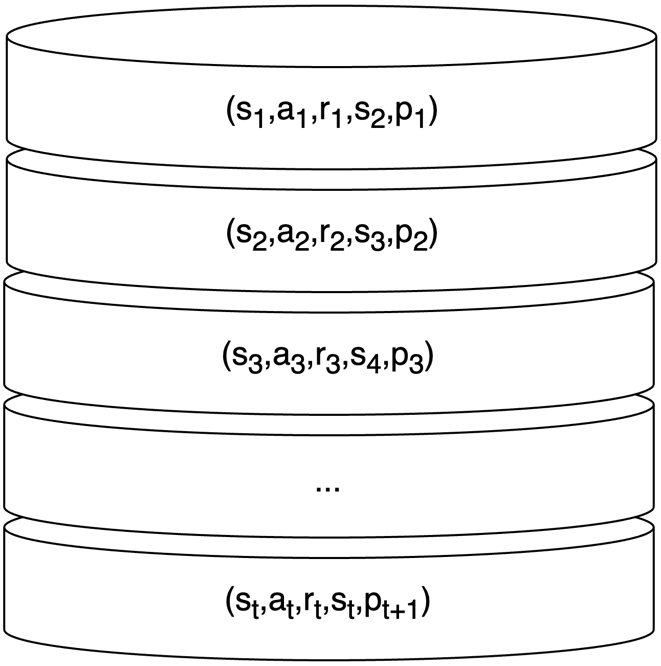

Figure 5. Replay memory schema. It contains all the previous experience of the agent (adopted from Ravichandiran, 2021).

This works as follows:

1. Initialize the replay memory D
2. For each step in the episode:
    a. Make a transition, that is, perform an action a in the state s, move to the next state s' and receive a reward r.
    b. Store the transition (s, a, r, s') in the replay memory D.

A schema of a replay memory is shown in Figure 5. For the algorithm to have stable behavior, the replay memory should be large enough to contain a wide range of experiences, but it may only sometimes be good to keep everything. If it only uses the very-most recent data, you will be overfitted to that. In contrast, using too much experience may slow down learning (Kanwar, 2019).

Loss function

The loss function that is generally applied is the Mean Squared Error (MSE) (Ravichandiran, 2021):

$$MSE = \frac{1}{K}\sum_{i=1}^{K}(y_i - \hat{y}_i)^2$$

Where:

- y is the target value,
- $\hat{y}$ is the predicted value,
- K is the number of training samples.

Using the optimal Q value as the target value and the predicted Q value as the predicted value, then the loss function L is defined:

$$L(\theta) = Q^*(s,a) - Q_\theta(s,a)$$

$$L(\theta) = r + \gamma.max_{a'}Q(s',a') - Q_\theta(s,a)$$

The Q value of the next state $Q(s',a')$ can be calculated using the same DQN Q_θ:

$$L(\theta) = r + \gamma.max_{a'}Q_\theta(s',a') - Q_\theta(s,a)$$

On every iteration, it is randomly sampled a minibatch of K transitions (s,a,r,s') from the replay memory D so that:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma.max_{a'}Q_\theta(s_i',a') - Q_\theta(s_i,a))^2$$

The network is trained by minimizing the loss function. Then, the parameter is trained using gradient descent:

$$\theta = \theta - \alpha\nabla_\theta L(\theta)$$

Where α is the learning rate and is an hyperparameter of the model.

Target network

There is an issue when calculating the loss function. As noted, both the predicted and target values using the network Q_θ. This causes instability in the function, and the network learns poorly. It also drives divergence during training, given that the target value is not a source of truth and changes on every iteration (Ravichandiran, 2021).

To prevent this, an accepted solution is to use two separate neural networks: one is used to select the best action, and the other is used to estimate its Q-value. The first one, Q_θ, known as the **predictive network**, is updated at each iteration of the algorithm to approximate the optimal policy better. The second network, $Q_{\theta'}$, parametrized by parameter θ' and known as the **target network**, is used to compute the Q-value of the selected action, but its weights are frozen for a fixed number of iterations, and its parameters are copied from the original network once every fixed number of steps F, usually 100.

Then, the loss function is:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma.max_{a'}Q_{\theta'}(s_i', a') - Q_\theta(s_i, a))^2$$

3.2.13. The Double Deep Q-Network (DDQN)

One of the issues of the DQN is the overestimation of the Q value of the next state-action in the target due to the max operator in the equation (Mosavi et al., 2020):

$$y = r + \gamma.max_{a'}Q_{\theta'}(s', a')$$

The optimization procedure is imprecise because the Q values are not determined and are estimated by stochastic samples. Then, the network tends to propagate errors from selecting the maximum Q-value to estimating its value. In the end, it leads to poor policy performance. To address this issue, Hasselt et al. (2016) proposed Double Deep Q-Networks (DDQNs) by just modifying the target value computation:

$$y = r + \gamma.Q_{\theta'}(s', argmax_{a'}Q_\theta(s', a'))$$

Now, the two Q functions are present in the target value equation. The predictive network Q_θ is used for action selection, and the target network $Q_{\theta'}$ is used for Q value computation.

The algorithm

Finally, the algorithm for the Double Deep Q-Network is given as follows (Ravichandiran, 2021):

1. Initialize the main neural network $Q_\theta(s, a)$ with parameter θ with random values.
2. Initialize the target neural network $Q_{\theta'}(s, a)$ with parameter $\theta' = \theta$.
3. Initialize the replay memory D.
4. For N number of episodes and for T number of steps in each episode:
    a. Get state s
    b. Select action a using the epsilon-greedy policy, which selects:
        i. A random action with probability ε
        ii. Action $a = argmax_a Q_\theta(s, a)$
    c. Execute the action and move to next state s'
    d. Obtain the reward r
    e. Store the transition (s, a, r, s') in D
    f. Randomly sample a minibatch of K transitions from D
    g. Calculate the target values:
    $$y_i = r_i + \gamma . Q_{\theta'}({s_i}', argmax_a, Q_\theta({s_i}', a'))$$
    h. Calculate the loss:
    $$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a))^2$$
    i. Update the parameter θ:
    $$\theta = \theta - \alpha \nabla_\theta L(\theta)$$
    j. Freeze the target network parameter θ' for a fixed subset of steps F , and then update it by copying the values from θ
5. Interact with the environment and obtain the state (s, a, r, s').
6. Calculate the loss given the Temporal Difference method:
$$V(S_t) = V(S_t) + \alpha . \delta$$
7. Update $Q_\theta(s, a)$ paremeter θ using stochastic gradient descent algorithm.
8. Repeat from step 2 until converged.

Figure 6 presents a generic diagram for the DDQN model.

