Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Reinforcement Learning for Blockchain-Based Trading

Semester Thesis

Yves Zumbach

yzumbach@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Ye Wang, Zhongnan Qu
Prof. Dr. Roger Wattenhofer

October 6, 2021

Abstract

This document summarizes the efforts conducted to implement a reinforcement learning environment and train an agent that learns to trade on the Ethereum blockchain, more specifically with the Uniswap smart contract. The goal is to be able to learn arbitrage transactions and maybe even more complex strategies like sandwich attacks.

Contents

Chapter 1

Introduction

1.1 Blockchains and Uniswap

Since 2020, the world of cryptocurrencies has attracted a lot of attention and the amounts of money invested in these systems has increased exponentially. The large sums featured in this ecosystem makes it a very relevant topic to investigate. Some researches, like the discovery of "Miner Extractable Value" (MEV) [1], have had deep impacts over the blockchain world, in this case it created a whole new research field related to miner attacks and the game theoretical aspects of transaction mining.

Aiming at creating a truly decentralized economy, the blockchain gave birth to the first Automated Market Makers (AMM). Those are exchange that are powered, not by buyers and sellers, but by liquidity pools. Uniswap was the first to introduce the fixed product formula [15] and rose to be the most important AMM if the amounts in the liquidity pools or the total transaction amounts are taken as metrics.

Exchanges can provide "arbitrage opportunities". An arbitrage opportunity arises when it is possible to buy an asset in one exchange cheaper than it is possible to sell it in some other. By buying in the first exchange and selling in the second, you are guaranteed to make a profit. Uniswap is an exchange like any other and can provide such opportunities if the price on Uniswap differ from those of some other exchanges. In this article, we will refer to the prices of the other exchanges as the "market price".

Being a smart contract on the Ethereum blockchain, all Uniswap pairs will also inherit the limitations of the Ethereum blockchain. One of them is that the transactions are first submitted to the mempool where all the miners can see them, and only then are they mined into a block, i.e. executed. Miners are not required to mine transactions in the order they were submitted. Instead they generally mine first the transactions that bring them the highest fees, i.e. the transactions with the highest gas price. This makes it possible to frontrun transactions, i.e. execute a transaction that was submitted after another one before it by setting a

higher gas price.

This gave rise to the so called sandwich attack that allows some attacker to extract value from a large Uniswap transaction by submitting two transactions: one that frontruns and another that postruns the large transaction. Appendix A explains the details of sandwich attacks.

1.2 Reinforcement Learning

Reinforcement learning is a specific type of unsupervised machine learning that consists in training a program, called *agent*, to interact with some *environment* and maximize some *reward* computed from the environment.

1.3 Goals

The goal of this semester project is to create a reinforcement learning pipeline able to train a reinforcement learning agent that will trade with the Uniswap smart contract using strategies to increase its wealth. This can include performing arbitrage transactions, sandwich attacks or some other kinds of guaranteed-benefits transactions strategies that we might not yet know of. The agent wealth will be computed in US dollar.

In this project, we will focus on the ETH/USDC Uniswap pair only. ETH is the currency symbol of Ether, the native currency of the Ethereum blockchain. USDC is the symbol of the USD Coin cryptocurrency which is a stablecoin pegged to the value of the US dollar (USD), i.e. it is a cryptocurrency that tries to have a value as close as possible to that of the US dollar. The ETH/USDC pair is one of the most popular, as such there is a lot of data available for it.

Chapter 2

Code Architecture

This project's structure follows the PyBuilder structure as PyBuilder [10] is the build automaton configured with this project. All the sources are to be found in the `src/` folder. They are sorted by source set, of which they are two: `main` and `unittest`. We will only concern ourselves with the `main` sourceset, the other contains only unit tests to check the correct implementation of the first. This sourceset is further divided in two important folders: `python/`, that contains a python package called `rlmbc` (Reinforcement Learning Meets BlockChain) listing all the classes and functions required by this project, and `scripts/`, that contains the executable files that make use of the functionalities provided by the `rlmbc` package.

We will focus on the `rlmbc` package in this section and describe its architecture. Some lower details explanations about the important classes of the package will be given in section 3. The outputs generated by the scripts are detailed in section 6.

2.1 Blockchain Simulation

The folder `blockchain/` contains the code required to simulate an Ethereum blockchain.

This includes a `Network` class which is the core of the simulated Ethereum blockchain. This class coordinates the whole simulation, it keeps track of the ERC-20 tokens [3] deployed, the miners, the regular accounts, the latest transaction nonce for each account and it decides which miner gets to mine the next block.

The `miner` class simulates the behavior of miners. It contains an algorithm that orders the transactions to mine in a way that should greedily maximize the benefits of the miner. This algorithm is a simplification of the ones used in real life, but it is probably sufficient for this project. The details of this algorithm are given in section 3.1. The importance of this algorithm for the simulation and the current limitations are highlighted in section 5.1.

The `cryptocurrency` class provides a simple implementation of ERC-20 tokens.

The `Transaction` class represents all the transactions that are simulated. It matches quite closely the real Ethereum transactions. Note that Ethereum transactions can be simple ETH transfers but they can also trigger smart contracts. Triggering smart contracts was implemented in Python by storing a function to execute as one of the transaction attributes. This simple hack made the implementation much simpler, although it provides less guarantees, i.e. the developer needs to make sure that it only stores functions that trigger simulated smart contracts and nothing else.

The `Uniswap` file provides an implementation of a Uniswap pair, as well as subclasses of `Transaction` specifically targeted at interacting with the Uniswap pair, i.e. subclasses for the three possible interactions with Uniswap: minting, burning and swaps. Details of the formulas and of the implementation are given in section 3.2.

The remaining classes in the `blockchain` package are more straightforward and are not described in more details. This includes classes to represent blockchain accounts and mined blocks.

2.2 History

The `history` packages contains the code to simulate prices based on historical values.

The `HistoricalDailyStock` class computes daily returns, average daily drift, daily volatility and squared daily volatility of some provided stock data. Some helper functions allow:

- Fetching data from Yahoo Finance [4] given the symbol of the stock or currency whose data should be fetched.
- Loading data from disk.
- Storing data to disk.

All helper functions cache the data for efficiency reasons.

The `HistoricalDailyStock` class expects the sampling rate of the provided data to be daily. This is the default sampling rate of data from Yahoo finance. For blockchain purposes, this is a somewhat low sampling frequency as volatility is quite high, nevertheless Yahoo Finance has data on many stocks or currencies, including cryptocurrencies, the datasets are pretty extensive, fast to download, and totally free which made it a reasonable fit for this project.

The `Oracle` class provides an easy-to-use interface to sample new prices based on historical values. On the Ethereum blockchain, blocks are mined approximately every 13 seconds. This means that the various statistics of the historical data provided by Yahoo Finance like the drift and volatility, which are based on daily time interval, need to be scaled to much smaller time scales for the oracles' needs.

There are two implementations of price oracles, i.e. two classes that generates price evolutions. The first is a geometric brownian motions [5]. Using geometric brownian motions to simulate prices is the most widely used approach in practice and is also used in the Black-Scholes model. The second implementation differs in that it does not use a Wiener process [16] (also called Brownian motion sometimes) to generate the increments as the first one. The Wiener process is a Lévy process [8] that has Gaussian increments. This process is easier to implement, but features less extreme events which we deemed also less realistic. The second implementation uses a Lévy process with increments drawn from the Student-t distribution which features more extreme events. The details of the oracle implementations are provided in section 3.3 and 3.4.

2.3 Probabilities

The `DiscreteProbabilityDistribution` class provides a straightforward implementation of discrete probabilities which is used in many places of the project.

2.4 Reinforcement Learning

This package provides all the functions and classes required to perform some reinforcement learning.

Inside the `agent` subpackage are classes that implements the abstract interface `BlockchainAgent` which provides a simple and unified way for agent to be queried. This interface is only use at test time, not during the training. Along a wrapper for `rllib`-trained agents, a few baseline classes are also provided:

`IdleBaseline` A baseline to compare the performances of a trained agent that does nothing with the money it is given.

`MintBaseline` A baseline to compare the performances of a trained agent that mints Uniswap shares at the beginning, then only waits and capitalizes on the fees generated by others that use the Uniswap pair the agent is a liquidity provider of.

The `data` subpackage contains hardcoded values and data needed to perform the simulation like some smart contracts' addresses, the path to CSV data files

containing Uniswap transactions, filtered datasets containing the ETH/USDC Uniswap pair data, etc.

The `generator` subpackage contains generators for various purposes:

- Generating account with randomly initialized amounts of different currencies. Some more details on this generator are given in section 3.5. The importance of this generator is further highlighted in section 5.3.
- Generating various types of Uniswap transaction (mint, burn, swap), including arbitrage transactions. Some more details on this generator are given in section 3.6. The importance of this generator is further highlighted in section 5.3.

The `environment` package contains the `BlockchainEnv` class with is the core of the reinforcement learning part of this project. This class defines the action space and the observation space, and setups and manages the simulation. More explanations about this class are given in section 4.1.

The `reward` file contains helper functions to compute the agent rewards as well as the agent total wealth. A brief explanation of this computation is given in section 3.7.

Chapter 3

Algorithms, Formulas and Implementations

This section focuses on the key classes and algorithms in this project. It describes them and also explain the rationale behind these choices.

