Optimizing Liquidity Provision in Uniswap V3 Using Reinforcement Learning

Kurlovich Nikolai
MSc Quantitative Finance
ETH Zürich
nkurlovich@student.ethz.ch

Rozman Anej
MSc Quantitative Finance
ETH Zürich
arozman@student.ethz.ch

Hachimi Mehdi
MSc Quantitative Finance
ETH Zürich
mhachimi@student.ethz.ch

Joly Julien
MSc Applied Mathematics
ETH Zürich
jujoly@student.ethz.ch

Abstract

Uniswap v3 introduces concentrated liquidity, enabling liquidity providers to allocate capital within custom price ranges. While this innovation enhances capital efficiency, it also exposes LPs to increased risk from impermanent loss and suboptimal positioning. In this paper, we formulate the optimal liquidity provision problem in Uniswap v3 as a sequential decision-making task and propose a Deep Reinforcement Learning framework to learn dynamic, data-driven liquidity management policies. In addition, we replicate key analytical results to ensure consistency with established theoretical foundations. As part of this effort, we generate visualizations of their formulas within the Uniswap v2 setting, providing intuitive insights into how LPs are affected by the mechanics of simpler pools.

1 Introduction

Decentralized finance has transformed digital asset management, with decentralized exchanges (DEXs) and their underlying automated market makers (AMMs) emerging as core infrastructure. Uniswap V3 represents a significant evolution in AMM design by introducing concentrated liquidity, a feature that enhances capital efficiency. This innovation allows liquidity providers (LPs) to allocate capital within specific price ranges, but in doing so, it transforms a once-passive activity into a complex, active management task. LPs must now dynamically select price bounds to balance fee generation against market risk. A poorly chosen range can result in periods with no fee income or expose the provider to losses as prices shift. Furthermore, because any adjustments to these ranges incur transaction costs, optimizing profitability requires LPs to navigate a complex trade-off between maximizing fee income, minimizing transaction costs, and managing the risks posed by volatile price dynamics.

In this work, we frame liquidity provision in Uniswap V3 as a sequential decision-making problem and propose a deep reinforcement learning (DRL) approach to learn adaptive strategies that maximize net returns. Our contributions are the following, we design and train RL agents capable of learning when and how to rebalance liquidity ranges, and we evaluate performance using historical price and transaction data, accounting for gas costs, impermanent loss and rebalancing costs. By doing so, we offer an entire model-free framework for automated liquidity management in AMMs.

To the best of our knowledge, this is the first work to propose such a general action space and comprehensive objective function that jointly account for all key practical factors faced by a liquidity provider, including fee income, impermanent loss, gas and opportunity costs.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

2 Related Works

A specific line of research investigates optimal liquidity provision strategies in Uniswap-style AMMs, aiming to maximize LP returns while mitigating risks such as impermanent loss and fluctuating fees. For example, (3) leverage a neural network-based framework to dynamically adjust liquidity ranges in response to price movements, optimizing the trade-off between rewards and gas costs. (4) implements a DRL strategy that integrates centralized futures market hedging with dynamic price range allocation, explicitly accounting for the loss-versus-rebalancing (LVR) effect as a rebalancing signal.

The very recent work by Xu and Brini (5) is the closest to ours in methodology, applying DRL for automated liquidity management. However, their approach drastically simplifies the action space, limiting it to a discrete set of two to five tick-size choices—e.g., $\mathcal{A} = \{0, 10, 20, 30, 40\}$—which define the width of a symmetric price interval centered at the current market price. This reduced action space is then optimized as a hyperparameter. In our work, we aim for broader generality, allowing for a significantly richer and more flexible action space that captures a wider range of LP behaviors. Our work is completely model-free as we impose very little assumptions and do not use any model dependent formula (like LVR). This generalization enables us to explore more realistic and nuanced liquidity provision strategies that better reflect the complexity faced by active participants in decentralized markets.

3 Decentralized Market Making and Uniswap V3 Mechanics

AMMs revolutionized decentralized trading by replacing traditional order books with algorithmic price discovery mechanisms. The foundational constant product AMM (CPAMM), introduced in Uniswap V1 and refined in V2, operates on the invariant $x \cdot y = k$, where x and y represent the reserves of two tokens in a liquidity pool, and k is a constant. For example, if we want to buy Δx from the liquidity pool, we will have to provide Δy such that

$$(x - \Delta x)(y + (1 - r)\Delta y) = k$$

holds. Here r represents the fee tier of the pool. Fees are distributed to LPs proportional to their investment in the pool. This elegant formulation ensures that liquidity is distributed uniformly across all possible price ranges from zero to infinity. While this uniform distribution provides continuous liquidity, it suffers from significant capital inefficiency. In practice, the majority of trading activity occurs within relatively narrow price ranges around the current market price. For instance, stablecoin pairs like USDC/USDT rarely deviate more than 1% from parity, yet the constant product formula allocates equal liquidity to price ranges far from the current price where trading is unlikely to occur. This inefficiency manifests itself in higher slippage for traders (the difference between expected and actual execution prices due to insufficient liquidity and time of trade execution) and suboptimal returns for liquidity providers.