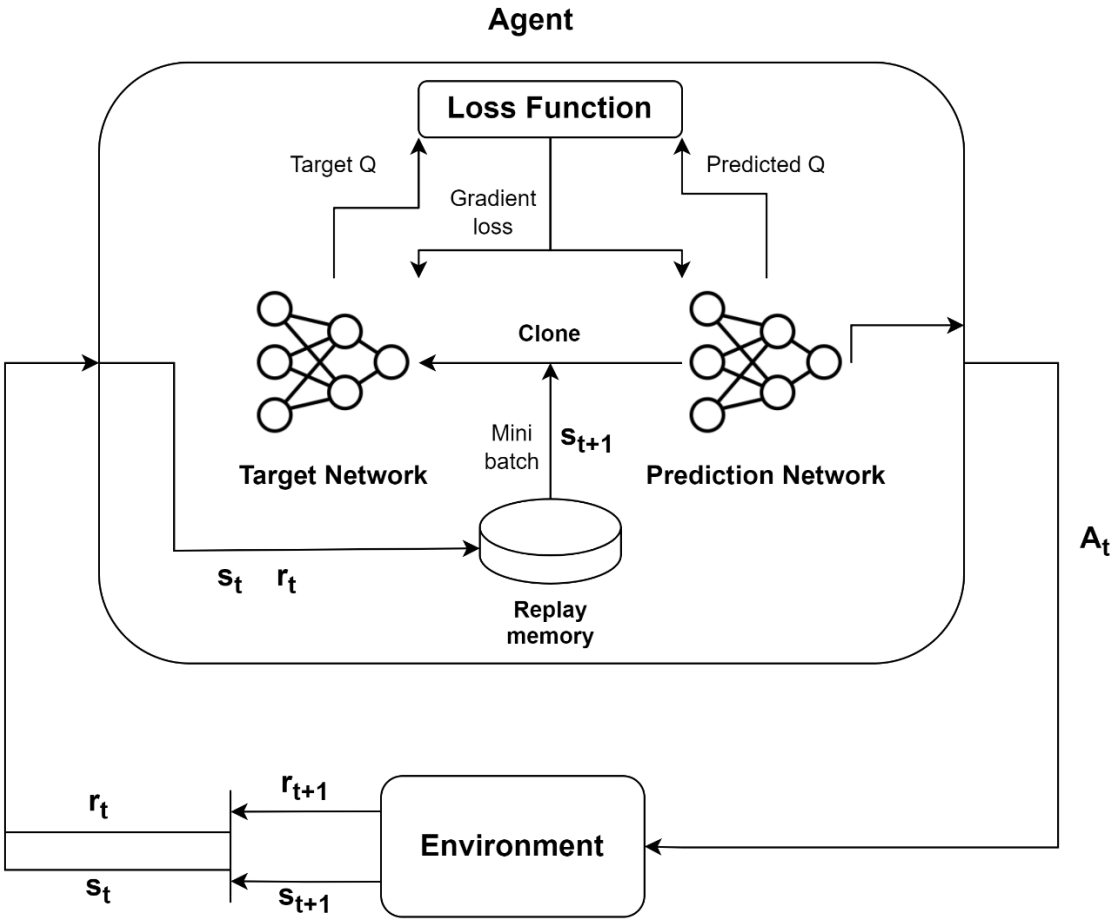

Figure 6. Double Deep Q-Network model (own figure).

DDQN has been shown to outperform Q-Learning in many tasks, particularly in tasks with high-dimensional state spaces and complex action spaces. However, DDQN is still subject to some limitations, such as the need for a large amount of training data and difficulty tuning the hyperparameters. Nevertheless, DDQN remains a promising approach for addressing the challenges of reinforcement learning in complex environments (Hasselt et al., 2015).

3.2.14. The DQN with prioritized experience replay

The conventional DQN model randomly samples a minibatch of K transitions from the replay memory D. This technique can be improved by assigning some priority to each transition in the replay memory and sampling the transitions with the best priority values for learning (Ravichandiran, 2021).

Retaking the Temporal Difference Learning error:

$$\delta = r + \gamma . max_{a'} Q_{\theta'}(s', a') - Q_{\theta}(s, a)$$

It is noted that a transition with a higher δ implies that the transition is not good enough. Thus, learning more about that transition is more convenient for minimizing the error than those with a lower error.

Given that, for every transition, the tuple (s, a, r, s', p), where p is the priority, is added to the replay memory D.

One of the famous methods to prioritize transitions is **proportional prioritization**. With this technique, the priority p, to select the transition i, measured in terms of probability, is:

$$P(i) = \frac{p_i^{\mu}}{\sum_k p_k}$$

Where:

- p_i is the priority $p_i = |\delta_i| + \tau$,
- τ is a parameter to considerate transitions with null errors, $\tau \ll 1$,
- μ is a parameter to considerate random samples. For $\mu \approx 1$, the buffer samples only transitions with high priority, and for $\mu \approx 0$, it samples random transitions.

One of the problems of this technique is that the model learns significantly from the samples with high priority. This may lead to the issue of overfitting, biasing the model to those transitions with high errors. This can be solved by adding the importance weight w, which tends to reduce the weights of transitions that have occurred many times:

$$w_i = \left(\frac{1}{N . P(i)}\right)^{\varphi}$$

Where:

- N is the length of the replay memory D,
- φ is a parameter to control the importance weight.

This weight is added when calculating the loss for the temporal difference method:

$$V(S_t) = V(S_t) + \alpha . w . \delta$$

4. Solution proposal

Some measures have been proposed based on related work to address the issues of impermanent loss and slippage in an AMM. On the one hand, the use of dynamic curves aims to mitigate the issue of impermanent loss resulting from arbitrage trades. On the other hand, the application of a Deep Q-Learning reinforcement learning model and its variants can adjust the leverage coefficient of the price curve, which can help to reduce both slippage and any remaining impermanent loss. Even though the exposed solution comes after a series of iterations on each variable, there is room for improvement in the model performance.

4.1. Application of dynamic curves and changes on liquidity

This work takes the study's conclusions from Engel and Herlihy (2021) and Krishnamachari et al. (2021), proposing the construction of an AMM using dynamic curves to eliminate arbitrage opportunities and minimize impermanent loss for the liquidity providers while maintaining liquidity and total value over a wide range of market prices. This implementation is called pseudo-arbitrage, which objective is to implement a dynamic curve using an external market oracle to match the pool pricing to the market pricing continuously, modifying the mathematical relationship between the assets (conservation function) so that the pool price continuously and automatically adjusts to be identical to the market price.

This fact can also be applied to an HFMM since it depends on CPMM. Now, with this dynamic implementation in the AMM, the traders still experience slippage, which is not reduced. However, this loss for the traders is entirely gained by the liquidity providers, which can be considered a slippage gain for them. Furthermore, since there is no change in the market price, the pool price remains unchanged in the dynamic CPMM, leading to no impermanent loss. In consequence, each trade generates a net income for the LP, resulting in a clear benefit from the LP's perspective.

4.2. Application of Reinforcement Learning

4.2.1. Description of the Events

In finance, reinforcement learning is often used to model financial time series, with the agent typically taking a step through the environment at regular time intervals (time-based). Depending on the trading strategy, the interval of time can be anywhere from seconds to days. However, for market making, tick events involved by buy and sell orders are used to trigger the agent to interact with its environment. This tick-based approach differs from a time-based one as events occur irregularly and more frequently. High frequency could be a problem, particularly concerning external factors such as noisy data and latency. To address this challenge, a price-based event approach is proposed.

In this study, the implementation suggested by Sadighian (2020) and followed by Lim (2022) is taken. The price-based approach defines an event as a change in the price, which in this case is the equilibrium valuation, using a sensitive threshold β_p. The agent then acts on events as they occur. The action space is

based on a typical market-making strategy where the agent cannot exit the market and is restricted to executing a single order. The events imply that actions are not regularly spaced in time.

This is, if there is a change in the equilibrium valuation (a change in the AMM price with respect to the market price provided by the oracle), and if this is greater than or less than β_p, the event is raised. β_p is the threshold and will be an hyperparameter of the model. These price change events are not regularly spaced in time, and β_p can be modified to reduce the time required to train the agent per episode, i.e. to execute a trading action.

