A Reinforcement Learning Approach to Optimal Liquidity Provision in Uniswap v3

Andy Hsu
Electrical and Computer Engineering
University of California, Los Angeles
andyh1469@g.ucla.edu

Akshay Sreekumar
Electrical and Computer Engineering
University of California, Los Angeles
akshay81@g.ucla.edu

***Abstract*—A deep reinforcement learning approach was taken to develop optimal liquidity provision strategies in Uniswap v3, particularly the USDC-ETH pool. We successfully developed a simulation framework for Uniswap v3 using OpenAI Gym, and formulated the liquidity provision problem as a finite state Markov Decision Process. Using this, we trained a deep RL agent using Proximal Policy Optimization (PPO). The agents that we trained in these experiments were not able to achieve performance better than the baseline strategies of holding 100% of starting cash and investing 100% of starting cash into ETH. Most likely, this is due to the immense size of the state space coupled with a lack of adequate training data. Still, this project validates the necessary infrastructure required to perform further exploration into deep RL strategies for optimal liquidity provision.**

I. Introduction

A. The Rise of Decentralized Finance

The past several years has seen significant growth in various cryptocurrencies that are touted as an alternative to fiat currencies. Cryptocurrencies such as Ethereum and Bitcoin are decentralized digital currencies, built on blockchain technology, that are independent of any central banking system. However, despite the currencies themselves being decentralized, the exchanges on which these cryptocurrencies are traded are largely centralized. Exchanges such as Coinbase, Binance, and Robhinhood are all centralized platforms—a central authority (the company itself) exists to oversee trades and match transactions.

In an effort to fully decentralize finance, the notion of automated market makers (AMMs) have gained popularity as a way to facilitate cryptocurrency exchanges without the need for a central authority. This means that users never forfeit their tokens to a central authority for the purpose of a trade, and that there is no longer the concept of a limit orderbook from which a market maker will match opposing trades. Rather, these agents are able to interact with a *pool*, meaning they directly exchange tokens with a smart contract. Because there is no limit orderbook and no market maker, users must act as liquidity providers (LP's) by contributing tokens to a pool. In theory, there can exist such a pool (smart contract) for any two pairs of tokens one wishes to trade.

B. Uniswap v3

One broad class of AMM's is the constant function market maker (CFMM), to which both Uniswap v2 and Uniswap v3 belong. To understand better the fundamental principles of a CFMM and motivate the introduction of Uniswap v3, we consider first Uniswap v2. Uniswap v2 is a CFMM based AMM for the Ethereum Virtual Machine that allows users to pool liquidity, and enables the exchange of two assets through the underlying CFMM invariant.

Uniswap v2 works as follows. Let us consider the exchange of two assets x and y (this could be two tokens like ETH and USDC, say). These assets can be exchanged for each other through interaction with a pool. The fundamental principle of the CFMM dictates that liquidity is distributed uniformly along the reserves curve according to:

$$x \times y = k \tag{1}$$

where k is some constant. A trade with the pool is governed by two simple principles:

1) If a user wishes to exchange some quantity of token x for token y, the amount of token y that the user receives is such that the product of the reserves of both tokens remains constant (k).
2) There is a trading fee associated with each trade with the pool, and the revenue from that fee is distributed proportionally to all the liquidity providers (LPs) of the pool.

The inherent assumption in Uniswap v2 is that liquidity providers are allocating their capital across the entire range of prices. Thus, their capital is used for all trades across all price movements. This is not only capital inefficient, but also dilutes the value of trading fees that an LP can receive. To see why, we consider an illustrative example from Fritsch [1].

Consider a pool of two stablecoins whose price ratio is known to always be within the tight range of [0.99, 1.01]. Imagine that when their ratio is 1, we deposit 1000 tokens of each into the pool. According to Eq. I-B, we have:

$$x \times y = k = 1000^2 = 10^6 \tag{2}$$

If the price movement is such that $\frac{x}{y} = 0.99$, then by the CFMM equation we will have 995 tokens of x, and 1005

tokens of y. Conversely, if the price movement is such that $\frac{x}{y} = 1.01$, then we will have 1005 tokens of x, and 995 tokens of y. Note that in both cases, we only ever needed 5 tokens of each asset to cover price movements across this range of price ratios, but we have 995 tokens that we never even touch. If we could only allocate 5 tokens of each to this price range, we could allocate significantly less capital while still getting paid fees equivalent to as if we had allocated 1000 of each across the entire price range. This notion motivates the introduction of *concentrated liquidity*, which is the chief innovation in Uniswap v3.