3.1 `Miner`

In real-life, miners can decide:

- Which transactions they want to include in the block they are mining.
- In which order they list the transactions.

They must, however, always order the transaction of a single account in the order they were produced, i.e. in increasing `nonce` order[1]. We remark that it is possible for a single account to submit multiple transactions with the same nonce. This allows an account to potentially override a previously sent but unmined transaction.

Miners earn money in one of three ways in real-life:

1. They earn a reward of a fixed number of ETH for mining a block.
2. They earn gas fees. Ethereum transactions are composed of instructions executed by the Ethereum Virtual Machine. Executing instructions on the EVM has a cost specified in *gas*. When transactions are submitted, they specify a gas price which is the maximum gas price that the sender is ready to pay to get the instructions in their transaction executed. The miner will earn the number of gas units consumed by the transaction times the

[1]The nonce of a transaction must always be equal to the number of transactions that the sending account has already sent, i.e. it increases by one for every new transaction.

maximum gas price specified in ETH that the user is ready to pay—or less if the miner decides to bill less than the maximum price, but this does not happen often in practice.

3. They can earn money by including their own transactions, e.g. arbitrage transaction or specific types of attacks, as they have oracle-like knowledge of the transactions that will be executed in the block.

As we are not interested in learning optimal miner behavior in this project, the first and third sources of revenue are of no concern to us. We focus on the second instead. Note that miners have a limit on the units of gas that they can consume in a single block. Therefore, they have an incentive to include only the transactions with the highest gas price to maximize their benefits.

3.1.1 Transaction Ordering

The miner implementation in this project includes as many transactions as possible until reaching the block gas unit limit which is set to 1'000'000[2]. Because we only consider Uniswap transactions, it is rather unlikely that this limit will ever be reached. Instead a simpler mechanism is also provided: the miner is provided a limit on the number of transactions that it can include in the block. This limit can be set by the developer and makes it possible to include a number of transaction that is realist given the specifics of the simulation at hand. For example, the `BlockchainEnv` class samples a count of transaction from the historical distribution of Uniswap transactions included per block so as to include a realistic number of Uniswap transaction in each block.

When ordering transactions, the miners take into account that transactions of a single user must always be ordered. They also support transaction overriding and will only ever include one transaction for a given nonce. The gas cost of a transaction is determined by the miner too by sampling a value from a Gaussian distribution specific to the type of Uniswap transaction being conducted, i.e. mint, burn or swap, where the mean and standard deviation were computed from historical values.

A pseudocode version of the miner algorithm is given in Algorithm 1.

The implementation of the transaction ordering algorithm has runtime complexity of $O(n^2)$ which was a concern as the algorithm is executed for every block, i.e. every 26 steps of the simulation on average as blocks include 26 Uniswap transactions on average. Being able to run the simulated environment very fast is of major importance to be able to learn rapidly enough when doing reinforcement learning. Empirically, it turned out to be reasonable, the main bottleneck was

[2]On the real Ethereum blockchain, this limit has had values between 3 mio in 2016 and 30 mio in 2021.

```
Data: transactions: All executable transactions available sorted by
      decreasing gas price
Result: A mined block
i ← 0;
mined transactions = {};
while i ≤ transactions.length do
    if current transaction is not the next transaction for the transaction
     sender then
        i++;
        continue;
    end
    if gas already used in block + gas used by transaction > gas available
     for block then
        i++;
        continue;
    end
    fee = gas used by transaction * gas price in ETH;
    if transaction sender has not enough funds for transaction then
        i++;
        continue;
    end
    pay the miner the gas fees;
    execute transaction (might trigger a smart contract);
    gas already used in block += gas used by transaction;
    next valid nonce for sender += 1;
    mined transaction.add(transactions[i]);
    if count of transactions in block ≥ maximum transactions in block
     then
        break
    end
    i = 0;
end
```

Algorithm 1: Miner Transaction Ordering Algorithm

computing the gradient and updating the weights of the neural network underlying the agent, not running the simulation.

We remark that the algorithm implemented is a simplification. The cost in gas unit of each transaction is sampled at random instead of being measured as in the real Ethereum blockchain[3]. Further, the greedy approach of taking the first valid transaction among the highest gas price transactions without taking

[3]Computing the real gas cost of transactions would require to run the transactions through the Ethereum Virtual Machine which would take unreasonable amounts of time and unreasonable amounts of efforts to implement.

the gas consumption into account is a simplification that might not be optimal for the miner. The problem of selecting the transactions in order to maximize the miner's benefit is a variant of the Knapsack Problem [7] which is NP-hard.

3.1.2 Atomic Transactions

Transactions can fail in many ways: the sender might not have enough balance to pay the transaction, or it might set the gas price so high that it does not have enough balance to pay the miner fees, or the smart contract function that it triggers might fail for some reason. Ethereum transaction are atomic, i.e. they are either fully executed or fully reverted, but never half-executed. Providing this guarantee is desired in our simulation to mimic reality closely and to prevent the simulation from unexpected behaviors or even crash because of invalid transactions.

Providing atomicity is notoriously hard. The current implementation does in fact provides transaction atomicity, but it relies on the implementation details of the Uniswap pair which is the only smart contract there is to interact with. Upon executing a transaction, a miner will first transfer the fees and the transaction amount from the transaction sender to the miner and transaction receiver respectively. Then it will execute the smart contracts function, if any. Each function in the Uniswap pair performs first many checks to assert that the transaction can be executed completely. Only once those checks are successful does the smart contract modify its internal state. If any check fails, the smart contract will not change its internal state, but will instead throw an exception which will cause the initial two transfers to be reverted by the miner. If the smart contract was to make any change to its internal state and to throw an exception only afterwards, then those changes could not be reverted as they are not tracked by the miner, i.e. the structure of the code does not guarantee transaction atomicity, only the low-level implementation details of the Uniswap pair does. In other words, it is the smart contract responsibility to make sure either that they fail an incoming transaction before making any changes to their state or to execute the transaction fully. This is something that might need to be improved in the future, especially if the simulation needs to accommodate other smart contracts.

3.2 `Uniswap`

The Uniswap implementation in this simulation follows the Uniswap API [15] quite closely. In the code, the two currencies that the contracts holds are called crypto0 and crypto1. We will keep this naming scheme in the following explanation as well as the following conventions:

- r_0 and r_1 denotes the reserves of crypto0 and crypto1 respectively.

- s will denote the total amount of shares assigned by the Uniswap pair. s_a will denote the amount of shares owned by user a.

When the Uniswap pair is created, the account that creates it specifies the initial amounts a_0 and a_1 of crypto0 and crypto1 respectively that the pair will contain. The creator is given an amount of shares equal to $\sqrt{a_0 \cdot a_1}$. The trading fee is set to 0.30%.

Note that the implementation of the Uniswap class performs extensive testing of all values during the computations so as to fail as early and with as much explanations as possible to make it easy to debug some issues in the code.

3.2.1 Mint Transactions

To perform a mint transaction, the sender only specifies the amount δ_0 of crypto0 that it wants to send to the smart contract. Then the smart contract will computes the corresponding amount δ_1 of crypto1 given the exchange rate of the pair and transfer it from the sender's account to itself.

We remark that this implies that the sender must have enough crypto1 to perform the transaction, otherwise, some exception will be thrown. When training an agent through reinforcement learning however, the programmer does not have such fine-grained control over the actions that are tried, so the action space must be designed in such a way that the agent cannot try to perform an invalid transaction which would otherwise crash the simulation.

We also remark that this is a simplification over the real-world Uniswap implementation which lets liquidity providers provide any amount of crypto1 given a fixed amount of crypto0. However, providing liquidity at any rate other than the market rate provides an arbitrage opportunity. The only rational choice, when providing liquidity is to do it at the market rate, which is the reason why we considered this simplification to be reasonable.

The liquidity provider, called lp hereafter, will be given an amount of shares $\delta_{s_{lp}}$ equal to $\delta_0 \cdot s/r_0$. The total amount of shares s will be additively increased by the same amount.

3.2.2 Burn Transactions

When some sender lp requests to burn $\delta_{s_{lp}}$ shares, it will receive $\delta_{s_{lp}} \cdot r_0/s$ crypto0 and $\delta_{s_{lp}} \cdot r_1/s$ crypto1. Further, the total amount of shares will be additively decreased by $\delta_{s_{lp}}$.

3.2.3 Swap Transactions

The following formula is used to compute the amount of crypto1 that the smart contract will send to a user a that is swapping some crypto0 for some crypto1. Given that a sends δ_0 crypto0 to the contract, it will receive $0.997 \cdot \delta_0 \cdot r_1/(r_0 + 0.997 \cdot \delta_0)$.

The formula to compute the amount of crypto0 that it would receive given some transferred amount of crypto1 is obtained by swapping the 0 and 1 indices in the above formula.

3.3 `GeometricBrownianMotionOracle`

This class implements a geometric brownian motion with Wiener process increments, i.e. increments drawn from the Gaussian distribution. The formula for a geometric Brownian motion [5] is that the stock price S_t of stock S at time t is given by:

$$S_t = S_0 \cdot \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right) \tag{3.1}$$

Where:

- S_0 is the price at the beginning of the simulation.
- μ is the drift of the stock price, i.e. the average of the return percentage sampled at a fixed frequency.
- σ is the stock's volatility, i.e. the standard deviation of the stock return percentage for the same sampling frequency as the drift.
- W_t which is the Wiener process increment.