Uniswap V3 addresses these limitations through concentrated liquidity, allowing LPs to allocate capital within custom price ranges $[P_l, P_u]$ where P_l and P_u represent the lower and upper price bounds respectively. Within this range, the position acts as a constant product AMM but the liquidity is spread out over a smaller interval making it more efficient. This concentrated liquidity results in significantly higher liquidity density within the active price range compared to a uniform distribution, thereby reducing slippage (how much a trade affects the price) for trades executed within that range. The mathematical foundation builds upon the constant product formula but applies it only within the specified range. For a position with liquidity L concentrated between prices P_l and P_u, the virtual reserves are given by:

$$x_v = L \cdot \left(\frac{1}{\sqrt{P}} - \frac{1}{\sqrt{P_u}}\right) \qquad y_v = L \cdot \left(\sqrt{P} - \sqrt{P_l}\right) \tag{1}$$

Here, $P = \frac{y}{x}$ is the current price, and L is the liquidity constant that determines the size of the position. The derivation of equations (1) can be found in Appendix A.1. In practice, the LP first selects a price range $[P_l, P_u]$, then decides how much of asset x (or y) to contribute. This determines the liquidity constant, which dictates how much of asset y (or x) must be deposited into the position. When providing liquidity in Uniswap V3, LPs must explicitly define their position by selecting a price range $[P_l, P_u]$ and a fee tier $(0.01\%, 0.05\%, 0.3\%, 1\%)$ at the time of deposit. These parameters determine where their liquidity is active and the rate at which they earn fees. LPs can open multiple

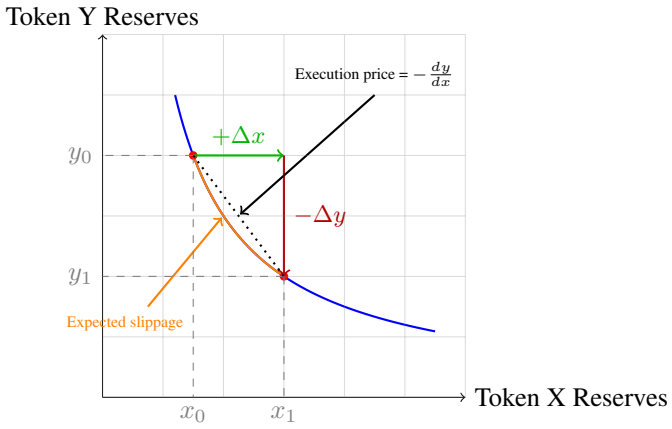

Figure 1: Uniswap V2 showing a token swap. Expected slippage is the price change based on current reserves. Unexpected slippage may occur if other trades are executed between the time when we submit our trade and its execution.

positions with different configurations to approximate arbitrary liquidity curves or to adapt to market conditions. This flexibility turns liquidity provision into an active portfolio management task.

When the current price P lies within the specified range $[P_l, P_u]$, the position provides both tokens and earns a share of the trading fees. As the price moves toward the boundaries of the range, the asset composition shifts toward a single token. If the price falls below P_l, the position holds only token x; if it rises above P_u, it holds only token y. Out-of-range positions do not participate in trading and therefore do not earn fees.

Fee generation is proportional to the share of active liquidity at the current price. The protocol supports multiple fee tiers to accommodate different levels of volatility and trading frequency across asset pairs. The fee earning rate for a position with liquidity L_i at price P is given by:

$$\text{Fee Rate}_i = f \cdot \frac{L_i}{\sum_j L_j(P)} \cdot \text{Trading Volume}(P) \tag{2}$$

where f is the pool's fee tier and the denominator represents the total active liquidity at price P. This mechanism incentivizes LPs to concentrate liquidity where trading activity is highest, leading to natural competition for the most profitable price ranges.

Fees accumulate over time but are not automatically reinvested. They must be manually collected by the LP. While reinvesting earned fees can increase a position's future fee-generating potential, the benefit must be weighed against the gas fees associated with rebalancing, especially in high-fee blockchains.

Gas fees represent a significant friction that directly impacts optimal LP strategies and motivates our RL approach. Every position modification, whether creating new positions (minting), closing existing ones (burning), or collecting accumulated fees requires on-chain transactions that consume gas. These fees create the fundamental tension between theoretical optimality of continuous rebalancing and practical constraints that our RL framework must navigate. Gas fees vary significantly with network congestion and position complexity, and during periods of high network activity, they can exceed the fees earned from small positions, making frequent rebalancing economically unviable. This creates a natural barrier to high-frequency strategies.

The concentrated liquidity design fundamentally alters the risk-return profile compared to uniform liquidity pools. In DEXs like Uniswap, LPs deposit two tokens into a smart contract to enable users to swap between them. In return, LPs earn a share of the swap fees. However, they are also exposed to market risk — if the relative price of the tokens changes, LPs may end up with fewer valuable assets than if they had simply held the tokens outside the pool. This phenomenon is often referred to as Impermanent Loss (IL). In Uniswap V3 the latter exhibits different characteristics, as it can either amplify or reduce IL depending on the range width and price movement patterns. For a concentrated position with initial price P_0 and current price P, the impermanent loss can be expressed as:

$$IL = V_{pool}(P) - V_{hold}(P) \tag{3}$$

where $V_{pool}(P)$ represents the position value at current price and $V_{hold}(P)$ represents the value of holding the initial tokens without providing liquidity.

The portfolio's value at time t of the LP holding x_t amount of ETH and y_t amount of USDC is $V_t = x_t \cdot P_t + y_t$. Then the change in portfolio's value from time t to $t+1$ is given by

$$\Delta V_{t+1} = V_{t+1} - V_t = x_{t+1}P_{t+1} - x_t P_t + y_{t+1} - y_t \tag{4}$$

Uniswap V3 implements a tick-based system that discretizes the continuous price space into discrete price points, enabling efficient on-chain computation. Each tick represents a 0.01% price movement, with the relationship between tick index i and price given by $p(i) = 1.0001^i$.

