Master's Thesis

Copenhagen Business School

Systematic Trading Strategies with Reinforcement Learning

Authors
Victor Emil Skov Lundmark (S120401)
Lucas Johan Boesen (S118819)

Professor
Søren Feodor Nielsen

MSc in Business Adminstration and Mathematical Business Economics

May 16th 2022

Characters: 173.873
Number of pages: 89

Abstract

Intertemporal choice is fundamental in many economic and financial decision-making problems. In this thesis we first present reinforcement learning theory and then show how it can be used as a tool to model and approach two of these problems fully automated. The first is the classical utility maximization problem framed as an investor performing portfolio optimization trading an equity index. Here we show that a risk-averse agent tends towards a classic buy and hold strategy the longer it trains. The second is a method to develop automated market-making trading strategies. We do this by simulating a stock market in an agent-based model, which lets us mitigate some common assumptions, such as no market impact and the absence of transaction cost, as well as model the dynamics of the order book. Here we see that the agent learns and improves its performance through time, but slowly. And due to the lag of computing power, we have not been able to run the experiment for as long as desired.

Resumé

I mange problemstillinger indenfor økonomi og finans er tid altafgørende for resultatet. Vi præsenterer først teorien bag reinforcement learning, og derefter hvordan den kan anvendes til at modellere og takle to af disse helt automatiseret. Den første er det klassiske nyttemaksimeringsproblem i et porteføljeoptimerings setup. Her viser vi, at en riskikofølsom agent konvergerer mod en klassisk køb-og-hold strategi desto længere den træner. Den anden er en metode til at udvikle automatiserede market making trading strategier. Vi gør dette ved at simulere et aktiemarked i en agentbaseret model, hvilket lader os se mildere på nogle klassiske antagelser så som ingen transaktionsomkostninger og ingen markedspåvirkninger, samt modellere dynamikkerne i orderbooken. Her ser vi, at agenten lærer og forbedre dens performance over tid, men gør det langsomt, og i mangel på computerkraft har vi ikke kunne køre forsøget så længe, som vi havde ønsket.

Contents

1 Introduction

The influence of systematic trading strategies on the financial markets is increasing every day, and today, trades executed through algorithms account for 63-70% of the volume in the US equity markets and 92% in the forex market (Kissell 2020). The idea of automatic and systematic financial decision-making has been around for a long time. It started in the 1970s, together with the growth of computer technology and the implementation of The Designated Order Turnaround, which made trading electronic. Back then, the focus was on statistical arbitrage and asset pricing models. In the last decade, machine learning (ML) and artificial intelligence (AI) techniques have gained enormous popularity and are today influencing numerous aspects of human life. Within the world of AI, especially reinforcement learning (RL) has accomplished great success. Through self-taught agents, it has developed state-of-the-art robots, self-driving cars, and computer programs for playing games. It is the core component in the first computer program to beat the world champion in the game Go and is likewise the method used in the best chess engine as of today. These achievements have drawn the attention of researchers within finance to investigate whether RL techniques can perform equally well playing in the financial markets. This thesis presents the theory of modern RL techniques and shows how they can be applied as tools to solve financial decision-making problems within portfolio optimization and market making.

1.1 Structure

This thesis is structured into four main parts, which individually is split into sections.

In Part I we present the theoretical background in RL. It includes section 2-4. In section 2 we introduce Markov decision processes, which is the framework used to model the sequential decision making we are attempting to solve. In section 3 we present classical methods to approach MDP's and the transition to the basic RL approaches. In section 4 we go through generalized and advanced RL methods, which can be used to solve most real life representations of Markov decision processes.

In Part II, we present our first experiment using RL for portfolio optimization trading a stock. The part is split into sections 5-10. In section 5, we introduce the experiment and its theory and background. In section 6, we present the data used and the methods that we use to simulate more data to train our RL agent upon. In section 7, we present the implemented RL agent and the framework in which it learns. In sections 8, 9 and 10 we represent the experiment's results, discussion and conclusion.

In Part III, we present the second experiment, where we use RL to train an agent using an agent-based model simulation of the underlying dynamics in a stock market. It is split into sections 11-15. In section 11, we introduce the background of the experiment and present some recent related research within the field. In section 12, we go through the agent-based model framework, how we construct it and how to use it for simulation. In section 13, we introduce real high-frequency data and suggest a method to calibrate the agent-based model to this data. Finally, we compare the statistics of the simulated data

and the real data. In section 14, we present the market making RL agent, which we train using the agent-based simulation. In section 15, we show the results, and in sections 16 and 17, we discuss and conclude upon the experiment.

In Part IV, we give an overall conclusion on the experiments and analyses done in the thesis, as well as reflect on how these can be improved for further studies. This is respectively done in section 18 and 19.

1.2 Problem Statement

In this thesis we aim to answer the following problem statement:

How can reinforcement learning be applied as a tool for implementing systematic trading strategies within portfolio optimization and market making?

To do this we answer the following sub-questions

• What is Markov decision processes?

• How do we solve Markov decision processes using reinforcement learning methods?

• How can policy gradient methods be used to directly model the decision process, rather than the possible outcomes?

• How can the portfolio optimization problem be framed, so that it can be solved using reinforcement learning?

• How can we model stock prices in order to simulate more data for the purpose of training a reinforcement learning agent?