In Uniswap v2, all LP's must provide liquidity over the entire price range $[0, \infty]$. In Uniswap v3, concentrated liquidity provision allows LP's to allocate liquidity to one or more specific price ranges, $[p_a, p_b]$. As evidenced by the example above, this is far more capital efficient and effectively functions as a form of leverage—a small amount of capital can be used to achieve higher fees by concentrating the price range over which the capital is allocated. In practice, this means that each position maintains only enough reserves to support trades within its price range, allowing it to act as if it were a constant product pool with larger (virtual) reserves within that range [2]. This relationship between "real" reserves and "virtual" reserves can be seen in Figure 1 below. A position is only solvent within its specified price range.

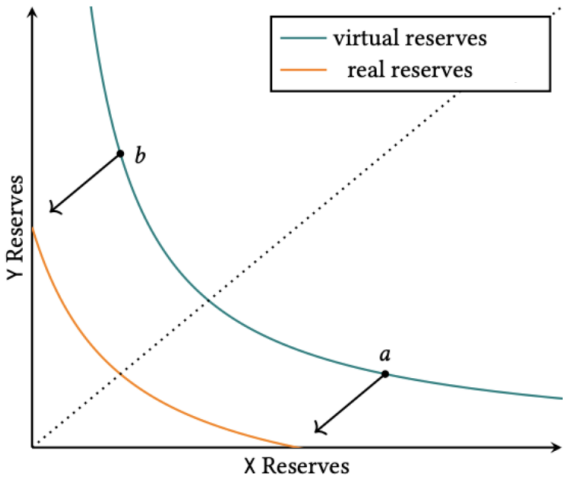

Figure 1: Virtual vs. Real Reserves in Concentrated Liquidity Provision [2]

II. Problem Formulation

We model the problem of optimal liquidity allocation as a Markov Decision Process (MDP). Concentrated LP opens up the possibility of various different types of strategies that enable a liquidity provider to make money. The profit made by an LP depends on two factors: (i) the value of their actual invested position, and (ii) the trading fees generated from their investment. The value of the position depends entirely on the price of the tokens in the pool—if one supplies ETH to a USDC-ETH pool and the price of ETH drops sharply, then clearly the value of the position has also deteriorated. Of course, this loss is "impermanent" in the sense that the losses are not realized unless the LP withdraws their money. If the LP provides liquidity into a suitable range where there is a non-zero volume of trades occurring, then in addition to the inherent value of their position they make fees proportional to the amount of liquidity they are providing. We assume that there are no fees associated with allocation, and that the value of any trading fees generated is withdrawn immediately to cash instead of being reinvested into the LP's position.

For practical consideration, we consider liquidity provision in the 0.3% ETH/USDC pool that allows users to swap ETH for USDC (and vice versa), with the fee rate being 0.3% of the proportional trade volume. The Uniswap v3 price space is discretized into *ticks*, which allow LP's to invest their capital into discrete price ranges.

Every pool defines a set of *active ticks*, which determine the price granularity of a particular pool. For the 0.3% ETH/USDC pool, this spacing of active ticks is every 60 ticks (i.e. this is the smallest distance between valid prices in the pool).

We now formulate the problem of liquidity provision as a discrete time, finite state MDP. Due to limitations on the availability of various market data, we set the timestep of the problem to be one hour. Note that at every given timestep, an LP can choose one of 3 actions:

1) Allocate (i.e. buy)
2) Withdraw (i.e. sell)
3) Do nothing (i.e. hold)

Furthermore, each of these actions can be taken on any particular price range $[p_a, p_b]$ for any arbitrary price that satisfies an LP's budgetary constraints. For example, assume that there are N tick ranges that divide up the price space of interest, the LP starts with $\$X$, and the pool's ETH price is p. The LP can then choose to allocate any amount worth $\leq \$X$ in any combination of price ranges $[p_{a1}, p_{b1}], ...[p_{ak}, p_{bk}]$ that are part of the N valid tick ranges. While this very general formulation allows us to capture the complexity of the Uniswap v3 trading environment, it gives rise to an infinite dimensional continuous action space. To simplify the problem while capturing the essence of the structure, we discretize the action space as follows.