4 Replication of AMM & LVR, Millionis et al. (2024)

The Loss-versus-Rebalancing (LVR) is an attempt to quantify the loss that an LP experiences compared to a hypothetical trader who continuously rebalances their portfolio to match the LP's token composition but always does so off-chain without participating in the AMM. This rebalancing strategy mimics the LP's position, but avoids trading fees and slippage. Liquidity provision in AMMs is equivalent to a passive, lagged rebalancing strategy that is systematically "picked off" by traders with better timing. In other words, LPs provide liquidity, but traders use this liquidity to trade at stale prices when the true market price has moved. This creates a structural loss for the LP, which is exactly what LVR measures.
To empirically validate the theoretical findings of the LVR, we replicate a core set of results using publicly available data from a major Uniswap V2 pool. The goal of this replication is twofold: first, to assess the coherence of pool P&L decompositions through empirical data; and second, to demonstrate the alignment between hedged LP P&L and the fees-minus-LVR metric.

4.1 Pool PnL Decomposition and Hedging Strategies

The liquidity provider's P&L can be decomposed into several components. Denoting the pool value at time t by V_t, the dollar value of mints and burns by Π_t^{mint} and Π_t^{burn}, and the price of the risky asset by P_t, the pool's discrete-time P&L can be written as:

$$\Delta\text{LP}_t = V_t + \Pi_t^{\text{burn}} - \Pi_t^{\text{mint}} - V_{t-1}.$$

This unhedged LP PnL can be compared to artificial delta-hedged strategies, where the LP adjusts their exposure to the risky asset periodically. These strategies are constructed by shorting LP's holding in the risky asset at the beginning of each interval (e.g., minutely, hourly, daily). The hedging P&L for a frequency $\mathcal{F}$ is:

$$\Delta\text{RB}_t^{\mathcal{F}} = x_t^{\mathcal{F}} \cdot (P_t - P_{t-1}),$$

where $x_t^{\mathcal{F}}$ is the ETH quantity held by the LP at the beginning of the interval defined by $\mathcal{F}$. The hedged P&L is then defined as:

$$\Delta\text{Hedged}_t^{\mathcal{F}} = \Delta\text{LP}_t - \Delta\text{RB}_t^{\mathcal{F}}. \tag{5}$$

Note that this is a discrete adaptation of the continuous-time formula obtained in (2):

$$\text{LP P\&L}_t - \int_0^t x^\star(P_s)dP_s = \text{FEE}_t - \text{LVR}_t$$

4.2 Fees, Volatility and Loss-versus-Rebalancing

According to the LVR theory, for Uniswap V2 pools, the cost incurred by LPs in providing liquidity without continuous rebalancing can be attributed to a term that scales with volatility, specifically:

$$\text{LVR}_t = \frac{\sigma_t^2}{8} \cdot V_t \cdot \Delta t,$$

where σ_t^2 is the daily realized variance of log returns on the risky asset, and Δt is the time step (e.g., one minute). On the other hand, LPs earn fees from swap activities, which we denote FEE_t. Aggregating over time, the net benefit to LPs from exposure to flow risk can be approximated by:

$$\Delta_t = \text{FEE}_t - \text{LVR}_t.$$

4.3 Empirical Findings and Interpretation

Our replication confirms a strong alignment between the cumulative hedged P&L and the cumulative *fees-minus-LVR* term, as predicted by the theory. The accompanying Jupyter Notebook implementing the relevant computations and visualizations is available in the Github repository under the name *replication_lvr.ipynb*. In line with Figure 5(b) from (2), we observe in Figure 3, for this specific Uniswap V2 pool, that the behavior of the residual term Δ_t closely tracks that of the hedged portfolio, providing further empirical support for the LVR framework.

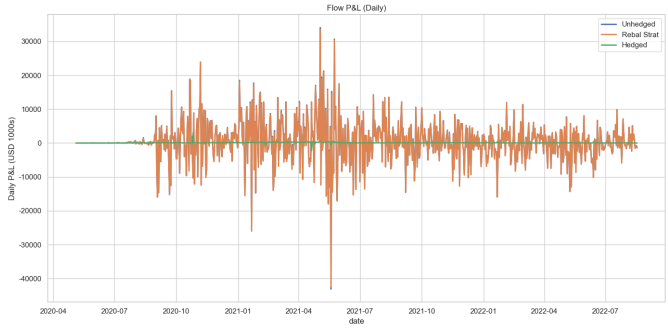

Figure 2: Unhedged LP P&L, Rebalancing Strategy P&L, and Hedged LP P&L

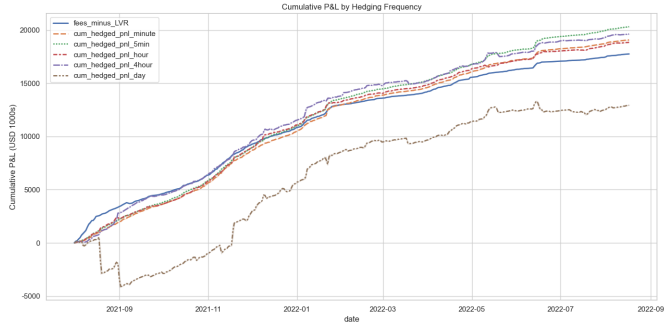

Figure 3: Delta-hedged P&L and predicted P&L from our expressions for LVR, for the Uniswap v2 WETH-USDC trading pair.

5 Problem Setting

Providing liquidity involves navigating a fundamental trade-off between the width of the liquidity provision interval and the fees accrued over the investment horizon. Expanding the interval reduces the proportion of active liquidity, thereby decreasing potential rewards. Conversely, if the interval is too narrow, the position risks becoming inactive once the price moves outside the specified range, halting fee generation. The interval is parametrized by its lower and upper bounds $(P_l, P_u)_{t\geq 0}$ and the objective is to determine these bounds in an optimal manner.
Additionally, gas fees can be significant must be accounted for as it can impact the strategy's profitability. Each rebalancing event incurs gas fees twice: once for burning the existing position and again for minting a new one. Importantly, in our setting, we do not assume gas fees to be constant (as assumed in related works), a choice that better reflects real-world conditions. Learning

optimal policies in this setting therefore presents significant challenges. As the choice of model and environment directly impacts agent performance, careful design is essential.