In this implementation, the sampling frequency was chosen to be 1 day, because data available to us had this sampling frequency. Generating values for the Wiener process was done by sampling from a Gaussian distribution with mean 0 and standard deviation equal to t.

An example of the prices generated by this oracle is given in figure 3.1.

3.4 `StudentGeometricBrownianMotionOracle`

This class provides a geometric Brownian motion with Student-t increments. The stock price follows the same process as the one described in equation 3.1. This

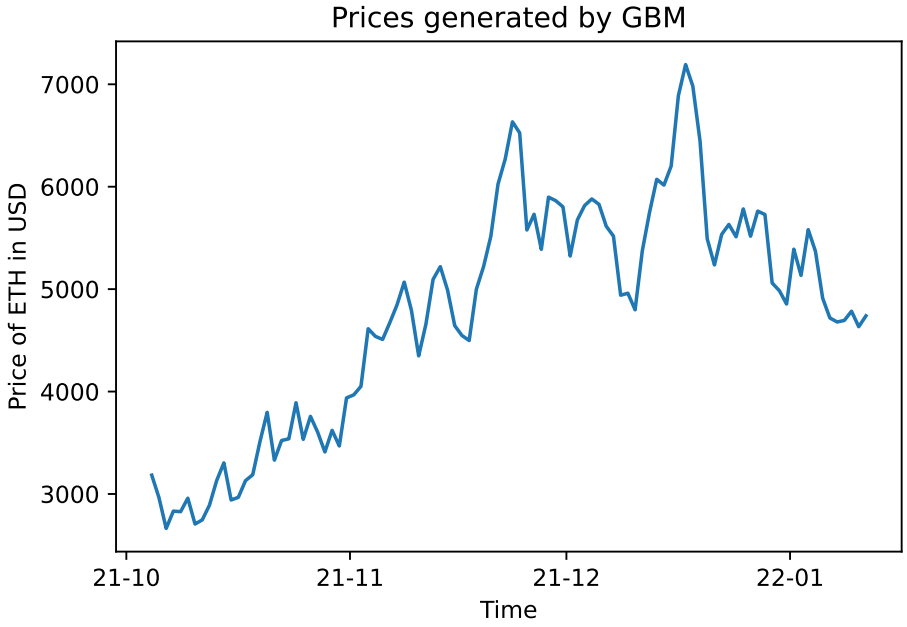


Figure 3.1: Some generated USD prices for ETH by a geometric Brownian motion.

time however, the Wiener process is replaced with a Lévy process with Student-t increments instead of Gaussian increments. We used five degrees of freedom, $\nu = 5$, as is standard in finance. When sampling from this distribution, the variance of the output will be equal to $\frac{\nu}{\nu-2}$ by property [14]. We need however the increments to have a variance equal to t, so we scale down the increments as follow.

$$X \sim \text{Student}(5)$$
$$\text{increment} = X \cdot \sqrt{\frac{(\nu-2)\cdot t}{\nu}}$$

We remark that

$$var\left(X \cdot \sqrt{\frac{(\nu-2)\cdot t}{\nu}}\right) = \frac{(\nu-2)\cdot t}{\nu} \cdot var(X)$$
$$= \frac{(\nu-2)\cdot t}{\nu} \cdot \frac{\nu}{\nu-2} = t$$

as desired.

An example of the prices generated by this oracle is given in figure 3.2.

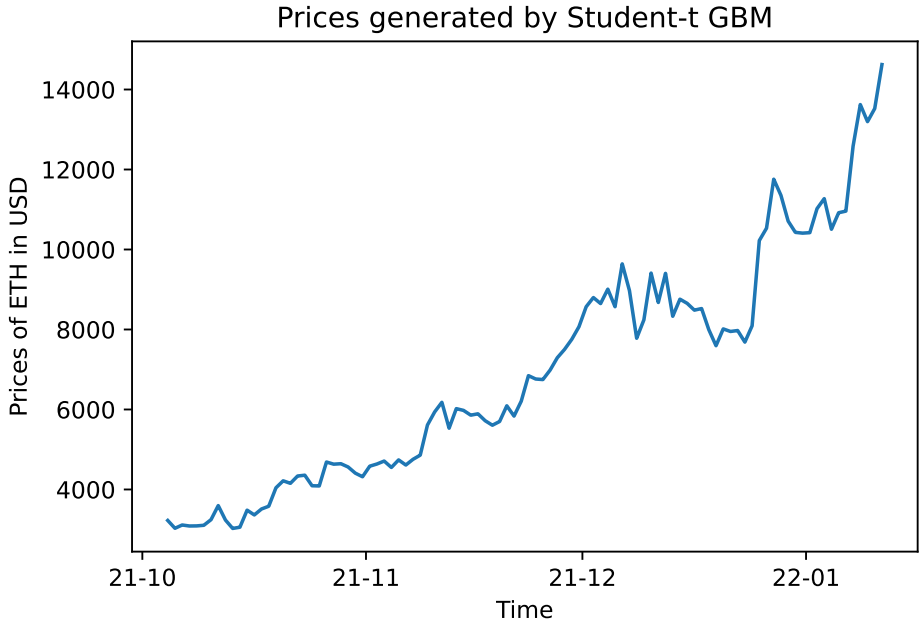


Figure 3.2: Some generated USD prices for ETH by a geometric Brownian motion based on Student increments.

3.5 `AccountGenerator`

The `AccountGenerator` class is used to give some initial balance to simulated accounts. Currently, the algorithm simply draws an initial balance uniformly at random between 1'000 and 1'000'000 for all the currencies the account should own.

3.6 `TransactionGenerator`

The `TransactionGenerator`'s goal is to generate Uniswap transactions. It can generate any kind of Uniswap transactions: mint, burn and swap. It can further generate arbitrage swap transactions. Finally, it offers methods to generate Uniswap transactions where the specific type of transaction is chosen at random.

To choose the type of transaction to generate, the `TransactionGenerator` samples from the real-life distribution of type of Uniswap transactions.

When generating transactions, the transaction amount is chose uniformly at random between 0.1% and 90% of the sender's balance. The reason for choosing these values is pretty much arbitrary but it guarantees that the transaction amount's will be rather diverse, and more importantly that no account will ever

end up with a zero balance. This in turns guarantees that any account will always be able to generate any kind of transaction.

When generating burn transactions, only accounts owning shares for the Uniswap pair are considered and the "sender's balance" is actually the amount of Uniswap shares it owns. When generating mint transactions, the generator has to be cautious about not generating mint transactions with an amount of crypto0 that corresponds to an amount of crypto1 too large for the sender to be able to pay or vice-versa. For this reason, the generator first computes the ratio between the cryptos of the Uniswap pair owned by the sender, then computes the ratio of crypto owned by the Uniswap contract and then defines the "limiting crypto" for the sender based on those two ratios.

Assume s_0 and s_1 are the amounts in crypto0 resp. crypto1 owned by the sender and r_0 and r_1 are the reserves of crypto0 resp. crypto1 of the Uniswap pair. The sender's ratio is $\alpha = s_0/s_1$. The Uniswap pair's ratio is $\beta = r_0/r_1$. If $\alpha \leq \beta$, then the limiting currency for the sender is crypto0 and vice-versa. In other words, we know for sure that even if the sender was to mint 100% of its crypto0, it could still pay the required amount of crypto1 to the Uniswap pair (see section 3.2 for the details on why it will need to pay some crypto1). The transaction amount is set to a uniformly random value between 0.1% and 90% of the limiting currency so as to guarantee that the transaction can always be executed successfully.

When generating swap transaction, the `TransactionGenerator` might decide to generate arbitrage transactions. The generator will first check whether there is an arbitrage opportunity. Whenever the price of the Uniswap pair deviates from the market price, there might be an arbitrage opportunity, i.e. a transaction that will swap some crypto0 (or crypto1) for some crypto1 (or crypto0) that can then be sold on the market for more crypto0 (or crypto1) than the original amount sent. The optimal transaction is the one that bring the price on Uniswap back to the market price. Uniswap applies a 0.30% fee which needs to be taken into account to determine if there is an arbitrage opportunity. If the benefits of the arbitrage are not enough to cover the fees, then this is equivalent to saying there is no arbitrage opportunity.

The optimal amount of token 0 to swap for some token 1 in an arbitrage transaction can be computed as follow [2].

$$\gamma = 1 - \text{trading fee}$$
$$k = r_0 \cdot r_1$$
$$\delta_1 = r_1 - \sqrt{\frac{k}{\gamma \cdot m_p}}$$
$$\delta_0 = \frac{1}{\gamma} \cdot \left(\frac{k}{r_1 - \delta_1} - r_0 \right)$$

Where:

- r_0 and r_1 are the reserves of crypto0 and crypto1 in the Uniswap pair.
- m_p is the market price rate given in crypto0 per crypto1

If δ_0 is negative, then there is not arbitrage opportunity that yields positive return when selling some crypto0. It can be checked whether there is an arbitrage opportunity in the other direction by swapping the 0 and 1 indices in the above formula. It can be the case, because of the trading fee of Uniswap, that there is no arbitrage opportunity at all, i.e. both δ_0 and δ_1 are negative values. This happens when the price of Uniswap deviates too little from the market price compared to the trading fee for anyone to be able to make a benefit. To check the presence of an arbitrage opportunity, one of the δ needs to be strictly positive.