It is important to note that in the present work, the index t is used to denote not the time step, but the event step.

The algorithm to create events, based on market prices and AMM deviation prices, is:

1. Get valuation v_t for every time $t = 0$
2. Set β_p threshold
3. Set $upper\_limit = v_t * (1 + \beta_p)$
4. Set $lower\_limit = v_t * (1 - \beta_p)$
5. For every $\boldsymbol{t}$, evaluate if $upper\_limit < v_t < lower\_limit$
6. If the valuation accomplishes the condition, register it as an event.
7. If not, ignore it.

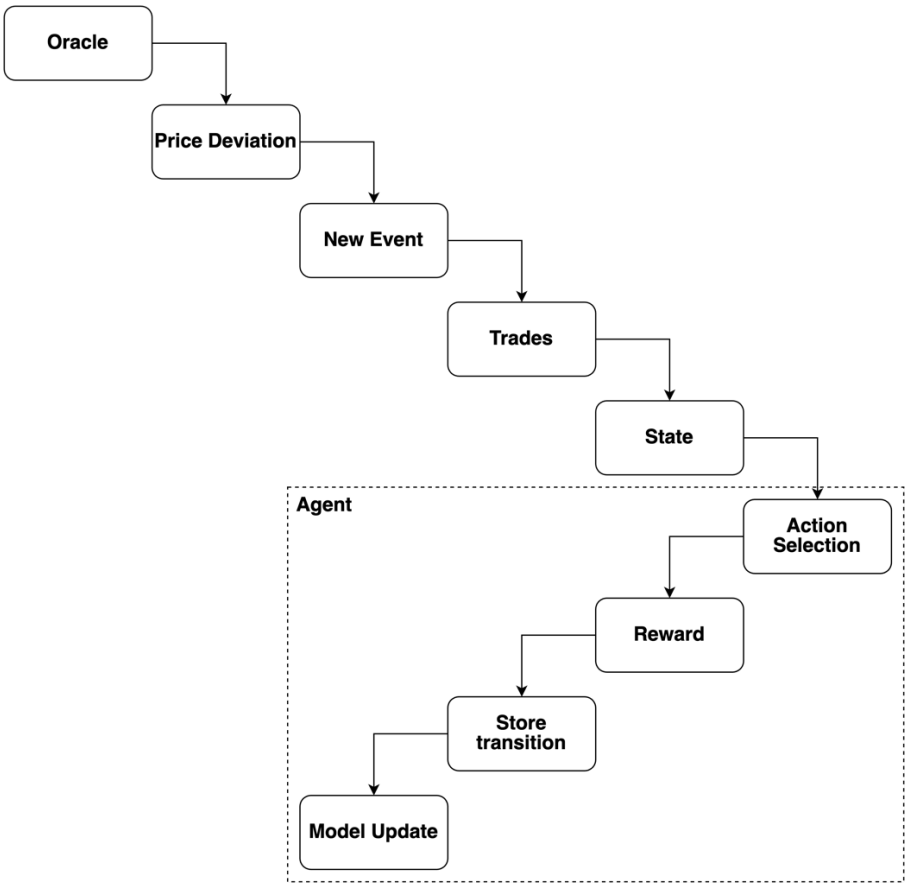

Figure 7. The procedure of the algorithm from Oracle price to model iteration (own figure).

Figure 7 shows an overview of the procedure. Given oracle prices, an event is raised for any significative price deviation. This event is denominated with t and represents the step in the model algorithm. Then, several trades are simulated from this event price, conforming to the state observation.

This state is the agent's input, which first selects the action according to its Q function. It uses the epsilon-greedy policy, so there is an epsilon probability of selecting a random action, which decays with advancing steps. The reward function is calculated with the action and the new state generated, and the new transition is stored. Finally, the model is updated according to the algorithm previously exposed.

4.2.2. Description of the Reward Function

Since the study aims to reduce divergence loss to liquidity providers and slippage loss to liquidity takers, the reward function needs to consider both factors.

First, the load function, provided by Engel and Herlihy (2021), is defined. Suppose that there is a change in the AMM price from $v_t = f(x_t, y_t)$ to $v_t{}' = f(x_t{}', y_t{}')$, which define the prices of tokens in terms of X. Then the load is:

$$load(v_t, v_t{}') = slipLoss(v_t, v_t{}') \,.\, divLoss(v_t, v_t{}')$$

Where:

- $slipLoss(v_t, v_t{}')$ is the slippage loss for an HFMM from v_t to $v_t{}'$,
- $slipLoss(v_t, v_t{}')$ is the impermanent loss for an HFMM from v_t to $v_t{}'$.

The main objective is to minimize this function, by reducing both slippage and impermanent loss.

The cumulative reward is:

$$R_t = \sum_{k=0}^{T-t-1} \gamma^k \,.\, r_{t+k+1}$$

Where γ is the discount rate, another hyperparameter of the model.

The reward function r is defined, at each step t:

$$r_{t+k} = \begin{cases} load, & if\ load\ \leq \beta_l \\ -2 * load, & if\ load > \beta_l \end{cases}$$

Where β_l is a hyperparameter to adjust the threshold within which load can be tolerated, determining the sensitivity of the reward function to the loss function. In this case, the proposal is to double penalize if the load exceeds the threshold.

4.2.3. Description of the Action Space

The agent controls the protocol´s leverage coefficient C with two possible actions:

$$A_t = \begin{cases} (1 - \beta_a).A_{t-1}, & if\ load\ \leq \beta_l \\ (1 + \beta_a).A_{t-1}, & if\ load\ > \beta_l \end{cases}$$

Where $\beta_a \in (0,1)$ is a hyperparameter to adjust the magnitud of the change of the leverage coefficient C from last action. This coefficient is also doubly bound:

- Lower limit: $C = 0$, representing the Uniswap coefficient.
- Upper limit: $C = 85$, representing the Curve Finance coefficient.

These limits were proposed by Churiwala and Krishnamachari (2022). They are not completely necessaries since $C \in (0, \infty)$. However, considering that a CSMM does not have enough value in a real-world application, it is better, for training purposes, to limit the range of this parameter.

4.2.4. Description of the State Space

At each decision epoch, t, provided by the event-driven algorithm, the state-space for the system can be characterized by:

$$S_t = (v_t, v_t', x_t, y_t, U, l_t, C_t)$$

Where:

- v_t: market valuation provided by a trusted external oracle. This is the AMM price value before the significant difference that raised the price event, measured in terms of X token.
- v_t': new AMM price, moved by the trades on the protocol, measured in terms of X token.
- (x_t, y_t): number of tokens of X and Y available in the pool,
- $U = \{U_1, U_2, \ldots, U_N\}$: set of traders interacting in the pool,
- l_t: summarized load for the event, given all the trades occurred on the period,
- $C_t = A_t$: leverage coefficient value which controls the bonding curve´s curvature.

4.3. Hypothesis and Assumptions

While the underlying process can be extrapolated for multiple tokens, the current study focuses on a dual-token context. This approach aligns from reality as many AMMs, such as Uniswap, operate with pools consisting of two tokens.

It is also noted that every exchange charges fees for each trade back to the asset pool, which in part is used to incentivize the liquidity providers. In this case, the effect of this fee is ignored, as they have minimal effect on costs. However, in most cases, fees can cause a minor decrease in the impermanent loss for liquidity providers and an increase in slippage cost for liquidity takers.