6 The Model

6.1 Proximal Policy Optimization Algorithm (PPO)

Policy gradient methods directly optimize the parameters of a stochastic policy $\pi_\theta(a \mid s)$ by estimating the gradient of the expected return:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\hat{A}_t\right] \tag{6}$$

While conceptually elegant and empirically powerful, vanilla policy gradient methods suffer from high variance (due to Monte Carlo estimates), instability (large policy updates) and poor sample efficiency. To address instability, natural policy gradients propose to account for the geometry of the policy space. The key insight is that naive gradient steps in θ-space may cause disproportionately large changes in the behavior of the policy. To correct this, natural gradients adjust the direction of the update by the Fisher Information Matrix $F(\theta)$:

$$\tilde{\nabla}_\theta J(\theta) = F^{-1}(\theta)\nabla_\theta J(\theta) \tag{7}$$

This can be viewed as performing gradient ascent in a Riemannian space, where the Fisher information matrix is the Riemannian metric describing the curvature of a statistical manifold:

$$\text{maximize } J(\theta) \quad \text{subject to } \mathcal{D}_{\text{KL}}(\pi_\theta \parallel \pi_{\theta+\Delta\theta}) \leq \epsilon$$

In practice, natural gradient methods are numerically brittle and do not always yield stable outcomes. Additionally, computing a matrix inverse is an operation of $O(N^3)$ complexity, which is rather tedious. Thus, for deep RL methods, natural policy gradients typically exceed both memory- and computational limits. Building on this idea, TRPO (Schulman et al., 2015) formalizes a constrained optimization problem:

$$\text{maximize } \mathbb{E}_{(s,a)\sim\pi_\theta}\left[\frac{\pi_{\theta+\Delta\theta}(a \mid s)}{\pi_\theta(a \mid s)}\hat{A}^{\pi_\theta}(s,a)\right] \text{ subject to } \mathbb{E}_s\left[\mathcal{D}_{\text{KL}}(\pi_\theta(\cdot \mid s) \parallel \pi_{\theta+\Delta\theta}(\cdot \mid s))\right] \leq \epsilon$$

This ensures that each update stays within a trust region, limiting how much the policy is allowed to change at each iteration. TRPO uses a second-order approximation of the KL divergence constraint and solves a constrained quadratic programming problem using conjugate gradient descent, making it relatively complex to implement and computationally intensive. This motivated the development of PPO — a method that preserves the core idea of a trust region, but in a first-order, more scalable way. PPO (Schulman et al., 2017) introduces a surrogate objective that penalizes updates leading to large changes in the policy:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left\{r_t(\theta)\hat{A}_t; \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right\}\right] \tag{8}$$

where $r_t(\theta) = \frac{\pi_{\theta+\Delta\theta}(a_t|s_t)}{\pi_\theta(a_t|s_t)}$ and ϵ is an hyperparameter (typically 0.1 or 0.2). This objective encourages updates that improve the advantage but clips the objective when the policy changes too much. As PPO is a first-order approximation of the trust region, there is no need for line search or Fisher matrix.

6.2 The Environment

Table 1 gives a summary of the design we chose to help the agent in its learning task. The details are given in Appendix B.2.1.

6.3 Our Architecture

Throughout this paper, we adopt a simple two-layer feedforward neural network. We chose this architecture because our state representation—whether raw agent descriptors or hand-crafted features—already embeds substantial domain knowledge. Convolutional layers would offer little benefit,

Component	Description
Observation space	• $\widetilde{p}_t$, p_t: pool and market prices • w_t: width from previous action • Current share of liquidity held in the pool • Current pool reserves • Agent's wealth • Additional market features (cf. Appendix B.2.1)
Action space	Continuous vector $a_t = (e, w_t)$: where $e \in \{0, 1\}$ encodes the decision to provide liquidity (or not) and the width $w \in [1, 50]$ (rounded to int), determining the range around the current tick tick_c in which the liquidity is provided: $\text{tick}_\ell = \text{tick}_c - 10\,w \qquad \text{tick}_u = \text{tick}_c + 10\,w$
Reward function	• $P\&L_t = \sum_{s \le t} AF_s + IL_s + \Delta V_s$ **(for the unhedged strategy)** • $P\&L_t = \sum_{s \le t} AF_s + IL_s - GF_s$ **(for the hedged strategy)** where AF_s are fees accrued from swaps, IL_s represents the Impermanent Loss, GF_s are gas fees (mint, burn, collect) and ΔV_s is the change of the portfolio's value in the pool.

Table 1: Overview of the Gym environment of the Model.

and recurrent models are ill-suited since our input does not encode temporal sequences that need to be tracked through time. For our reinforcement-learning experiments, we conducted a grid search over learning rate and episode length to identify optimal hyperparameter configurations. In particular, we increased the entropy coefficient to encourage broader exploration: in DeFi liquidity provision, agents often mint tokens and then must wait, making meaningful actions sparse and exploration slow.