First, we assume that the LP can only allocate/withdraw liquidity to/from one valid tick i per hour. Secondly, the LP selects an integer from $[1, 100]$. If the action is a buy, then this integer represents the percentage of their *current cash reserve* $\$X$ that they choose to allocate into tick i. If the action is a sell, then this represents the percentage of *the value of tick* i that they choose to withdraw back into cash. Note that in both these cases, we consider *position value*—the actual amounts could either be USDC, ETH, or both depending on the pool price p. In the case of holding, nothing changes in the allocation distribution. Thus, an action can be described by the tuple (a, i, k), where $a \in \{0, 1, 2\}$ denotes whether agent will buy/sell/hold, $i \in \{0, 1, ..., N-1\}$ denotes which is the tick of interest, and $k \in \{, 1, ..., 100\}$ denotes what percentage

the LP will trade. Thus, there are $3 \times N \times 100$ discrete actions that the LP can take.

We now describe the discrete state space of the problem. The position of an LP can be described by specifying for each of the N tick ranges how much of each token is in that tick range. Thus, we have $2N$ variables to describe the distribution of both ETH and USDC along the price range. Additionally, we have an extra variable to capture how much cash (i.e. money not invested in Uniswap) the LP has. In addition to these $2N + 1$ variables, the LP keeps track of: the last 24 hours of ETH pricing, the trading volume V in each of the N tick ranges, and the fraction of liquidity ρ provided by the LP in each of the N tick ranges. These additional variables allow us to compute fees given the current state of an LP's position by baking certain market conditions into the state space. Thus, our state space has dimension $4N + 1 + 24$. An example of the state x at timestep t is shown below:

$$x_t = \begin{bmatrix} Token0_{tickrange1} \\ \vdots \\ Token0_{tickrangeN-1} \\ Token1_{tickrange1} \\ \vdots \\ Token1_{tickrangeN-1} \\ Cash \\ Price_{t-23} \\ \vdots \\ Price_t \\ Volume_{tickrange1} \\ \vdots \\ Volume_{tickrangeN-1} \\ \rho_{tickrange1} \\ \vdots \\ \rho_{tickrangeN-1} \end{bmatrix} \tag{3}$$

We now finally define the reward structure of the MDP environment. Our training is episodic, which means that at the end of the episode the agent is given a reward that is equal to the difference in total profit it made vs. a baseline strategy of doing nothing (i.e. just keeping the initial starting cash as cash). For all intermediate steps, the agent receives 0 reward. This is because we want the agent to not be beholden to price variation in the market, but learn the strategy that maximizes *overall* profit at the end of an episode. More formally, let p_T denote the price at the end of an episode, $Token1$ denote the total number of tokens of Token 1 in the position at the end of the episode, $Token0$ denote the total number of tokens of Token 0 in the position at the end of the episode, C_T denote the total amount of uninvested cash in the position at the end of an episode, λ denote a scalar weighting parameter, F denote the total amount of fees collected over the episode, and C_0 denote the initial cash the agent starts with. The reward r at the end of the epsiode is given by:

$$r = (p_T * Token0 + Token1 + C_T + \lambda * F) - C_0 \tag{4}$$

III. Methods

A. Uniswap Simulation Environment

We implemented the Uniswap v3 MDP described above using the OpenAI Gym library. This Gym environment is initialized through the following steps:

1) Define a min/max price, where price is the exchange ratio between the two tokens of the liquidity pool.
2) Define tick spacing of the pool. The 0.3% ETH/USDC pool that was studied in these experiments groups individual ticks into batches of 60. These are called the "active ticks" of the pool.
3) Define the fee weighting λ and the starting cash of the agent.
4) Set the initial agent positions in each tick to 0.
5) Randomly select an hour from the available data to begin the first episode.

Algorithm 1 Uniswap v3 Gym Environment

```
procedure RESET(TokenPositions, Cash, Timestep)
    TokenPositions ← 0
    Cash ← 20000
    Timestep ← random(marketData)
    r ← 0
end procedure
procedure STEP
    if Timestep == end(marketData) then ▷ Episode End
        r ← computeReward()
    else
        while Timestep ≠ end(marketData) do
            takeAction()
            adjustTokenPositions()
            advanceStepCount()
            getNextObservation()
        end while
    end if
end procedure
```

During the reset, the agent starts off with \$20000 in cash, and no money invested in Uniswap v3. A random hour is selected from the available data to begin the first episode.