This work proposes a Q-Learning algorithm. This implies a model-free, off-policy algorithm. The environment is modeled as a Markov Decision Process (MDP) with discrete states and actions. Q-learning

is a model-free, off-policy reinforcement learning algorithm that can be used to learn the optimal action-selection policy for any given environment.

Particularly, the algorithm is composed of neural networks. In this case, additional assumptions, such as the function approximator being able to converge to the optimal Q-values and the data being stationary and independent, need to be satisfied.

4.4. Implementation Details

4.4.1. Simulation Setup

Events

The events were obtained based on the algorithm exposed in section 3.3.2. Given the oracle price, the events are raised from a significant deviation. The hyperparameter used was:

- Price threshold: $\beta_p = 0.01$

Users

The number of simulated users was $N = 10\ 000$. Each of them started with a set of tokens (x_k, y_k) with a total sum of 1000 tokens each.

The environment generates trades on each step and assigns them to random users. One crucial configuration is that if the balance of the token to sell is less than 0.2 of the other token, the swap is inverted. This mechanism prevents the drying up of either token's balance. This configuration is proposed by Churiwala and Krishnamachari (2022).

Price Data

Given that the AMM follows the trusted external Oracle price, a dataset from a Centralized Exchange was used. Notably, data was downloaded from Binance, for the ETH/USDT swap, at the hour level, from 2018-01-01 to 2022-12-12.

- Token X: USDT
- Token Y: ETH

Agent

Three models were trained:

- A simple DQN model,
- A Double DQN model,
- A Double DQN with prioritized experience replay model.

Following recommendations suggested by diverse authors, the models were trained with the following hyperparameters configuration.

For the neural network:

- For every learner, the network is composed of 2 CNN layers with 100 neurons each, followed by 2 fully connected layer streams with 50 neurons each (Lim, 2022),
- Batch Size: 50,
- Steps: 7000,
- Weight optimization algorithm: Gradient descent was tried, but Adam was finally selected,
- Activation function: ReLu,
- Learning rate: $\propto = 0.08$.

Figure 8 shows a general diagram of the used network.

This is a central part of the model, and due to its complexity, it can be designed in many different ways. Some authors use LSTM or RNN. It should be noted that there is much room for improvement and experimentation in this section of the algorithm.

For the rest of the agent's components:

- Discounted factor: $\gamma = 0.98$ (Lucarelli & Borrotti, 2019),
- Action threshold: $\beta_a = 0.02$, with $C \in [0, 85]$,
- Loss threshold: $\beta_l = 0.01$,
- F steps: $F = 100$,
- Exploration-exploitation strategy: $\varepsilon - greedy$.

For the DQN with prioritized experience replay:

- Null values prioritization $\tau = 0.001$,
- Random sample parameter: $\mu = 0.5$,
- Importance weight parameter: $\varphi = 0.5$,
- Length of the replay memory: $N = 1000$.

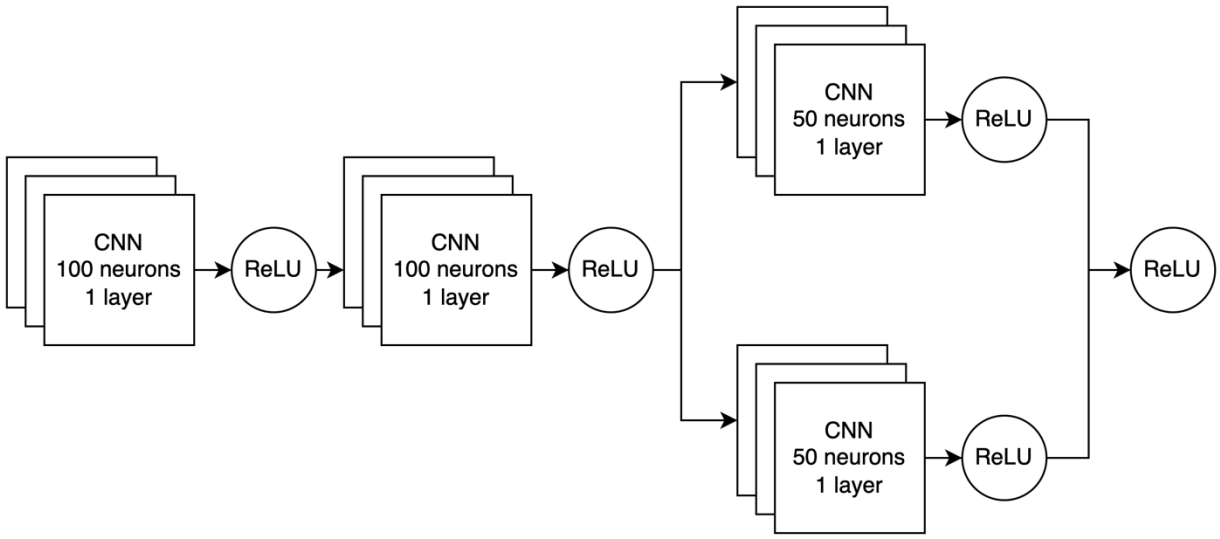

Figure 8. Neural network for the DQN, DDQN and DDQN with prioritized experience replay models.

4.4.2. Libraries and frameworks

To train the model, the following tools were used:

- Processor: Intel Core i7, 2.6 GHz
- RAM: 16 GB
- Programming language: python v3.11.2
- Neural network framework: pytorch v1.11.0

The reinforcement learning model was developed and trained with the Pytorch library. Tensorflow python library was also used, but it was finally selected considering that Pytorch is better when the CPU does the process.

5. Results

5.1. Deep Q-Network

Figure 9 exposes the total cumulative reward over iterations for the DQN model. In red are the rewards at every step, and in black is the 10-step window moving average. The unit measure of the total reward is in terms of the load.