7 Experiments

7.1 Unhedged P&L

As presented in Table 1, the reward function is the following P&L:

$$P\&L_t = \sum_{s \le t} AF_s + IL_s + \Delta V_s$$

This reflects actual gains and losses experienced by the LP, without any hedging. We illustrate the temporal evolution of this unhedged P&L, in the following. We observe in Figure 4 that the unhedged

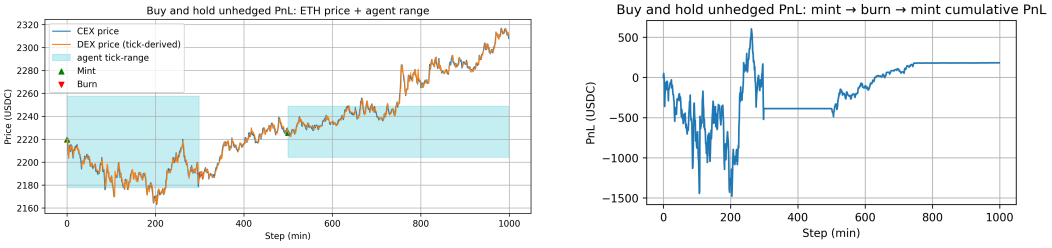

Figure 4: P&L of the unhedged agent and visualization of a deterministic strategy.

P&L exhibits significant volatility and is highly sensitive to fluctuations in the ETH price. In the initial phase of the episode, the agent selects a range that is soon crossed from below, resulting in losses. This is primarily due to the agent holding a portfolio concentrated entirely in ETH, which declines in value as ETH prices fall. Although the strategy ultimately yields a positive P&L, it remains heavily exposed to the directional movement of the underlying asset, making it inherently

risky. As a consequence, the agent gradually learns to refrain from trading. Indeed when learning, in most cases,it loses around $1 - 5\%$ of its initial wealth and thus decides to not provide liquidity. To mitigate this dependence on market trends, we introduce in the following section a hedged strategy.

7.2 Hedged P&L and Impermanent Loss

As seen in the case where no hedge is performed, the strategy is risky and a LP will try to avoid these kind of risks. Thus, as done in (2), we implement a rebalancing strategy that mimic the behavior of a delta-hedged position. At each decision time, the agent adjusts its exposure to the risky asset (ETH) by shorting its ETH holding at the start of the interval. This approach ensures that the agent is locally neutral to price movements over the interval, as gains or losses from price fluctuations are offset by changes in the value of the ETH position—akin to a classical delta hedge in continuous-time finance. By comparing the liquidity provider's P&L with the P&L of the rebalancing strategy, we isolate the component of the LP's risk attributable to exposure beyond pure market-making activity. The reward function is then:

$$P\&L_t = \sum_{s \leq t} AF_s + IL_s - GF_s \tag{9}$$

This analysis mirrors the earlier replication work. In particular, we analyze whether the fees accrued through swap activity are sufficient to offset the two primary sources of loss: the intrinsic loss incurred by any LP entering the pool and gas fees. If those fees are too high, then the agent will learn to stay out of the pool. It is known that market-making is not an easy task, specially with limited resources. In the following plots, we investigate if the RL agent is capable of learning strategies that yields profits. To motivate the agent entering in the pool, we introduce an incentive parameter $1 > \zeta > 0$ that boost trading activities through P&L regularization. The parameter rewards the agent more, if it is being active.

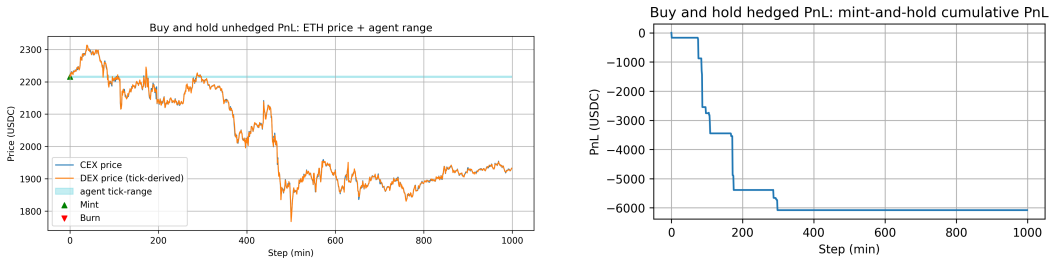

Figure 5: P&L of the hedged agent and visualization of a deterministic strategy.

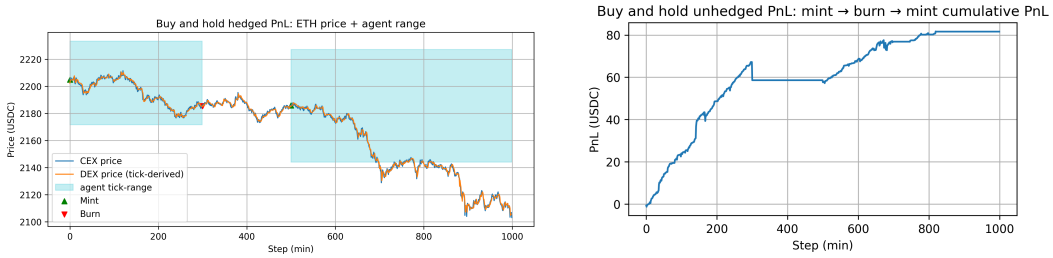

Figure 6: P&L of the hedged agent and visualization of a deterministic strategy.

Figure 5 illustrates a scenario in which the selected range is too narrow, resulting in significant losses. The high impermanent loss observed is driven by substantial price fluctuations, while the narrow range prevents the agent from earning meaningful fee income, thereby exacerbating the negative outcome. Figure 6 depicts the opposite approach, where the agent opts for a wide range, resulting in a steady accrual of fees, albeit in smaller amounts. This strategy can yield linearly increasing profits over time as shown in the figure. However, despite this potential, the average returns across most training iterations remain negative. The comparison highlights the trade-offs between narrow and

wide range strategies, and the key mechanisms the agent must learn. Notably, the agent struggles to maintain its position once it starts incurring losses, often exiting prematurely.