Once the `TransactionGenerator` has verified that there is an arbitrage opportunity, it will decide at random whether it produces an arbitrage transaction, i.e. a transaction that will bring the price ratio of the Uniswap pair back to the market price, or if it just generates a random swap transaction. The proportion of arbitrage was defined arbitrarily to be 1/5. We will expand in section 5.3 on why this value would need to be fine-tuned further than it currently is.

3.7 Agent Total Wealth

This helper function makes it easier to compute an account's total wealth in USD. We decided to define the wealth of an agent as the equivalent value in USD of the assets the agents owns. This simplifies many aspects of the simulation, indeed the agent can perform an arbitrage transaction simply by buying an asset like ETH on Uniswap at a cheaper price than the market price to perform an arbitrage transaction. It does not need to sell the currency bought to "enact" the arbitrage; instead we compute the value of the assets with the market USD price thus virtually completing the arbitrage. Using the US dollar is also quite instinctive for humans and many platform provide data on the exchange rate between any cryptocurrency and the US dollar.

In the case of this project, the total wealth of an account is comprised of:

- The value in USD of the ETH that the agent owns.
- The value in USD of the USDC that the agent owns.
- The value v_{shares} in USD of the Uniswap pair shares that the agent owns. The value of the shares owned is computed as follow:

$$
\begin{aligned}
r_{account} &= s_{account}/s \\
v_{ETH\to\$} &= r_{account} \cdot r_0 \cdot m_{p,ETH\to\$} \\
v_{USDC\to\$} &= r_{account} \cdot r_1 \cdot m_{p,USDC\to\$} \\
v_{shares} &= w_{USDC\to\$} + w_{ETH\to\$}
\end{aligned}
$$

Where:

  - $s_{account}$ is the amount of shares owned by the account.
  - s is the total amount of shares in the Uniswap pair.
  - r is the ratio of shares owned by the account for the Uniswap pair.
  - $v_{ETH\to\$}$ and $v_{USDC\to\$}$ are the value in dollar of the ETH resp. USDC owned by the account.
  - r_0 and r_1 are the ETH resp. USDC reserves of the Uniswap pair.
  - $m_{p,ETH\to\$}$ and $m_{p,USDC\to\$}$ are the market prices of ETH resp. USDC in USD.

Chapter 4

Reinforcement Learning

Reinforcement learning is a type of machine learning that falls in the category of unsupervised learning. The idea is to make a program learn from interactions it has with an *environment*. The environment is generally a simulation, like a physical simulation or a blockchain in our case. The program, called *agent*, is fed with an *observation* of the environment. This might be a total description of the environment, in which case it is called a *state* of the environment, or only a partial description. Given this observation of the environment, the agent is asked to choose an *action* among the set of possible one. Note that some part of the environment might be randomized. The agent is also fed with a feedback, called *reward*, on how well it performed so far, i.e. not only on how positive the last action it took was, but more globally on how well the agent is doing so far. This is the value that makes it possible for the agent to *learn*.

4.1 `BlockchainEnv`

The `BlockchainEnv` class is the heart of the reinforcement learning part of this project. It handles the simulated blockchain, the price oracles, keeps track of the time, generates transactions using the `TransactionSimulator`, etc.

A reinforcement learning environment, as defined by OpenAI/gym[1] [9] provides an interface with mainly two functions:

1. A `step` function that steps the simulation.
2. A `reset` function that start a new simulation.

Below is a short summary of the design choices made for the simulation.

[1] OpenAI/gym is the most commonly used reinforcement learning interface and there are many libraries that are compatible with it. Unfortunately, the documentation available online is very partial. A very useful although not straightforward reference is the GitHub repository hosting the code of the Python package.

Upon calling `reset`, the environment will create a new blockchain, reset the price oracles, create new simulated blockchain accounts—which implies that a different number of account might be created, and that each account might be initialized with different amounts of each crypto—and create a new Uniswap ETH-USDC pair. The Uniswap pair can be initialized with different policies regarding its exchange rate. One can decide to provide the exchange rate explicitly, to draw it at random or to have one that is identical to the market prices. This can be provided as an argument of the environment constructor. The default is to have a price identical to that of the market as this is probably the most useful for learning arbitrage transactions on popular Uniswap pairs. We remark however, that for exotic pairs, the exchange rate of the pair might not reflect the market price. It might be interesting to train the agent to be robust against exotic pairs too.

Then, the `BlockchainEnv` class generates everything that will happen in the next block in the function `_prepare_new_block`. It samples the time in second until the next block is mined from a Gaussian distribution Gaussian$(13, 3)$ as estimated from real life values of the Ethereum network. The number of Uniswap transactions that will occur in this block is also sampled from a Gaussian distribution whose parameters were computed using real-life data (Gaussian$(26.188, 10.603)$). The `TransactionGenerator` (see section 3.6) is used to generate the transactions at random. The time at which each transaction will be executed is also drawn at random uniformly in the time span of the block.

Calling the `step` function will lead to the release in the mempool of the next prescheduled transaction if there is any left. If there are none left, then calling `step` will cause the environment to trigger the mining of the next block, prices will be generated from the various price oracles, the agent wealth under the newly sampled prices will be computed as well as the reward. Finally the environment will prepare everything that will happen in the next block. The `step` function, upon completing a prepared block, will return that the simulation should be stopped if the agent has reached a total wealth of zero (which can happen because of the fees that the agent pays to execute transactions) or if the number of blocks completed so far is equal to the limit set when constructing the environment.

The environment will allow the agent to make a decision every time a transaction is received in the mempool of the simulated Ethereum network, i.e. once after each randomly generated transaction is submitted.

We remark that `rllib` (see 4.2) splits any simulation into episodes to perform the learning. These episodes might not coincide with the Ethereum blocks from the simulation.

4.1.1 Observation Space

Each observation contains the following information:

1. The amount of each token in the Uniswap pair, i.e. how liquid the pair is.
2. The price ratio in the Uniswap pair.
3. The price ratio of the market.
4. 30 slots for information about transactions that were submitted to the network but that are not yet mined. The transactions are ordered by decreasing gas price.

Each transaction will be described using the following values:

1. The type of the transaction (`mint`, `burn`, `swap0For1` or `swap1For0`).
2. The transaction amount.
3. The transaction sender.
4. The transaction nonce.
5. The transaction gas price.

Here is a short rational for each of these values. The liquidity of the pair is required to compute the optimal arbitrage transaction amount. The price ratio of Uniswap and that of the market are required to compute whether there is an arbitrage transaction to perform. The submitted-but-not-yet-mined transactions are required for the agent to be able to learn how to perform sandwich attacks (see section A) and arbitrage transactions altogether as the transactions that will be included in the block might shift the price of the Uniswap pair thus creating or destroying arbitrage opportunities. If there are less than 30 such transactions, the slots will be left with 0-values. Using 30 slots leave enough space for including all transactions of the block in more than half of the cases on average, as the number of Uniswap transactions in each block is sampled from a Gaussian distribution with mean 26.18 and standard deviation of 10.60. The transactions are ordered by decreasing gas price because transactions with high amounts generally have a high gas price in real-life and high-amount transactions are important for arbitrage opportunities as they might shift the price of the Uniswap pair. Thus the agent might learn that transactions in the first slots are more important when learning to perform arbitrage transactions or sandwich attacks. Remark that the transaction generator does not implement this correlation between high transaction amount and high gas price, i.e. transactions with high amounts will have a gas price coming from the same price distribution as all other transactions, which is a limitation of the current implementation. Another reason is that ordering transactions by decreasing gas price makes it possible to learn gas priority auctions. In such an auction, two arbitrage bots will compete over the same opportunity by submitting transactions with identical nonce but increasing gas price. As the auction might

happen fast in the case of bots, they might submit many transactions to win the opportunity, i.e. more than 30. At this point, it becomes important that the transactions are ordered by decreasing gas price, so that the agent knows of latest, i.e. the highest gas price, transaction submitted in the auction.

Transaction observations contain the transaction type which is required to know what effect the transaction will have. To reduce the size of the observation space, it might be interesting to only list swap transactions and therefore be able to remove the transaction type altogether from the transaction observations. Note that this will however prevent the agent from knowing about changes in liquidity in the pair during the block due to `mint` or `burn` transactions. The transaction amount is required to know whether the transaction will be large enough to generate an arbitrage opportunity. The sender and nonce makes it possible to know whether this transaction might override another transaction. Finally the gas price is important to know how it will be ordered in the block.

We remark that the observation space is pretty large, thus making training harder.

4.1.2 Action Space

An agent needs to provide actions that currently are defined as containing the following fields:

1. The type of Uniswap transaction to generate, i.e. `mint`, `burn`, `swap1For0` or `swap0For1`.
2. The percentage of the agent's balance to invest.
3. The transaction nonce.
4. The gas price to use.