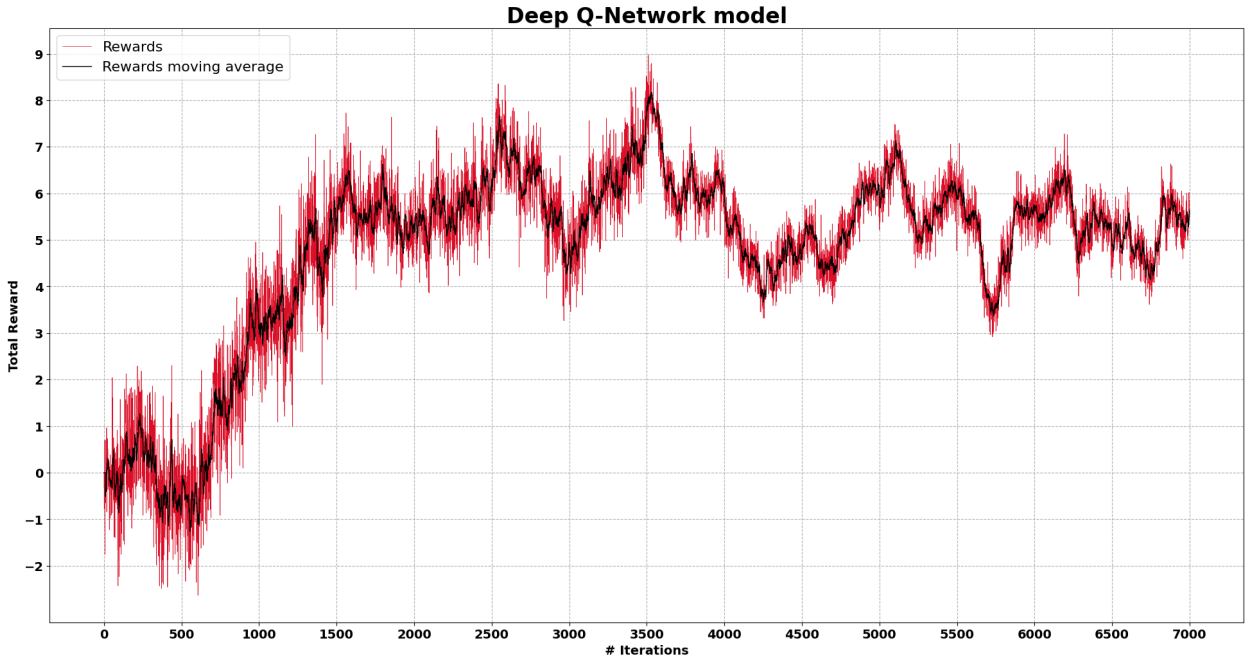

Figure 9. Performance of the basic Deep Q-Network model.

First, it is observed that the agent at the beginning has much variability, which indicates that it performs trial and error. This is consistent with the epsilon-greedy policy applied and that the network parameters were initialized randomly. Even though the variability seems to decay over iterations, the moving average still contains it.

As the iterations progress, the algorithm learns and accumulates a higher total reward. However, performance peaks and then does not improve. Something interesting to emphasize is that the algorithm follows trends. This may indicate that it is prioritizing experiences close in time.

5.2. Double Deep Q-Network

Figure 10 exposes the total cumulative reward over iterations for the DDQN model. In blue are the rewards at every step, and in black is the 10-step window moving average.

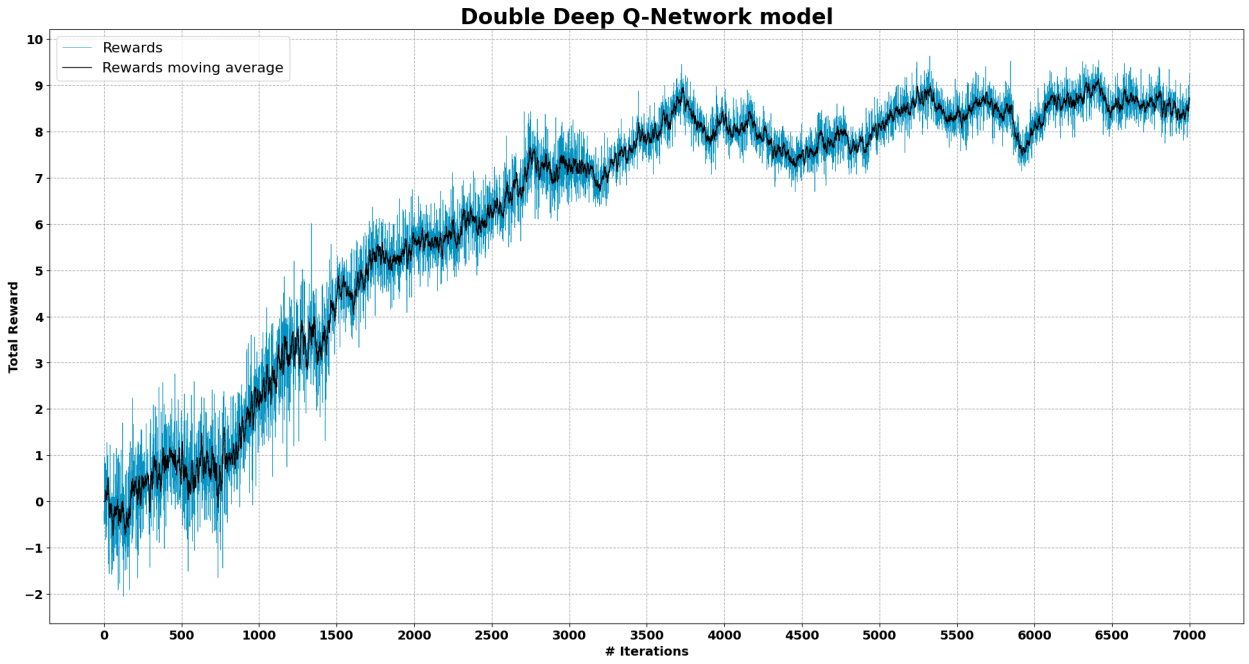

Figure 10. Performance of the Double Deep Q-Network model.

The image shows a significant improvement over the previous model. First, although high, the variability of the curve is much lower. On the other hand, the model converges to a value, unlike the DQN, where the model varies while it is already evolved.

Another feature to highlight is that the algorithm converges more slowly, despite the learning rate being the same. This is related to the fact that the algorithm does not overestimate the values, which is why it takes time to evolve.

Lastly, no noticeable tendencies of priority to the last events are observed, something that did happen in the previous model.

5.3. Double DQN with prioritized experience replay model

Figure 11 shows the total cumulative reward over iterations for the DDQN model. In green are the rewards at each iteration, and in black is the 10-step window moving average.

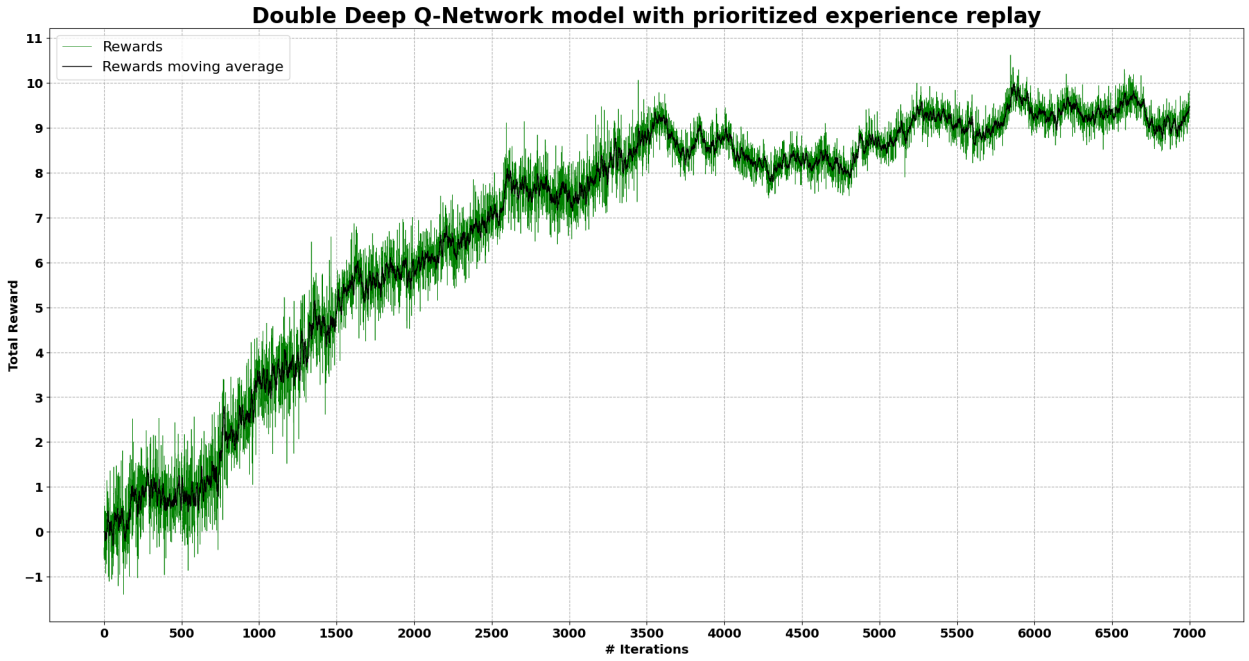

Figure 11. Performance of the Double Deep Q-Network with prioritized experience replay model.