7.3 One-dimensional actions Experiment

In order to force the agent to enters the pool and provide liquidity, we can modify the design of the space of actions. Removing the engagement flag $e \in \{0, 1\}$ transforms the problem into a one-dimensional policy optimization problem. The agent must find optimal widths to quote in order to maximize its profits. In this configuration, the agent is now trading and actually is able to make profits in some cases. We also put a plot where it incurs losses at the end to show that training can lead to diverse outcomes. As illustrated in Figure 7, an agent constrained to continuously provide liquidity

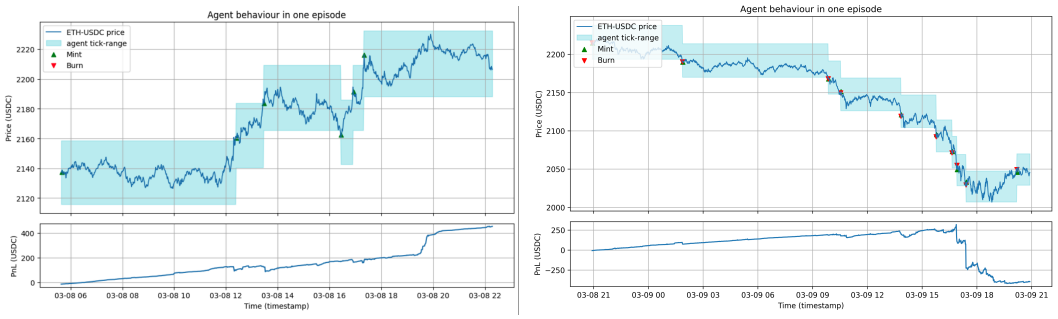

Figure 7: Invested RL agent P&L strategies

can, in certain cases, perform reasonably well. However, during periods of elevated volatility, such as is the case on the right, suffers significant losses driven by Impermanent Loss. While this represents the best performing strategy identified in our experiments, the agent's performance remains generally suboptimal. As reported in the accompanying active rebalancing Jupyter notebook available in the project repository, the average return of the agent is -66.89, with a standard deviation of 84.41, highlighting both low profitability and high variability.
A table comparing the results from the different approaches can be found in Appendix B.3.

8 Conclusion

Our findings indicate that, within the general RL framework we propose, the agent struggles to consistently learn profitable strategies even though such strategies may exist. One of the primary challenges stems from impermanent loss, which significantly erodes performance, while accrued fees often fail to sufficiently compensate for these losses. But when we enforce trading, then the agent is learning some profitable strategies. This suggests that the one-dimensional problem is easier to learn in our environment. This underscores the fundamental difficulty of liquidity provision in volatile markets, particularly when adopting general models that relax many simplifying assumptions.
Importantly, our work tries to extend the relatively new literature by tackling this problem in a broader, more realistic setting. This added generality introduces considerable complexity and highlights the challenges in designing robust learning agents for continuous-time market-making tasks.
The other key objective of our study was to replicate and validate the theoretical insights of the paper we were assigned. In that regard, we successfully reproduced the main empirical findings, and our results strongly support the core tenets of the LVR theory. However, it is worth noting that several assumptions underpinning the LVR theory are not satisfied in real-world settings.

Acknowledgments

We would like to express our sincere gratitude to our supervisor, Prof. Patrick Cheridito, for his guidance, insightful discussions, and constructive suggestions throughout the course of this project. We also thank Dr. Nino Antulov-Fantulin for his helpful feedback and thoughtful remarks. Finally, we are grateful to Dr. Arthur Gervais for his kind assistance with the data collection process and for sharing resources that were crucial to the empirical analysis.

References

[1] Álvaro Cartea, Fayçal Drissi, and Marcello Monga. Decentralised finance and automated market making: Predictable loss and optimal liquidity provision. *Available at SSRN 4273989*, 2022.

[2] Milionis, J., Moallemi, C.C., Roughgarden, T., Zhang, A.L., 2022a. Automated market making and loss-versus rebalancing. *arXiv preprint arXiv:2208.06046*.

[3] Fan, Z.; Marmolejo-Cossio, F.; Moroz, D.J.; Neuder, M.; Rao, R.; and Parkes, D. C. 2021. Strategic liquidity provision in uniswap v3. *arXiv preprint arXiv:2106.12033*.

[4] Zhang, H.; Chen, X.; and Yang, L. F. 2023. Adaptive Liquidity Provision in Uniswap V3 with Deep Reinforcement Learning. *arXiv preprint arXiv:2309.10129*.

[5] Xu, H., and Brini, A. Improving DeFi Accessibility through Efficient Liquidity Provisioning with Deep Reinforcement Learning. *arXiv preprint arXiv:2501.07508*, 2025.

A Introduction to Decentralized Finance

A.1 Mechanism of Uniswap V3

We first give an overview of the Uniswap V2 mechanism. We have first that

$$L^2 = \kappa = xy$$

where κ is a constant called the *depth* of the pool (here the range). L is the liquidity and x, y are the (current) reserves (amounts held) of the two tokens. This relation must always hold.