The reason for not using absolute values when specifying the transaction amount, and to rely on balance percentage instead, is that it limits the size of the action space which would range from 0 to infinity while it ranges from 1 to 100 in our case. It further avoids the problem of transaction amounts that are larger than what the agent actually owns thus making learning easier. On the other hand, this strategy has the drawback that the agent does not know the absolute amount it is setting for the transaction. This makes it impossible to create optimum arbitrage transactions and is a major drawback. This problem might be solved by including the agent's ETH and USDC balance in the observation, but would require the agent to learn more things. Another option would be to use absolute values (which will make the action space much larger) and clip the actual value to the account's balance. This time, the issue is that the agent cannot know in

advance if it has the balance required to perform the transaction it wants, thus making it hard to exploit arbitrage opportunities again. Lowering the action space to only 10 values that would represent 10%, 20%, ..., 100% of the agent's balance would make learning faster at the cost of limiting more the agent's ability to produce transactions that have the optimal arbitrage value.

A general problem when specifying the transaction amount is that an agent might perform multiple transactions and thus try to sell more than 100% of its balance. As we cannot know which transactions will be mined (especially if there are multiple miner strategies implemented, see section 3.1), this behavior should be considered as valid. The check for valid amount must be performed at mining-time. In turn, this requires that transaction are atomic, i.e. that transactions either are fully executed or fully reverted, but never half-executed (see section 3.1.2 for details on atomic transactions).

The transaction nonce is implemented as a simple boolean value. A 0-value indicates that the agent desires to override its previous transaction. A 1-value indicates that the agent desires to use a new transaction nonce. We remark that this makes it impossible for the agent to override any other transactions than those with the previous nonce. This also means that an agent will not be able to play priority gas auction for sandwich attacks as it generally requires to compete over two transaction slots at the same time. This implementation was chosen as it lowers considerably the size of the action space. The alternative is to use an integer space going from 0 to infinity which would allow overwriting any non-mined nonce and to generate transaction with not-yet valid nonce which might yield interesting results.

The gas price is an integer value in the range 0 to infinity. This makes the action space quite large. Using only three values, e.g. fast transaction, normal speed and slow transactions like is found on many web interfaces these days, was considered. It would make learning much faster at the cost of making it impossible to play priority gas auctions. This might however still leave the agent enough freedom to perform arbitrage transactions if there are no aggressively competing arbitrage bot (which would steal the arbitrage opportunity by setting optimum gas price). Implementing three levels of gas price would, however, require to handle gas more thoroughly throughout the simulation (see section 5.2 to learn more about how to better handle gas).

4.1.3 Reward

Rewards from the simulation are always set to 0 except right after a block is mined. This specific moment is indeed the only one when it makes sense to compute the agent's wealth which is required to compute the agent's reward. Indeed, to be able to compute the agent's wealth (as described in section 3.7), one needs to know the amount of each crypto that the agent owns and the number of Uniswap

shares. And as long as the agent has outstanding transactions that might or might not be mined in the block, it is unclear what these numbers are.

In turn, this implies that the rewards of each of the potentially many transactions performed by the agent during a block are aggregated into a single value making learning harder as the agent cannot know which transaction or transactions improved or worsened the reward. Further, the reward value dependent on the market price of currencies. This means that there might exist some blocks during which it is not possible to have a positive reward because the price of ETH and USDC both go down steeply. Similarly, there might exist blocks during which the reward will be positive thanks to market prices going up, even though the agent's behavior was sub-optimal or even detrimental. It is not clear what the consequences on learning the randomness of market prices have.

As a final remark, we note that the dependence of the reward on the market prices induces the agent into trying to predict the market prices' evolution. Indeed, by betting on a currency whose value in USD goes up, the agent would always receive a positive reward. Investigating how the prices are generated is therefore important. This simulation makes exclusive use of price oracle that use random walks. We might believe that the market prices are thus unpredictable. However this is not true: the price oracles use increments of price that are drawn from distribution with potentially non-zero expected value (see sections 3.3 and 3.4). The problem is close to being non-existent for USDC as this is a stable coin pegged to the value of USD, thus on average the increment for USDC will have a zero value. However, the situation is quite different for ETH which has seen dramatic USD value increases since its early days. Thus the agent trained on this simulation will have an incentive to rely on the assumption that market evolution follows some distribution over time, e.g. that the price of ETH will go up and might therefore only learn to bet on ETH and wait.

4.2 Rllib

Providing a simulation environment is the first step to doing some reinforcement learning. The second is to have a framework in which to use the said environment to train an agent. This raises multiple questions like what model should we use to build the agent? how does the agent "learn"? how to choose the actions that the agent performs will learning (also called the exploration/exploitation tradeoff)? etc.

Implementing reinforcement learning from scratch is a very complex task and would represent more that a semester project in itself. There are indeed many algorithms to choose from: Advantage Actor-Critic, Proximal Policy Optimization, etc. Some exploration regarding reinforcement learning was conducted in order to find a framework that would allow drop-in use of reinforcement learning algorithms.

The RL field is quite novel however and there are not so many options available out there. The most common models for the agent are neural networks. Neural network libraries like PyTorch and Tensorflow have tutorials or even sometimes light-weight libraries for doing RL, e.g. Tensorflow Agent [6]. Those however are generally partial and still require the user to implement many aspects like the actual learning algorithm.

We decided to use `rllib` [12] instead. The library is based on `ray` which is a distributed-computing library. It is not mandatory to use the distributed computing aspect of `ray` to use `rllib`.

While being state of the art, implementing many RL algorithm and interfacing with PyTorch and TensorFlow, `rllib` does not provide a very extensive documentation, and the examples showcasing the usage of the library are still rather sparse. This being probably caused by how recent the whole field is. Using the library was therefore not straightforward and required perusing the documentation to find specific bits of information or to understand the cause of some errors.

`Rllib` provides many reinforcement learning algorithms out-of-the box. We settled on using Proximal Policy Optimization as it features great properties and is recommended as a default options by `rllib`.

Chapter 5

Approximating Reality

If we want our agent to learn things that can be transferred in reality, we need the simulation to approximate reality closely enough. There are a few shortcoming in the current implementation that are described hereafter.

5.1 Transaction Ordering Algorithms

In the real Ethereum blockchain, there are many transaction ordering algorithms being used. Ordering transactions in order to maximize the miner's benefit is probably a goal that most of these algorithms follow. However, there are multiples ways to achieve this. A pretty standard one is to order the transactions by decreasing gas price, but this greedy strategy might not yield the best possible ordering and there are many other aspects to consider.

The transaction ordering algorithm implemented in this project probably mimics closely enough what most miners do in real-life, however, it does in no way provide a comprehensive view of all the algorithms possible.

Note that transaction ordering is an important aspect that the agent must learn in order to be able to perform sandwich attacks. Indeed, to perform such an attack, the agent must understand how to frontrun and postrun a transaction with its own transactions (see section A for the details of the sandwich attack). Training the agent in an environment with multiple miner algorithms that might order transactions differently would make the simulation much more realistic and might enable more robust learning.

5.2 Gas Costs

Currenlty, gas price of generated transactions is drawn at random from a single Gaussian distribution generated from historical values. This model is pretty far from reality in which it is considered that there is a gas price (expressed in ETH) and accounts performing transactions in the Ethereum blockchain can choose to

use the market price to get their transactions mined at a regular speed, set the price lower for slow transaction or set the gas price higher for fast transactions.

This also has implication on what the agent can learn (see section 4 for details). In this specific case, generated transactions in the simulation will always have prices drawn from the same price distribution, while the reality would be that transactions have price drawn from a distribution whose mean changes over time. As the agent needs to learn how to order transactions, this is probably a simplification that would is not reasonable if the agent is to run in real life.

For example, implementing the market gas price with an oracle and having the generator choose prices from a distribution centered around this price would be closer to reality.

5.3 Uniswap Exchange Rate

On a Uniswap pair, each swap transaction will cause the exchange rate to shift. For popular Uniswap pairs, the Uniswap exchange rate never deviates too much from the market prices because of fierce competition over the arbitrage opportunities. For exotic pairs on the other hand, it is often the case that the price on Uniswap can deviate quite a lot from the market prices.

We want to be able to simulate all these behaviors in our simulation too, which is the reason why the transaction generator can generate arbitrage transactions. Note that simulating exotic pairs is much simpler: limiting the amount of arbitrage transactions will yield the desired result. Simulating popular pairs is much harder. Unfortunately, simulating popular pairs is also a lot more interesting, indeed as they have much larger liquidity pools, arbitrage opportunities will generally be much larger. Finding the good ratio of arbitrage transactions to regular swap transaction is crucial to mimic reality, i.e. keep the Uniswap price close to that of the market. The liquidity on the Uniswap pair as well as the amounts allocated at the beginning of the simulation to simulated accounts need to strike a balance. On one hand, accounts need to have enough liquidity to perform arbitrage transactions that actually bring the Uniswap rate back to that of the market. On the other hand, they must not be able to shift the Uniswap price too much[1].

Figure 5.1 shows the exchange rates in the ETH/USDC Uniswap pair along with the market prices generated by the price oracles in a 100 block simulation, which is approximately equivalent to 20 minutes[2]. We remark that the current setup would need some further refinement as the Uniswap price diverges too much from the market price.

[1] In reality, large swap transactions are generally split into multiple transactions to prevent sandwich attacks.