In this case, it is seen that the accumulated total reward is higher in this case compared to the previous model. This may be because it learns faster based on large differences in transitions. However, no significant difference is observed in the other variables, such as variability.

Figure 12 shows the moving average of slippage and impermanent loss for the Double Deep Q-Network model, measured in terms of % of token X.

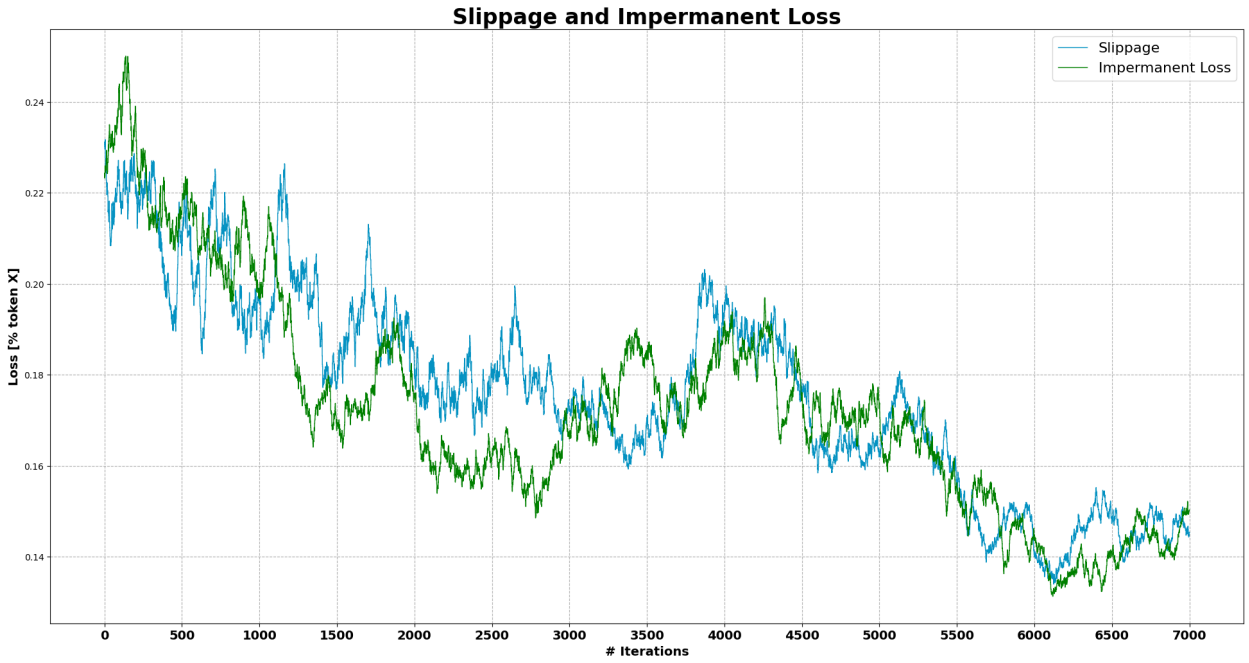

Figure 12. Slippage and Impermanent Loss, measured in terms of token X.

It is observed that both curves are correlated, and it makes sense to consider that, given a leverage coefficient C, before a trade and the price changes, the movement on the curve generates both slippage and impermanent loss.

In general, a decrease in both curves is observed with the advancement of the steps, generating a lower load for the model. However, the variability is significant. This can be explained by the same reason as in the reward curves of the different models. The variability in the choice of action for the agent, mainly in the first steps where the actions are mainly chosen randomly, generates this variation.

5.4. Hyperparameters Optimization

In this section, some of the primary hyperparameters and their implication for the model's performance are exposed. The model used to show the results is the Double Deep Q-Network with prioritized experience replay, which performs the best results in terms of performance.

It is important to note that the hyperparameter space is vast, and there is room for improvement in most of them. In this section, only some of them are detailed for reference, considering that those related to the reward function are the ones that tend to have the most significant importance.

Since the total reward has variability, it would be wrong to compare the performance of different models using only their maximum value. For this reason, an average of the last 2000 steps was used as a performance metric to compare the different models. Figure 13 shows the average reward of steps 5000 to 7000 over three hyperparameters.

In particular, Figure 13.a shows performance variation over the discounted factor γ. Although oscillating values are observed, the performance generally tends to increase with the increase in γ, reaching its maximum value at 0.98. This value is susceptible to the reward function, so it would be interesting to analyze its variation combined with the function used.

Figure 13.b exposes performance versus variation of the parameter β_a, which controls the action space. The best value is reached at 0.02 and then decreases. A significant cut of 0.05 is observed. As previously stated, these values significantly depend on the rest of the parameters, especially the reward function used. Here its variation is exposed only with the best model reached and maintaining the rest of the variables.

Finally, Figure 13.c shows performance over the parameter β_l, which is the threshold of the load tolerated. The total reward decreases with the increase of this parameter, which makes sense considering that, being more permissive, the algorithm usually makes higher losses. The highest value reached is at 0.01. However, minor values were not analyzed, indicating that there may be room for improvement in this parameter.