During a trade (swap) the depth must remain the same. Suppose that we want to buy some amount of token y. The following formula holds:

$$(x + r\Delta x)(y - \Delta y) = \kappa$$

where Δx is the amount of token x we give to receive the amount Δy. Note that the term $r = 1-$swap fee account for the fee that we pay to do the transaction. From this relation we can obtain the output amount if we know the input we give:

$$\Delta y = \frac{yr\Delta x}{x + r\Delta x} \quad \text{and} \quad \Delta x = \frac{x\Delta y}{r(y - \Delta y)}$$

Token prices are $P_x = \frac{y}{x}$ for token x and $P_y = \frac{x}{y}$ for token y. Note that the previous formulas free us from calculating prices ! We now look at the core of the paper: Uniswap V3. The mechanism is the same as V2 if we are within a price range.
For computational precision, we now work only with L (instead of κ) and $\sqrt{P}$ (instead of P):

$$L = \sqrt{xy} \quad \text{and} \quad \sqrt{P} = \sqrt{\frac{y}{x}}$$

Note that here, we use the convention of Uniswap V3 when expressing the price of tokens. WLOG, we will use only the one of x (since they are reciprocals of each other). L and $\sqrt{P}$ are related by the relation:

$$L = \frac{\Delta y}{\Delta\sqrt{P}}$$

and thus we obtain

$$\Delta y = \Delta\sqrt{P}L \quad \text{and} \quad \Delta x = \Delta\frac{1}{\sqrt{P}}L$$

which is exactly the formula for the virtual reserves 1. These formulas allow us to not store and update pool reserves. Also, we don't need to calculate $\sqrt{P}$ each time because we can always find $\Delta\sqrt{P}$ and its reciprocal.

In Uniswap V3, the entire price range is parametrized by evenly distributed discrete ticks with distance of 1 basis point (0.01%). Each tick has an index and corresponds to a certain price:

$$\sqrt{p(i)} = \sqrt{1.0001^i} \implies i = \log_{\sqrt{1.0001}}\left(\sqrt{p(i)}\right) \tag{10}$$

Ticks are finite integers that can be positive and negative. Uniswap V3 stores $\sqrt{P}$ as a fixed point Q64.96 number, which is a rational number that uses 64 bits for the integer part and 96 bits for the fractional part. Thus, prices (equal to are within the range: $[2^{-128}, 2^{128}]$. And ticks are within the range:

$$[\log_{1.0001}(2^{-128}), \log_{1.0001}(2^{128})] = [-887272, 887272]$$

A.2 Calculation of the Liquidity provided

What is going to be of particular interest for the present paper is to understand the procedure to add liquidity to the pool. To do that the provider must specify the tick range in which we want to provide and the amounts of liquidity (amount of two tokens).

From now on, we agree to use ETH as the x reserve and USDC as the y reserve. We will denote a price range by $[P_l, P_u]$ which is delimited by the lower and upper price. Since we compute the prices in terms of square roots, we have to compute those and once we choose the price range, we need to get the current, lower and upper ticks. This procedure is straightforward with Equation (10). We summarize the latter in the following:

1. Calculate the current ($\sqrt{P_c}$), lower ($\sqrt{P_l}$) and upper ($\sqrt{P_u}$) prices.
2. Compute the ticks (i_c, i_l, i_u) using Equation (10).
3. The last step is convert the prices to a Q64.96 number (simply multiply them by 2^{96}).

Now, we need to calculate L specifically for the price range we're going to deposit liquidity into. The previous formula $L = \sqrt{xy}$ holds generally but not in a limited price range. The idea is that we want enough liquidity for the price to reach either of the boundaries of a price range. Thus, we want L to be calculated based on the maximum amounts of Δx and Δy. We give the formulas without further justifications and refer to (1) for the details. We take

$$L = \min\left\{\Delta x \frac{\sqrt{P_c}\sqrt{P_u}}{\sqrt{P_u} - \sqrt{P_c}}; \frac{\Delta y}{\sqrt{P_c} - \sqrt{P_l}}\right\} \times 2^{96}$$

We cannot deposit any amounts at any price range; the liquidity amount needs to be distributed evenly along the curve of the price range we're depositing into. Thus, even though users choose amounts, the contract needs to re-calculate them, and actual amounts will be slightly different (at least because of rounding). We have

$$\Delta x = L\frac{\sqrt{P_u} - \sqrt{P_c}}{\sqrt{P_u}\sqrt{P_c}} \quad \text{and} \quad \Delta y = L(\sqrt{P_c} - \sqrt{P_l}) \tag{11}$$

B Implementation Details

B.1 Data

B.1.1 Uniswap v3 data

We retrieve raw Mint, Burn, Swap, and Collect event emissions from the Uniswap V3 USDC/ETH 0.05% pool between February 12, 2024, and April 6, 2025 using the Alchemy Ethereum archive node. Using web3.py (see Github repository), we filter logs from the pool's contract address for the relevant event signatures. Events are queried by block range and decoded locally. Timestamps are attached by fetching the corresponding block data for each event. We thank Arthur Gervais for running the data extraction scripts and providing access to his Alchemy API key, which enabled us to obtain the complete historical data, including features like ticks (upper, lower, current), transaction amounts, liquidity, tokens' balances, current price, etc.

Uniswap on-chain event data—such as swaps, mints, and burns—is typically stored in CSV format for post-processing and analysis. Each record corresponds to a blockchain event and contains token amounts as raw integers encoded in base units. Specifically, token quantities are represented in their smallest denomination: for ERC-20 tokens like USDC and DAI, this means values are stored in millionths (i.e., with 6 decimals), while for tokens like ETH and WETH, values are expressed in wei, corresponding to 18 decimals. As such, proper scaling is required to recover human-readable values. For example, a value of 10^{18} for a token with 18 decimals represents one full unit (e.g., 1 ETH), while 10^6 represents 1 USDC. Time information is recorded in ISO 8601 format with full precision and UTC timezone. All timestamps are later truncated to the minute to allow alignment with external data sources, such as centralized exchange prices. This preprocessing ensures numerical stability and consistency across datasets used in modeling and training. The price unit (sqrtPriceX96) is also non standard but it is handled in the step function of our environment code. The ETH and USDC scaling are by convention. For liquidity values, a special scaling is needed. The use of 12 decimals follows from the following rigorous justification: For Uniswap V3 pools, the depth can be written as $L^2 = x \cdot y$, which implies that $L = \sqrt{x \cdot y}$. Now, looking at the dimensions we obtain units of liquidity as

$$\sqrt{10^{18} \times 10^6} = 10^{\frac{24}{2}} = 10^{12}$$

which justifies the use of the discounting factor 10^{12} in the preprocessing.