[2] A new block is produced every 13 seconds on average in our simulation, thus with 100 blocks, we have a pproximately 1300 seconds which is 21.66 minutes.

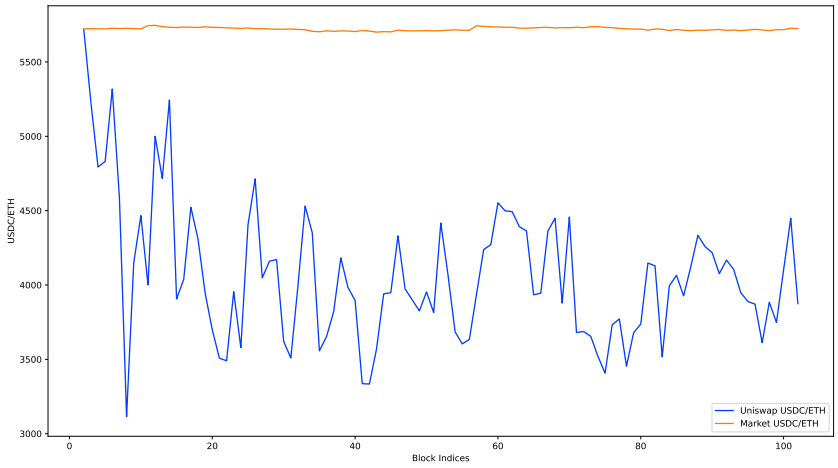


Figure 5.1: This figure shows the price of ETH in USDC over time, where time is indexed by Ethereum blocks mined. The blue line displays the exchange rate in the Uniswap pair. The yellow line shows the market exchange rate as randomly generated by the price oracles of ETH and USDC.

Chapter 6

Results

The simulation is, to the best of our knowledge, bug-free. However the training still crashes when `rllib` tries to set the gas price to infinity. The stack trace for such a case is given in appendix B. Fixing this issue requires changing the action space of the agent, i.e. changing how the agent specifies the gas price of its transactions. Unfortunately, it is not a straightforward change; some more explanations are given in section 8.1.

So far, we were able to train the agent for 135 iterations at most. The rest of the time, either the computer crashed because of lacking RAM memory (see section 7.3), or `rllib` tried to set the gas price of some transaction to infinity thus crashing the simulation. Nevertheless, it seems interesting to look at whether the agent was able to learn something in such a little time and to explore an example simulation to understand what the simulation does. This section delves into the outcome of the agent trained for 135 iterations.

6.1 Simple Evaluation

The only way we know of to evaluate an agent is to run the agent on multiple simulations and look at how it performs compared to some other baseline. The `evaluation` script does exactly this: it runs the agent as well as some baseline like the idle baseline and the mint baseline (see section 2.4) a fixed number of times and compare the average wealth increase percentage.

After running this evaluation script, we obtained the following averaged results:

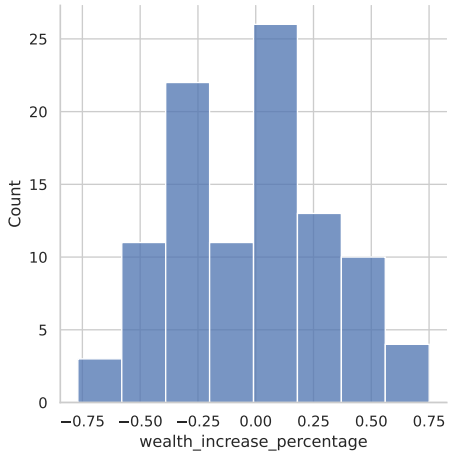
Idle Baseline -0.013% of wealth increase on average.

Mint Baseline 0.007% of wealth increase on average.

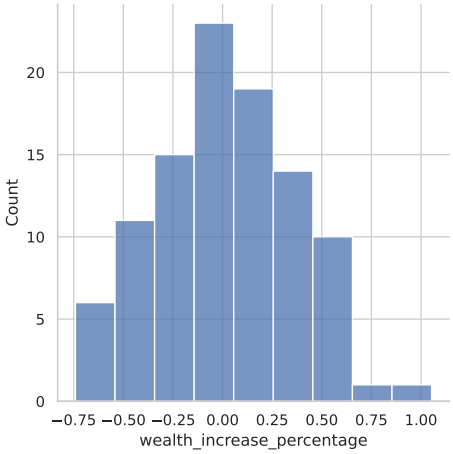
Trained Agent -7.018% of wealth increase on average.

The distribution of wealth increase percentage of each of the agents across the 100 simulations is shown in figure 6.1.

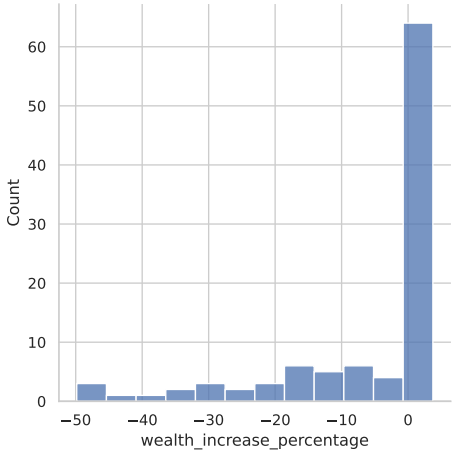
Figure 6.1: Distribution of the wealth increase percentages across the different baselines and the trained agent.



(a) Distribution of the wealth increase in percentage of the idle baseline.



(b) Distribution of the wealth increase in percentage of the mint baseline.



(c) Distribution of the wealth increase in percentage of the trained agent baseline.

It seems quite clear that, if the current learning environment makes it possible at all to learn something, then the agent was not trained for long enough to learn it. Unfortunately, these statistics give only little details about what is going on and how the agent is actually behaving.

6.2 Advanced Evaluation

This second script, `advanced_agent_evaluation.ipynb` makes it possible to explore deeper what is going on in a single simulation. Its goal is to make it possible to understand what the agent learned or to do some debugging. Indeed, any implementation mistake might provide opportunities for the agent to "hack" the simulation, which it most likely will as the agent tries to maximize its wealth not follow the rules.

Hereafter are given some information on an example simulation to give some feeling of what is actually going on. The simulation was run for 100 blocks. In total, there was 8 blockchain accounts that took part:

```
Account(address=miner_1, 22.930280609753773ETH, 0.0USDC)
Account(address=agent, 1000.1953977610376ETH, 60.3USDC)
Account(address=generated_account_0012, 176325.49172518332ETH,
↪  1430522338.1633081USDC)
Account(address=generated_account_0013, 157766.16543026388ETH,
↪  841260424.6515439USDC)
Account(address=generated_account_0014, 118300.21093045667ETH,
↪  33984247.855853155USDC)
Account(address=generated_account_0015, 68664.3262678783ETH,
↪  15949278.197112404USDC)
Account(address=generated_account_0016, 275398.84974007425ETH,
↪  15497208.399045324USDC)
Account(address=uniswap_pair_ETH/USDC, 6059209.301757226ETH,
↪  23475763015.034077USDC)
```

Beside the account of the agent, we see one miner account, an account for the Uniswap smart contract and 5 simulated blockchain accounts (`generated_account_0012` to `generated_account_16`). The balance of the accounts at the end of the simulation is also included.

The agent only performed two transactions in this simulation which are summarized in table 6.1.

Block	Type	Amount [USDC]	Gas Price [ETH]
4	`swap1for0`	910	0.0
5	`swap1for0`	30	0.0

Table 6.1: The transactions sent from the agent that got mined during the simulation.

We remark that the agent performs very few transactions. This behavior was remarked for all the agents trained so far, i.e. all those trained on the latest version of the simulation, but also for those trained on previous versions of the simulation. It is not entirely clear why the agent act so rarely.

We remark that the agent only performs one type of transaction: `swap1for0`. It sells almost all of its USDC to get ETH: from the 1000 USDC that the agent is allocated at the beginning of the simulation, it only keeps 60.

We remark that the gas price is always set to 0 which seems to indicate that transaction ordering is of little concern to the agent. In real life, transaction with a gas price of zero would likely never be mined. Because this simulation only considers Uniswap transactions, the miner have only very few transactions to choose from, thus even with a gas price of zero the transactions are mined.

A guess that could explain why the agent act so little, with such big transactions and to sell all its USDC is that it bets that the value of ETH in USD will go up and maximizes therefore the benefits in USD. Statistically speaking, this makes sense as the ETH price oracle will generate increasing prices on average. Furthermore, keeping USDC, which is a stablecoin pegged to the value of the US dollar has no chance of yielding benefits in US dollar over time. This could also explain why the agent ignores the gas price: as long as the transaction is mined rapidly enough, it does not make much difference. This also means that the current setup fails at making it possible or interesting for the agent to learn arbitrage transactions. This might be caused by the learning time which is too short, i.e. the agent had not enough time to learn such complex behaviors, or it might be because the reward function is defined in a way that incentivizes betting on the exchange rate of ETH (high risk, high reward) over performing arbitrage transactions (no risk, low reward). If the second is true, then the reward function will need to be changed to something that enables learning of arbitrage transactions.