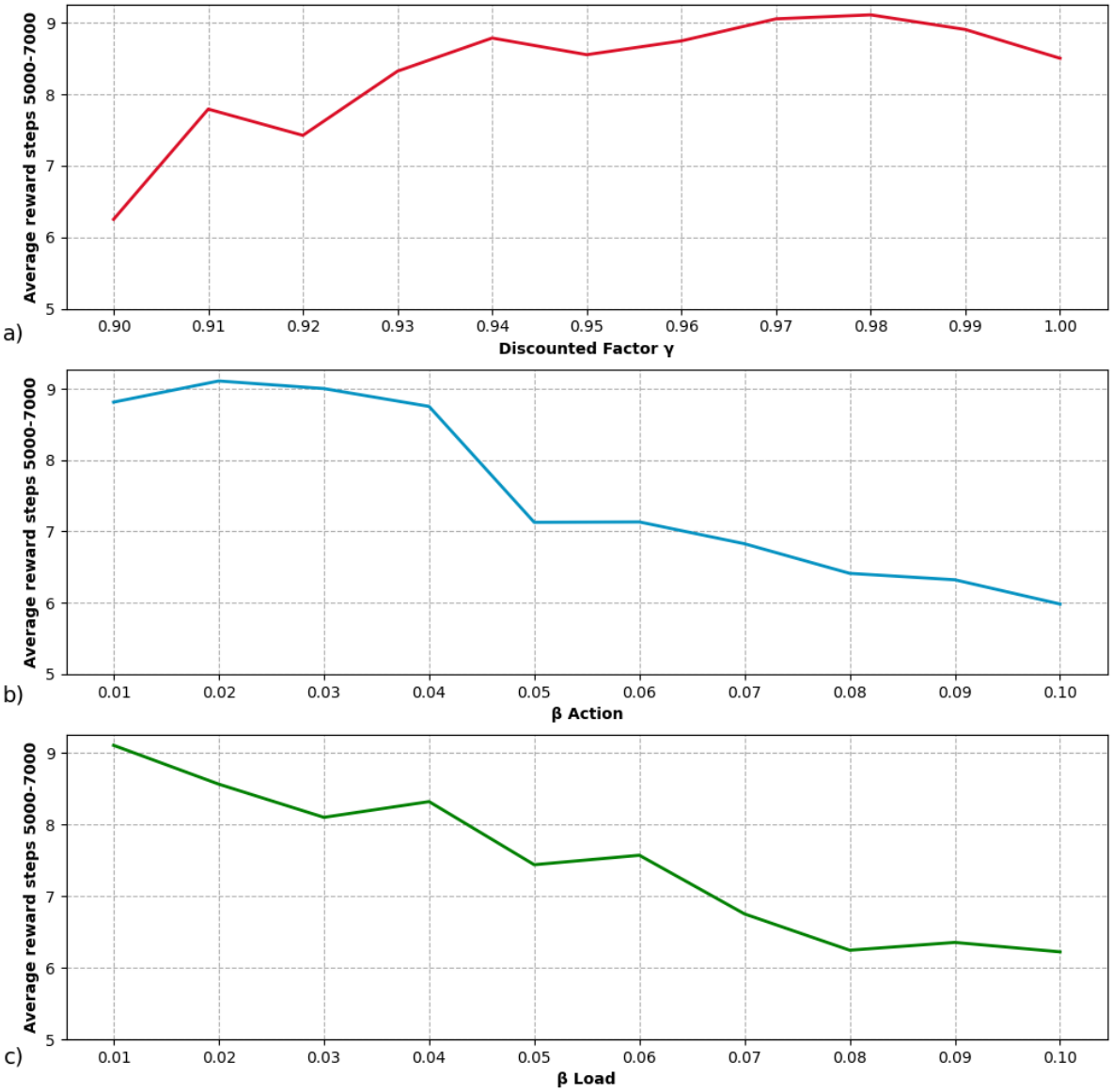

Figure 13. Performance over some hyperparameters variation. Average reward over the last 2000 steps versus a) discounted factor γ, b) β_A threshold variation of the leverage coefficient and c) β_C threshold variation of the load.

6. Conclusions and further research

In the last few years, Decentralized Finance applications have been gaining popularity. In addition, Automated Market Makers are a relatively new phenomenon that is in its early stages of growth. Although the current market solutions are innovative, there is still significant room for improvement. In fact, numerous studies in the area have emerged in the last two years. As the DeFi industry continues to expand and gain more attention, the need for DEXs to compete with CEXs becomes increasingly crucial. However, the exigency to rectify the challenges specific to DEXs remains a prerequisite for accomplishing this objective. These issues, ranging from high slippage and impermanent loss to security, privacy, and user experience concerns, can damage the image and limit the potential growth of DEXs.

This work formalizes and exposes the implicit costs to a liquidity taker and provider and the use of a deep reinforcement learning mechanism for market making. Its primary objective is to solve the slippage and impermanent loss problems in AMMs. Considering the rapid evolution of the industry, a generic AMM framework is selected as the basis for investigation, which means the use of a Hybrid Function Market Maker conservation function.

The work implements a pseudo-arbitrage rule, in order to tackle impermanent loss concerns arising from arbitrage opportunities, which can severely damage the AMM's economy. Following previous studies on this subject, the mechanism implements a trusted external oracle to get the market conditions on the AMM and match the bonding curve to them. Next, to solve the remanent impermanent loss and slippage, the work proposes a deep reinforcement learning algorithm, which aims to keep control of the leverage coefficient of the HFMM conservation function. The work also proposes a price-based event, where each event is identified when a significant price difference occurs, given existing AMM trades that shift the current market price.

Specifically, three Deep Q-Learning models are implemented: a conventional Deep Q-Learning, a Double Deep Q-Learning, and a Double Deep Q-Learning with a prioritized experience replay model. The last one, which applies a proportional prioritization technique, presents the best performance over the load metric. In general, both Double DQN models present a better convergence and more stable curve compared to the traditional DQN model.

Furthermore, one of the findings is that this type of models typically exhibits a substantial degree of variability. Despite potential convergence, they tend to manifest a high sensitivity to both the training data and the own model experience. Another notable aspect is that algorithms tend to follow trends, which can be attributed to their prioritization of short-term occurrences. These issues are partially solved through the development of more complex models, such as the ones proposed in this study. Although the obtained performance is good, the sensitivity to the trained model's environment and initial conditions does not yield a sufficiently robust and replicable model.

This work also presents the performance in response to variation in the hyperparameters that modify the discount factor (γ), action space (β_A), and tolerable load (β_L). Optimal performance was achieved with $\gamma = 0.98$, $\beta_A = 0.02$ and $\beta_L = 0.01$. However, it is important to clarify that the hyperparameter space is vast, and there is room for improvement in most of them.

There are several aspects that could be further enhanced in future research. One of them with particular interest is the incorporation of the fee parameter in the model. This can be accomplished by including it as a constant term that can be easily added to the reward function. Alternatively, an agent could be designed to control the fee's value, aiming to either maximize profits or minimize the losses incurred by liquidity providers and takers. Another potential area of improvement is related to the implementation of a liquidity pool that handles a more significant number of tokens, with a corresponding analysis of the resulting variance. Other possibilities include the implementation of a multi-agent algorithm. In these types of models, multiple learning agents coexist in a unique environment. Each agent is motivated by its own reward and does actions for its own interest, resulting in complex group dynamics (Dixon et al., 2020). In this case, each agent could be a liquidity pool for a different pair of tokens or a different DEX. Another potential improvement would be to dive deeper with hyperparameter tuning of the presented model to improve performance and make it more robust. It is also essential to try different methods in the reward function due to the high dependence of the model on it. Finally, it would be interesting to explore the implementation of policy-base models to apply a probability distribution over actions.

7. References

Aoyagi, J. (2020). Lazy Liquidity in Automated Market Making. *SSRN Electronic Journal*. doi: 10.2139/ssrn.3674178

Aoyagi, Jun and Ito, Yuki (2021). Coexisting Exchange Platforms: Limit Order Books and Automated Market Makers (2021). doi: 10.2139/ssrn.3808755

Bakshaev, A. (2020). Market-making with reinforcement-learning (SAC). Doi: arXiv.2008.12275

Bartoletti, M., Hsin-yu Chiang, J., Lluch-Lafuente, A. (2020). SoK: Lending Pools in Decentralized Finance. doi: arXiv.2012.13230

Bartoletti, M., Hsin-yu Chiang, J., Lluch-Lafuente, A. (2022). A theory of Automated Market Makers in DeFi. doi: arXiv.2102.11350

Bellman R. (1952). On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, USA, 38(8), 716-9. doi: 10.1073/pnas.38.8.716

Bergault, P., Bertucci, L., Bouba, D., Guéant, O. (2022). Automated Market Makers: Mean-Variance Analysis of LPs Payoffs and Design of Pricing Functions. doi: arXiv.2212.00336

Biais, B., Hillion, P., & Spatt, C. (1995). An Empirical Analysis of the Limit Order Book and the Order Flow in the Paris Bourse. *The Journal of Finance*, *50*(5), 1655-1689. doi: 10.2307/2329330