B.1.2 Binance data

We retrieve historical minute-level price data for ETH traded against USDC on Binance using their public rest API. Using web3.py and the official Binance endpoints, we query the exchange info to identify the active ETH/USDC trading pair. We then fetch one year of historical candlestick data at 1-minute resolution, including open, high, low, close prices, volume, and timestamps. We also fetch the same data for perpetual futures which we use for hedging our LP position. Binance does not provide raw trade-level data or sub-minute resolution through its free API tier, so 1-minute intervals are the highest available temporal granularity without paid access.

B.1.3 Replication data

To gather the relevant Uniswap V2 data from the blockchain, the following SQL queries were used:

Swaps

```
SELECT
  date_trunc('minute', evt_block_time) AS minute,
  SUM("amount0In") AS "amount0In",
  SUM("amount1In") AS "amount1In",
  SUM("amount0Out") AS "amount0Out",
  SUM("amount1Out") AS "amount1Out"
FROM uniswap_v2_ethereum."Pair_evt_Swap"
WHERE contract_address = 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc
  AND evt_block_time >= TIMESTAMP '2021-05-05 00:00:00'
  AND evt_block_time <  TIMESTAMP '2022-08-17 00:00:00'
```

```
GROUP BY 1
ORDER BY 1 ASC
```

Mints

```
SELECT
    date_format(evt_block_time, '%Y-%m-%dT%H:%i:%s') AS ts, *
FROM uniswap_v2_ethereum."UniswapV2Pair_evt_Mint"
WHERE contract_address = 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc
  AND evt_block_time >= TIMESTAMP '2021-05-05 00:00:00'
  AND evt_block_time <  TIMESTAMP '2022-08-17 00:00:00'
ORDER BY evt_block_number, evt_index ASC
```

Burns

```
SELECT
    date_format(evt_block_time, '%Y-%m-%dT%H:%i:%s') AS ts, *
FROM uniswap_v2_ethereum."UniswapV2Pair_evt_Burn"
WHERE contract_address = 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc
  AND evt_block_time >= TIMESTAMP '2021-05-05 00:00:00'
  AND evt_block_time <  TIMESTAMP '2022-08-17 00:00:00'
ORDER BY evt_block_number, evt_index ASC
```

Pool reserves

```
WITH ranked_syncs AS (
    SELECT
        date_trunc('minute', evt_block_time) AS minute,
        reserve0,
        reserve1,
        ROW_NUMBER() OVER (
            PARTITION BY date_trunc('minute', evt_block_time)
            ORDER BY evt_block_number DESC, evt_index DESC
        ) AS rank
    FROM uniswap_v2_ethereum."Pair_evt_Sync"
    WHERE contract_address = 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc
        AND evt_block_time >= TIMESTAMP '2021-08-01 00:00:00'
        AND evt_block_time <  TIMESTAMP '2022-09-01 00:00:00'
)

SELECT
    minute,
    reserve0,
    reserve1
FROM ranked_syncs
WHERE rank = 1
ORDER BY minute ASC
```

In a similar manner, we also obtained the Uniswap V3 data for the Deep Reinforcement Learning environment.

B.2 RL Environment

B.2.1 Observation space and market features

We detail here the composition of the observation space outlined in Table 1. At each time step t, the agent receives an observation vector $o_t \in \mathbb{R}^d$ that encodes both its current position and a set of market-derived features relevant for liquidity provision and risk assessment.

In addition to the core state variables including the current pool price $\tilde{p}_t$, the external market price p_t, the liquidity share, and the wealth process, the observation vector includes the following market features:

- The recent *price trend*, measured as the return over a fixed window preceding t, to capture short-term directional signals;
- The *realized volatility*, computed over the same window, to proxy for market uncertainty and concentration risk;
- The *cumulative traded volume* during the previous episode;
- The *average swap and gas fees* incurred in the last episode;
- The current token holdings (x_t, y_t);
- The square-root prices $(\sqrt{P_l}, \sqrt{P_u})$ corresponding to the active liquidity range defined by the lower and upper ticks.

Together, these features provide the agent with a comprehensive view of both its own position and the evolving market context.

B.2.2 Reward function

Gas fees

We calculate gas fees by taking the average of the last 20 gas fees of the same event. This allows us to have a realistic value of the gas fees that may also vary significantly with time depending on the market conditions.

Accrue fees

To calculate the fees earned by the LP, we follow the following procedure (function *_accrue_fees* in the environment code): For each minute, we look at the swap events that happened such that their corresponding tick was within the range where we provide liquidity. Then for each swap, we receive the following fee:

$$AF = 0.05\% \times \text{amount In} \times \frac{L_s}{L_{\text{pool}} + L_s}$$

where L_s is the swaper's liquidity and L_{pool} is the pool's liquidity and amount In is the amount being swaped.

B.3 Training results

Table 2: Performance Comparison of the different Models

Model	**Info 1**	**Info 2**	**Info 3**	**Info 4**
Unhedged	0.00	0.00	0.00	0.00
Hedged	1.00	1.00	1.00	1.00
——	2.00	2.00	2.00	2.00
——	3.00	3.00	3.00	3.00
Invested	4.00	4.00	4.00	4.00