The agent balances across time are given in figure 6.2. It illustrates visually the transaction data: the two transactions in blocks 4 and 5 are clearly visible

The value of ETH in USD as generated by the price oracles during the simulation is given in figure 6.3. The price of USDC in USD is of very little interest as the ratio was 1:1 up to the third decimal at least at all time during the simulation. We remark that the agent's wealth evolution closely follows that of the value of ETH in USD. This can be explained by the fact that the agent basically bought all the ETH it could at the beginning of the simulation and then waited.

We also remark that the agent never became a liquidity provider, i.e. never performed `mint` transactions. This can be explained by the fact that the simulation generates probably less swap transactions than there are in real-life, thus yielding less fees which makes the liquidity provider strategy less interesting.

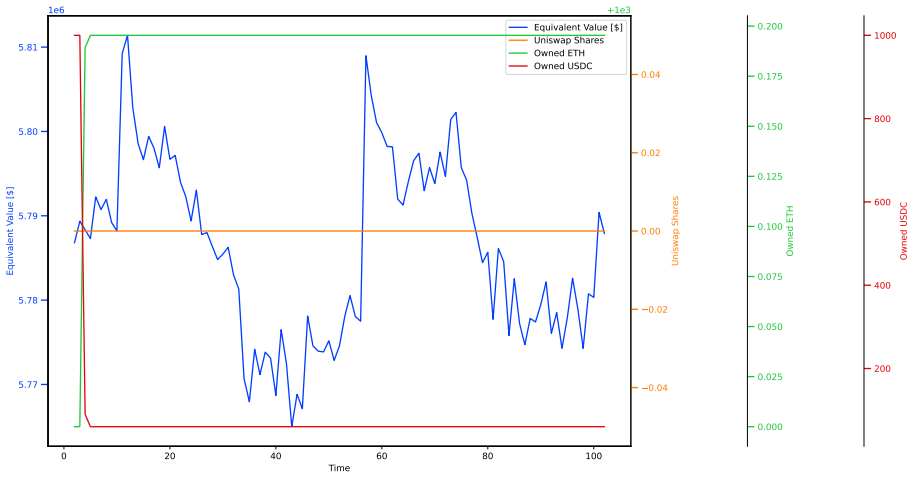


Figure 6.2: Agent balances of various currencies and equivalent wealth in USD across the blocks of the simulation.

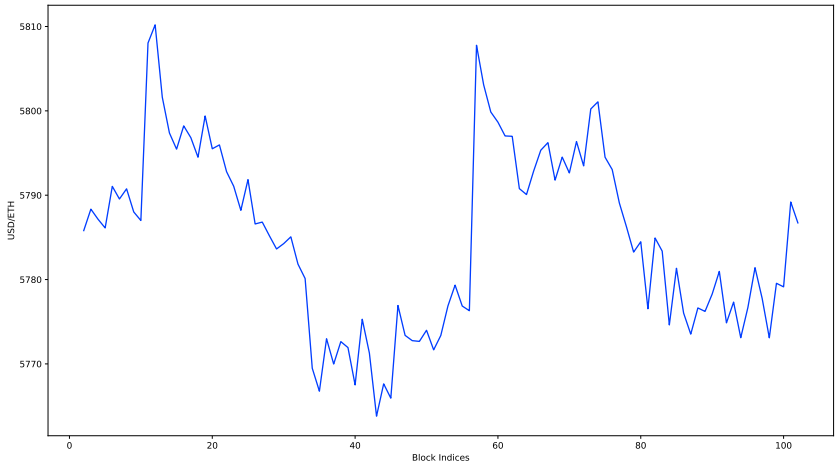


Figure 6.3: The price of ETH in USD during the simulation.

Chapter 7

Issues During the Project

Hereafter are listed some of the issues or difficulties that arose during the project.

7.1 Reinforcement Learning

Performing reinforcement learning is both highly technical and tricky. Some slight variation in the setup of the environment might make the difference between learning something and learning nothing at all. Therefore, finding some good library to do it was a major requirement.

Reinforcement learning is still a cutting edge technology and there are not so many implementations available online. Most of the available ones are under-documented and there is no consensus on which library is the best to use. Therefore we were pretty much left to trying to figure out which library would be the quickest and easiest to use by actually trying them out.

Also, the interaction between reinforcement learning libraries, that perform the actual learning, and standardization libraries like `OpenAI/gym`, that only define the interface that environments and agents must provide, was confusing. There are indeed multiple standardization libraries available, and each reinforcement learning library integrates with only a subset of the standardization ones. They might not even list the libraries the integrate in their documentation.

We found it pretty hard to navigate the `rllib` documentation that sometimes gives very low-level information making it generally irrelevant and sometimes forgets to mention important high level information like how to use the logger or to give a working example of how to perform some RL training with reasonable settings.

Even though `rllib` is pretty hard to use and the documentation a bit sparse and complex to navigate, we believe that it was the best choice of RL library. Of the many alternatives we explored, it is the only one that provides off-the-shelf implementation of RL algorithms. Most other libraries still require the developer to implement many aspects of the learning, which would have made this project

both longer and more complex.

7.2 Logger

`Rllib` uses the standard python `logging` library. When running the training in multi-processed mode (which is the default mode), `rllib` does not print the logs of the workers. This makes it much harder to understand what is going on and what might not be working properly in the environment. This was a major problem for quite some time. The solution we used in the end was to revert to the "local mode" of `rllib` which runs the training code in a single process and makes it possible to see the logs generated by the environment. The downside is that we do not benefit from parallelism anymore, but it was deemed reasonable during the coding phase.

7.3 RAM

Training a RL agent uses a very large amount of RAM space as the gradients of each weight of the neural network underlying the agent needs to be saved. This caused the training process to fail regularly on our 32GB RAM computer. Training on a computer with larger memory might solve the issue. Furthermore, training on a cluster, i.e. using the parallelism provided by `ray` which is the library underlying `rllib`, might make the learning process faster. However, deploying `ray` on a cluster is not trivial and will require some time. Instructions to do this can be found in [11].

Chapter 8

Future Improvements

We list here what we believe to be the major points that need improvement.

8.1 Gas Price Action Space

Currently, the agent can set any value for its transactions' gas price, from 0 to infinity. Setting the price to infinity crashes the simulation which has prevented us from the training the agent long enough for it to learn anything useful. It also makes the action space very large. A better solution needs to be found. This solution however, should still make it possible for the agent to learn gas priority auctions. Also, the agent is not currently fed the price of gas which could be useful.

8.2 Simulated Gas Price

Gas price needs to be handled in a more detailed fashion in the `BlockchainEnv` class. The gas price should evolve over time using a price oracle and the transaction generator should make clever choices about the gas price to set in the generated transactions. For example, it could sample gas prices from a distribution centered around the current gas price. Transactions with higher gas price should be the ones executed the fastest, while those with a lower gas price will be executed in a slower fashion. The miners might also benefit from better heuristics for estimating the gas cost of a transaction than sampling from the historical gas costs.

8.3 Uniswap Exchange Rate

Some fine-tuning is required on the liquidity of the Uniswap pair and on the amounts that the generated accounts are given at the beginning of the simulation in order to guarantee that the difference in exchange rates between the Uniswap

pair and the market matches those found in real life. We remark that currently the difference are larger than what they are in real life. We further remark that the real life differences might sometimes be large too, for example for exotic pairs, and it might therefore be interesting to train different agents for different types of pair, i.e. from mainstream to exotic ones.

8.4 Code Refactorings

At the implementation level, almost all classes, methods and attributes have been documented extensively which should make extending the code rather straightforward. The one class that would benefit from a rewrite is the `BlockchainEnv` one which is more than 600 lines of code long and could be broken down to multiple subclasses, each handling one aspect of the simulation.

8.5 Miner Algorithm

Currently, the miners have very few transactions to choose from when mining their block as only Uniswap transactions are considered in this simulation while the real-life Ethereum blockchains also features all the other transactions. This implies that even with a gas price set to 0, a transaction has a high probability of being mined fast. This is obviously quite different from what happens in real life and will probably impact what the agent can learn regarding transaction ordering.

8.6 Uniswap Transactions

Even though the number of Uniswap transaction per block is realist, the amounts of the transactions are too small to generate enough fees to make the liquidity provider strategy an interesting one. Fine-tuning the amounts of the generated transactions to increase the amount of fees generated seems like a desirable improvement that could impact what the agent learns: it could end up choosing to become a liquidity provider in some instances.

Chapter 9

Conclusion

This project was very interesting, it was the opportunity to learn about many aspects of the Ethereum blockchain like transaction ordering algorithms, constant-product automated market maker like Uniswap, the mempool, etc. Implementing the Ethereum simulation was fun, even though more complex than initially envisioned.

The reinforcement learning part was the trickiest. Finding RL libraries, testing them out and then using them proved to be very difficult as they are not well documented. Further, designing the action space, observation space and reward function was a challenging task that might require some more tuning in the future.

This project was also an opportunity to learn more about finance, arbitrage transactions and some of the mathematical models that we use today.

Overall, the very broad scope of this project made it both very interesting but also difficult. The lack of defined goals in term of what we wanted to learn made it also hard to decide which road to follow. Learning arbitrage transactions requires different observation space, action space, reward function and simulation functionalities than learning gas priority auctions for example. We ended up implementing an environment that could accommodate both, with the upside that it was very interesting to implement, but with the downside that it is rather complex.