Bichuch, M., Feinstein, Z. (2022). Axioms for Automated Market Makers: A Mathematical Framework in FinTech and Decentralized Finance. doi: arXiv.2210.01227

Cao, J., Chen, J., Hull, J., Poulos, Z. (2021). Deep Hedging of Derivatives Using Reinforcement Learning. doi: arXiv.2103.16409

Carlsson, S., Regnell, A. (2022). Reinforcement Learning for Market Making. Available at: https://kth.diva-portal.org/smash/get/diva2:1695877/FULLTEXT01.pdf

Churiwala, D., Krishnamachari, B. (2022). QLAMMP: A Q-Learning Agent for Optimizing Fees on Automated Market Making Protocols. doi: arXiv.2211.14977

Dixon, M., Halperin, I., Bilokon, P. (2020). *Machine Learning in Finance: From Theory to Practice*. Cham: Springer. doi: 10.1007/978- 3- 030- 41068- 1

Egorov, Michael (2019). StableSwap - efficient mechanism for Stablecoin liquidity. Available at: https://berkeley-defi.github.io/assets/material/StableSwap.pdf

Egorov, Michael (2021). Automatic market-making with dynamic peg. Available at: https://classic.curve.fi/files/crypto-pools-paper.pdf

Engel, D., Herlihy, M. (2021). Composing Networks of Automated Market Makers. doi: arXiv:2106.00083

Engel, D., Herlihy, M. (2021). Presentation and Publication: Loss and Slippage in Networks of Automated Market Makers. doi: arXiv.2110.09872

Fritsch, R. (2021). Concentrated Liquidity in Automated Market Makers. doi: arXiv:2110.01368

Fritsch, R., Käser, S., Wattenhofer, R. (2022). The Economics of Automated Market Makers. doi: arXiv:2206.04634

Heimbach, L., Schertenleib, E., Wattenhofer, R. (2022). Risks and Returns of Uniswap V3 Liquidity Providers. doi: arXiv.2205.08904

Jensen, J. R., Pourpouneh, P., Nielsen, K., Ross, O. (2021). The Homogenous Properties of Automated Market Makers. doi: arXiv.2105.02782

Jumadinova, J., Dasgupta, P. (2010). A Comparison of Different Automated Market-Maker Strategies. Available at: https://www.semanticscholar.org/paper/A-Comparison-of-Different-Automated-Market-Maker-Jumadinova-Dasgupta/bef03da46b778d2cc305907452da1598b6b15be5

Khakhar, A., Chen, X. (2022). Delta Hedging Liquidity Positions on Automated Market Makers. doi: arXiv.2208.03318

Kuan, J. H. (2022). Liquidity Provision Payoff on Automated Market Makers. doi: arXiv.2209.01653

Kanwar, N. (2019). Deep Reinforcement Learning-based Portfolio Management. Presented to the Faculty of the Graduate School of The University of Texas at Arlington. Available at: https://rc.library.uta.edu/uta-ir/bitstream/handle/10106/28108/KANWAR-THESIS-2019.pdf

Krishnamachari, B., Feng, Q., Grippo, E. (2021). Dynamic Curves for Decentralized Autonomous Cryptocurrency Exchanges. doi: arXiv.2101.02778

Kumar, P. (2020). Deep Recurrent Q-Networks for Market Making. Paper presented at The Thirteenth Conference on Artificial General Intelligence. AGI-20 Virtual Conference. Available at: http://agi-conf.org/2020/wp-content/uploads/2020/06/AGI-20_paper_39.pdf

Lapan, M. (2020). *Deep Reinforcement Learning Hands-On.* Birmingham, U.K.: Packt.

Lehar, A., Parlour, C. A. (2021). Decentralized Exchanges. doi: 10.2139/ssrn.3905316

Lim, T. (2022). Predictive Crypto-Asset Automated Market Making Architecture for Decentralized Finance using Deep Reinforcement Learning. doi: arXiv.2211.01346

Liu, C. (2020). Deep Reinforcement Learning and Electronic Market Making. Available at: https://www.imperial.ac.uk/media/imperial-college/faculty-of-natural-sciences/department-of-mathematics/math-finance/Chenyu-Liu.pdf

Lucarelli, G., Borrotti, M. (2019). A Deep Reinforcement Learning Approach for Automated Cryptocurrency Trading. doi: 10.1007/978-3-030-19823-7_20

Mani, M. A., Phelps, S. (2019). Applications of Reinforcement Learning in Automated Market-Making. Available at: http://www.agent-games-2019.preflib.org/wp-content/uploads/2019/05/GAIW2019_paper_5.pdf

Markovich, S. (2021). Transparency and Learning: Evidence from Defi Markets. doi: 10.2139/ssrn.3962517

Milionis, J., Moallemi, C. C., Roughgarden, T., Lee Zhang, A. (2022). Automated Market Making and Loss-Versus-Rebalancing. doi: arXiv:2208.06046

Mosavi, A., Ghamisi, P., Faghan, Y., Duan, P., et al. (2020). Comprehensive Review of Deep Reinforcement Learning Methods and Applications in Economics. *Arxiv*, 1-45. doi: 10.48550/arXiv.2004.01509

Rao, A., Jelvis, T. (2022). Foundations of Reinforcement Learning with Applications in Finance. Available at: https://stanford.edu/~ashlearn/RLForFinanceBook/book.pdf

Raynor de Best (2023, March 31). *Decentralized Finance (DeFi) - statistics & facts*. Statista. Available at: https://www.statista.com/topics/8444/decentralized-finance-defi

Ravichandiran S. (2020). *Deep Reinforcement Learning with Python*. Birmingham, U.K: Packt.

Sadighian, J. (2019). Deep Reinforcement Learning in Cryptocurrency Market Making. doi: arXiv.1911.08647

Sadighian, J. (2020). Extending Deep Reinforcement Learning Frameworks in Cryptocurrency Market Making. doi: arXiv.2004.06985

Singh, V., Chen, S., Singhania, M., Nanavati, B., Kumar Kar, A., Gupta, A. (2022). How are reinforcement learning and deep learning algorithms used for big data based decision making in financial industries–A review and research agenda. *International Journal of Information Management*, 2(1), 1-15. doi: 10.1016/j.jjimei.2022.100094

Spooner, T., Savani, R. (2020). Robust Market Making via Adversarial Reinforcement Learning. doi: arXiv.2003.01820

Sun, T., Huang, D., Yu, J. (2022). Market Making Strategy Optimization via Deep Reinforcement Learning. doi: 10.1109/ACCESS.2022.3143653

Tiruviluamala, N., Port, A., Lewis, E. (2022). A General Framework for Impermanent Loss in Automated Market Makers. doi: arXiv.2203.11352

Van Hasselt, Hado. (2010). Double Q-learning. Available at: https://papers.nips.cc/paper/3964-double-q-learning

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. doi: *ArXiv*. /abs/1509.06461

Wang, Y. (2020). Automated Market Makers for Decentralized Finance (DeFi). doi: arXiv.2009.01676

Xu, J., Paruch, K., Cousaert, S., Feng, Y. (2022). SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. doi: arXiv.2103.12732

8. Data and code

The data and code for this project are available at https://github.com/agustinparrotta/drl-in-amm. Follow instructions from the *README.md* file located in the root folder to replicate results.