At each step, the agent can either buy, sell, or hold the position within a single active tick. An action is selected from the discrete space described in Section I-B. The new token positions for the tick are computed using the Uniswap v3 concentrated liquidity equations [3]. These ensure that the rules of Uniswap v3 are respected—the number and type of tokens within each tick should be correctly determined in accordance with the current pool price as dictated by the rules of Uniswap v3. The equations that describe token amounts per tick depending on the price are given below. Let Token 0 be denoted as x, Token 1 be denoted as y, the current price be

P, and the liquidity be given by L. Now, considering a price range $[p_a, p_b]$, we have the following three cases:

$$\begin{cases} x = L\frac{\sqrt{p_b}-\sqrt{p_a}}{\sqrt{p_a p_b}}, y = 0 & \text{if } P \leq P_a \\ x = 0, y = L(\sqrt{p_b} - \sqrt{p_a}) & \text{if } P \geq P_b \\ x = L\frac{\sqrt{p_b}-\sqrt{P}}{\sqrt{P}\sqrt{p_b}}, y = L(\sqrt{P} - \sqrt{p_a}) & \text{if } p_a < P < p_b \end{cases} \quad (5)$$

After taking the action, the next hour of market data is retrieved including the next price. The token positions within each active tick are updated to reflect the new current price. Finally, the reward for the step is calculated and the new observation vector is created. If it is the end of the episode, the reward is computed using Equation II.

B. Data Pipeline

The Uniswap Gym environment requires the following data to simulate episodes:

1) The price (exchange ratio of the two tokens) at each hour
2) The volume of token swaps within each tick at each hour
3) The total liquidity within each tick at each hour

The first two items were obtained by making queries to the Uniswap v3 Subgraph, an API that is used to retrieve historical information on liquidity pools [4]. At the time of these experiments, the Subgraph query for retrieving liquidity information was not working correctly. Thus, the third item was obtained by parsing through all mint/burn/swap events from the raw blockchain data of the pool and manually calculating the liquidity at each hour.

C. RL Algorithm

There are several considerations in selecting an appropriate algorithm for training our RL agent. The formulation of our problem as an MDP with a discrete action space suggests that potentially learning a deep Q-network (DQN) might be an appropriate choice as they are meant to work only with discrete action spaces. Additionally, DQN could help compensate for our limited data by leveraging a replay buffer to make use of past experiences in an off-policy manner.

However, a fundamental feature of our problem is the episodic reward structure which leads to delayed feedback. This is essentially the credit assignment problem—with sparse rewards, how can we know which actions were responsible for a potentially large reward at the end of an episode? A DQN based approach would struggle in this regard. The Q function for the majority of state-action pairs along a trajectory in an episode will not get updated frequently as all intermediate rewards are set to 0. Only towards the end of the trajectory, when we receive an actual non-zero reward, will there be a significant update of the Q function. However, due to the nature of the Q-learning algorithm, it will take time for this update to propagate backwards to earlier state-action pairs. Thus, the DQN is not necessarily ideal for dealing with our sparse reward structure.

To compensate for this, we instead consider the class of actor-critic methods using the policy gradient. We make use of both a policy network and a value network, and we optimize the policy directly using the gradient. The introduction of the advantage function allows us, after the conclusion of an episode, to determine how advantageous our actions were along a trajectory relative to some baseline estimate. While the updates of the advantages via the updating of the value network may suffer from the same slow updates as DQN, the key here is that we can directly optimize the policy in a direction that takes into account the advantages along the entire trajectory. This helps to deal with the credit assignment problem by providing a learning signal even when the feedback via the reward is delayed. Of the policy gradient methods, we choose to use Proximal Policy Optimization (PPO) [5]. PPO helps guarantee a trust region such that our policy does not change too drastically in between steps—this is important in making sure the learned trading strategy is robust and stable. However, as an on-policy algorithm PPO tends to be more sample inefficient which is problematic for our limited data environment in a very high dimensional state space.

IV. Results

We trained multiple agents in the Uniswap environment and performed a grid search across the following hyperparameters:

1) Number of nodes in the hidden layer of the policy/value networks: {2048, 4096}
2) Learning rate for training: {0.03, 0.0003}
3) Number of steps taken before updating network parameters: {256, 2048}
4) Weight of the fee contribution to the reward: {0, 10, 100}

The results are shown in Figure 2. For each parameter set, we compared the average reward across 40 episodes of training with two baselines:

1) The starting cash if held until the end of the episode
2) The average position value if the starting cash had been 100% invested into ETH and held until the end of the episode

V. Discussion

Based on the results, the dynamics of the Uniswap environment appear to be working as expected. However, the low average reward across all parameters of the grid search indicates that the agent had difficulty learning a policy that could do better than the baselines. We believe this is could be due to the following reasons:

1) Not enough training data. Uniswap v3 has been running for just over 2 years at the time of writing with only 15k hourly data points available. RL problems (especially those with an inherently continuous state space) often require millions of samples for an agent to learn an effective policy. As Uniswap continues to operate, more real-world data will become available for use in the future. Additionally, our choice of RL algorithm was

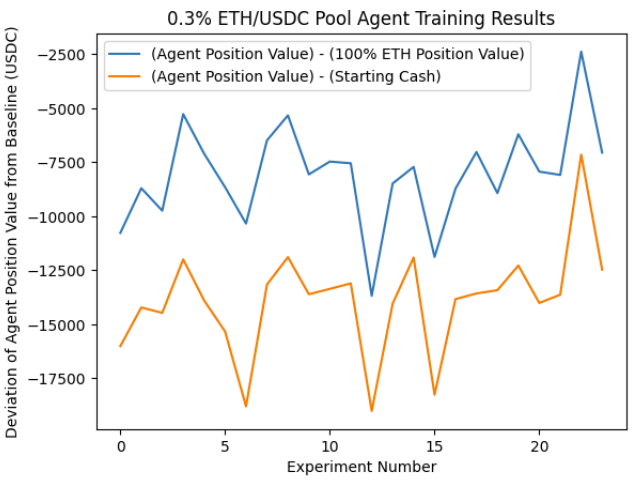

Figure 2: Plot of the agent's performance relative to the 2 baselines, across all parameter combinations of the grid search. Note that the starting cash of the agent is 20,000 USDC.

not the most sample efficient, but was chosen for other reasons related to the structure of our problem.

2) Insufficient pre-processing of market data. The agent likely has trouble extracting meaningful insights from the raw price history/liquidity/volume numbers. Future work could potentially consist of defining a pre-processing pipeline (time series interpretation/price forecasting) for this market data in order to feed more valuable features directly to the agent.
3) Sub-optimal policy network architecture. Since the data is inherently sequential, using fully-connected layers may overly complicate the loss landscape and make training more difficult. A deep learning architecture that is more suited for series data (ex. adding LSTM layers, attention layers) might make it easier for the agent to learn a good policy.

More broadly, in making optimal decisions under uncertainty as we are in this problem, there are fundamentally three key pieces to building a successful system. The first of these is having some model or simulation of the system, which we were able to successfully develop. This allows us to iterate on different agents, evaluate different strategies, and examine the behavior of the system of interest (Uniswap v3 in this case). To deal with uncertainty, in our case market conditions such as price, volume, and liquidity, it would make sense to develop some kind of forecasting system. The final component is the actual technique/decision making algorithm being used. In our case, using a deep reinforcement learning approach, this technique depends significantly on data availability and quality. Although DQN has theoretical concerns related to how quickly updates to action-values affect the learned policy, it may still be helpful to run experiments and review results using this algorithm given the low rewards when using PPO.

VI. Conclusion

In this work, we formulated investment in a Uniswap v3 liquidity pool as an MDP and successfully created a simulation of the Uniswap environment using OpenAI Gym. Although the results of the agents we trained did not exceed the defined baselines, we believe there is still potential in this framework and have outlined several steps that can be taken to continue exploring this application.

References

[1] Robin Fritsch. *Concentrated Liquidity in Automated Market Makers*. Oct. 4, 2021. DOI: 10.48550/arXiv.2110.01368. arXiv: 2110.01368[q-fin]. URL: http://arxiv.org/abs/2110.01368 (visited on 05/06/2023).

[2] Hayden Adams et al. *Uniswap v3 Core*. Uniswap, Mar. 2021. URL: https://uniswap.org/whitepaper-v3.pdf.

[3] Atis Elsts. *Liquidity Math in Uniswap v3*. Sept. 30, 2021. URL: https://atiselsts.github.io/pdfs/uniswap-v3-liquidity-math.pdf.

[4] *Uniswap V3 Subgraph*. URL: https://thegraph.com/hosted-service/subgraph/uniswap/uniswap-v3.

[5] John Schulman et al. *Proximal Policy Optimization Algorithms*. Aug. 28, 2017. DOI: 10.48550/arXiv.1707.06347. arXiv: 1707.06347[cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 05/21/2023).

Acknowledgment

We would like to thank Professor Lin Yang for introducing us to this problem space, as well as his mentorship/guidance throughout the project.