Appendix A

Sandwich Attacks

The sandwich attack is summarized as follow by [13]:

> The problem: Crypto traders lose millions every month because their transactions are being sandwich-attacked. There are dozens of bots continuously monitoring the mempool to find transactions they can frontrun. So far it was difficult for traders to estimate whether their swaps on decentralized exchanges were susceptible to sandwich attacks or not.
>
> The attack: If a victim swaps asset A for asset B, this will increase the price of A in the respective liquidity pool. Bots spot the victim transaction before it executes. They then release two transactions that surround the victim trade (frontrunning and backrunning it). In their first transaction, 'Attacker Tx1', they swap A for B. After the victim transaction executes, they swap B for A, and make a profit due to the price increase after the victim transaction. Attackers have to pay an exchange fee (on Uniswap this is 0.3% of the input amount). In case they are not colluding with the miner, they also have to pay for gas. The price difference resulting from a victim transaction is not necessarily large enough for a profitable sandwich attack.

Appendix B

Stack Trace of the Infinite Gas Price Error

The following error was thrown at the 136$^{\text{th}}$ iteration of the agent's training by `ray`. It is caused by some computation failing when `rllib` tries to set the gas price to `np.inf`, i.e. to infinity.

```
Failure # 1 (occurred at 2021-09-03_22-11-41)
Traceback (most recent call last):
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/tune/trial_runner.py", line 739, in
↪  _process_trial
results = self.trial_executor.fetch_result(trial)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/tune/ray_trial_executor.py", line 729, in
↪  fetch_result
result = ray.get(trial_future[0], timeout=DEFAULT_GET_TIMEOUT)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/_private/client_mode_hook.py", line 82, in
↪  wrapper
return func(*args, **kwargs)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/worker.py", line 1564, in
↪  get
raise value.as_instanceof_cause()
ray.exceptions.RayTaskError(AssertionError):
↪  ^^[[36mray::PPO.train()^^[[39m (pid=1042179, ip=10.68.152.189)
File "python/ray/_raylet.pyx", line 534, in
↪  ray._raylet.execute_task
File "python/ray/_raylet.pyx", line 484, in
↪  ray._raylet.execute_task.function_executor
```

```
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/_private/function_manager.py", line 563, in
↪  actor_method_executor
return method(__ray_actor, *args, **kwargs)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/agents/trainer.py", line 640, in
↪  train
raise e
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/agents/trainer.py", line 629, in
↪  train
result = Trainable.train(self)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/tune/trainable.py", line 237, in
↪  train
result = self.step()
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/agents/trainer_template.py", line 170, in
↪  step
res = next(self.train_exec_impl)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 756, in
↪  __next__
return next(self.built_iterator)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 843, in
↪  apply_filter
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 843, in
↪  apply_filter
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
```

```
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
[Previous line repeated 1 more time]
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 876, in
↪  apply_flatten
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 828, in
↪  add_wait_hooks
item = next(it)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 783, in
↪  apply_foreach
for item in it:
[Previous line repeated 1 more time]
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 471, in
↪  base_iterator
yield ray.get(futures, timeout=timeout)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/_private/client_mode_hook.py", line 82, in
↪  wrapper
return func(*args, **kwargs)
ray.exceptions.RayTaskError(AssertionError):
↪  ^^[[36mray::RolloutWorker.par_iter_next()^^[[39m (pid=1042172,
↪  ip=10.68.152.189)
File "python/ray/_raylet.pyx", line 534, in
↪  ray._raylet.execute_task
```

```
File "python/ray/_raylet.pyx", line 484, in
↪  ray._raylet.execute_task.function_executor
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/_private/function_manager.py", line 563, in
↪  actor_method_executor
return method(__ray_actor, *args, **kwargs)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/util/iter.py", line 1151, in
↪  par_iter_next
return next(self.local_it)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/evaluation/rollout_worker.py", line 339, in
↪  gen_rollouts
yield self.sample()
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/evaluation/rollout_worker.py", line 740, in
↪  sample
batches = [self.input_reader.next()]
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/evaluation/sampler.py", line 101, in
↪  next
batches = [self.get_data()]
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/evaluation/sampler.py", line 231, in
↪  get_data
item = next(self.rollout_provider)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/evaluation/sampler.py", line 652, in
↪  _env_runner
base_env.send_actions(actions_to_send)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/env/base_env.py", line 363, in
↪  send_actions
self.vector_env.vector_step(action_vector)
File "/home/yves/.virtualenvs/rlmbc/lib/python3.9/site-packages/ra
↪  y/rllib/env/vector_env.py", line 173, in
↪  vector_step
obs, r, done, info = self.envs[i].step(actions[i])
File "/home/yves/Documents/studies/2019_ethz/class/2021-02_compute
↪  r_science/deep_learning_meets_blockchain/rl_blockchain/src/mai
↪  n/python/rlmbc/reinforcement_learning/environment/blockchainen
↪  v.py", line 484, in
↪  step
self.network.mine_next_block(len(self._transactions_to_execute))
```

```
File "/home/yves/Documents/studies/2019_ethz/class/2021-02_compute
↪  r_science/deep_learning_meets_blockchain/rl_blockchain/src/mai
↪  n/python/rlmbc/blockchain/network.py", line 175, in
↪  mine_next_block
new_block = miner.mine_block(maximum_transactions_count)
File "/home/yves/Documents/studies/2019_ethz/class/2021-02_compute
↪  r_science/deep_learning_meets_blockchain/rl_blockchain/src/mai
↪  n/python/rlmbc/blockchain/miner.py", line 118, in
↪  mine_block
self._network.ETH.transfer(transaction.sender, self.address,
↪  fee_amount_eth)
File "/home/yves/Documents/studies/2019_ethz/class/2021-02_compute
↪  r_science/deep_learning_meets_blockchain/rl_blockchain/src/mai
↪  n/python/rlmbc/blockchain/cryptocurrency.py", line 71, in
↪  transfer
assert self._balances[sender] >= amount, f'You tried to transfer
↪  {self.to_str(amount)} ' \
AssertionError: You tried to transfer nanETH from agent to
↪  miner_202. This is not possible because agent has only
↪  100.0ETH. Delta: nanETH.
```

ETH transfers from the agent to a miner only happen to pay fees of a transactions, and the only way to achieve a value of `nan` for the fee is to multiply by `np.inf`.

Bibliography

[1] *[1904.05234v1] Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges*. URL: https://arxiv.org/abs/1904.05234v1 (visited on 09/27/2021).

[2] Guillermo Angeris et al. *An Analysis of Uniswap Markets*. Feb. 9, 2021. arXiv: 1911.03380 [cs, math, q-fin]. URL: http://arxiv.org/abs/1911.03380 (visited on 09/24/2021).

[3] *ERC-20 Token Standard*. ethereum.org. URL: https://ethereum.org (visited on 09/23/2021).

[4] *Ethereum USD (ETH-USD) Price, News, Quote & History - Yahoo Finance*. URL: https://finance.yahoo.com/quote/ETH-USD/ (visited on 09/23/2021).

[5] *Geometric Brownian Motion*. In: *Wikipedia*. June 4, 2021. URL: https://en.wikipedia.org/w/index.php?title=Geometric_Brownian_motion&oldid=1026817534 (visited on 09/23/2021).

[6] *Introduction to RL and Deep Q Networks | TensorFlow Agents*. TensorFlow. URL: https://www.tensorflow.org/agents/tutorials/0_intro_rl (visited on 09/27/2021).

[7] *Knapsack Problem*. In: *Wikipedia*. Aug. 26, 2021. URL: https://en.wikipedia.org/w/index.php?title=Knapsack_problem&oldid=1040735728 (visited on 09/23/2021).

[8] *Lévy Process*. In: *Wikipedia*. Sept. 21, 2021. URL: https://en.wikipedia.org/w/index.php?title=L%C3%A9vy_process&oldid=1045574085 (visited on 10/04/2021).

[9] *Openai/Gym*. OpenAI, Sept. 24, 2021. URL: https://github.com/openai/gym/blob/65eebce90457cab66786f69db9658b00e2eb0808/gym/core.py (visited on 09/24/2021).

[10] *PyBuilder — an Easy-to-Use Build Automation Tool for Python*. URL: https://pybuilder.io/ (visited on 10/04/2021).

[11] *Ray with Cluster Managers — Ray v2.0.0.Dev0*. URL: https://docs.ray.io/en/master/cluster/deploy.html (visited on 10/02/2021).

[12] *RLlib: Scalable Reinforcement Learning — Ray v2.0.0.Dev0*. URL: https://docs.ray.io/en/master/rllib.html (visited on 09/27/2021).

[13] *Sandwich Calculator*. URL: https://www.defi-sandwi.ch/ (visited on 09/27/2021).

[14] *Student's* t*-Distribution*. In: *Wikipedia*. Aug. 31, 2021. URL: https://en.wikipedia.org/w/index.php?title=Student%27s_t-distribution&oldid=1041600935 (visited on 09/23/2021).

[15] *Uniswap V2*. Uniswap, Sept. 23, 2021. URL: https://github.com/Uniswap/v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Pair.sol (visited on 09/23/2021).

[16] *Wiener Process*. In: *Wikipedia*. Aug. 2, 2021. URL: https://en.wikipedia.org/w/index.php?title=Wiener_process&oldid=1036675781 (visited on 10/04/2021).