• How do we train a reinforcement learning agent in order to perform portfolio optimization?

• How can we model the underlying dynamics of the financial markets using agent-based model simulation?

• How can we evaluate how well the agent-based model reflects the stylized facts of the financial markets

• How can reinforcement learning be used to learn systematic trading strategies in an agent-based model simulation of the financial markets?

1.3 Research Delimitation

In this thesis, we wish to dive into reinforcement learning, focusing on the class of policy gradient methods to tackle portfolio optimization and market making, where the under-

lying dynamics of the order book come into play. We will only apply the reinforcement learning class of policy gradient methods within the experiments. Concerning the portfolio optimization problem, we will limit it to a simple representation of an equity market, including a single index. In the market making problem, we will focus on the applications of reinforcement learning for market making solely, though it could easily be extended to other types of actors too.

1.4 Code

All the code for this thesis is implemented in Python and can be found at github.com/rl_for_trading.

Part I
Reinforcement Learning Theory

RL is a class of ML techniques. The main difference between ML and statistics is their purpose. Generally statistics aim to explain inference regarding the relationship of variables, while ML attempts to make precise predictions out of sample. This does not mean that statistical models can not be used for prediction, just that it is generally not their main purpose. Within ML we have three major categories: *Supervised learning* (SL), *Unsupervised learning* (UL) and *reinforcement learning* (RL).

SL attempts to learn from labeled data, i.e. we wish to explain Y (label) from X (input). The term learn generally refers to estimation and the two words will be used interchangeably. This could for instance be the act of 'learning' the parameters of an underlying distribution, or 'learn' the parameters optimizing a parameterised objective function. The purpose of SL is to learn in a generalized way, such that given another sample of inputs X_2 we can predict the corresponding labels Y_2.

In contrast UL tries to learn from unlabeled data. The goal is to extract information from the data, which then can use to label provide labels to the data. A common use case is clustering. UL can be used to create clusters of observations, which reveal some of the underlying similarities in the data.

RL is focused on learning policies through trial and error. In RL problems, there is not a human modeler to collect or label any data. The goal is to make decisions, interact and observe through agents, and then let the agents learn from these experiences (observations). An example could be a game playing agent. The agent trains by playing the game and will hopefully be better the more it plays. An appealing factor of RL modeling is that the agents can output an optimal policy, which can be directly implemented without the need for human input afterwards. This is practical for the purpose of full automation of tasks. In comparison, normal statistical models will output a probability distribution over possible outcomes, which humans then have to select their decision upon.

2 Markov Decision Processes

A Markov decision process (MDP) is a framework satisfying the Markov property, while allowing a model to interact with and affect an underlying probability distribution. In this section we will thoroughly go through the concepts needed to understand MDP's as well as showing methods to model and find optimal solutions to MDP problems. We use the notation introduced by S. R. Sutton and Barto 2018 with slight modifications.

2.1 Markov states

We let $S_t \in \mathcal{S}$ be the state of a stochastic process, where $\mathcal{S}$ is a finite set of all its possible states. For the purpose of this introduction, we assume $\mathcal{S}$ to be finite, though we will later show how to generalize to infinite state representations. A stochastic process is Markov (a Markov process or Markov chain) if it meets the Markov property:

Definition 2.1. (Markov Property) *A state S_t is Markov if and only if*

$$\mathbb{P}(S_t \mid S_{t-1}) = \mathbb{P}(S_t \mid S_1, ..., S_{t-1}).$$

I.e the current state is a sufficient statistic of the future and thus we can ignore the history of the process and keep our model simple by only keeping track of the information in the current state.
Let

$$p(s' \mid s) = \mathbb{P}(S_t = s' \mid S_{t-1} = s)$$

be the one step transition probability from state s to s'. All transitions are probabilities, i.e

$$0 \leq p(s' \mid s) \leq 1, \qquad \forall s, s' \in \mathcal{S}$$

and the process will always find itself in a state

$$\sum_{s' \in \mathcal{S}} p(s' \mid s) = 1.$$

Further we denote the n-step transition probability from s to s' as $p_n(s' \mid s)$. To calculate the $n + m$ step we can use the Chapman-Kolmogorov equation

$$\begin{aligned} p_{m+n}(s \mid s') &= \mathbb{P}\{S_{m+n} = s' \mid S_0 = s\} \\ &= \sum_{z \in S} \mathbb{P}\{S_{m+n} = s', S_m = z \mid S_0 = s\} \\ &= \sum_{z \in S} p_m(s \mid z) p_n(z \mid s'), \end{aligned} \tag{1}$$

which says that the transition from state s to s', in $n + m$ steps, can be written in terms of the transition probability from s to all possible states z in n steps multiplied by the probability from these new states to s' in m steps.

2.2 Environment and actions

In the case of a simple Markov process, there is no way to interact with and influence the environment since one moves from state to state through the transition probabilities. Therefore a Markov process alone is not an adequate representation of the environments we need to solve real-life problems. For an environment to be complex enough to describe problems that we face in the real world, we need it to be interactable and change through interactions. To formalize this way of interacting with the environment, we make use of the MDPs, which is a method that enables the agent to influence the transition probabilities and thereby interact with the environment. This environment where the agent

moves around consists of states, actions, and rewards, where rewards are used to achieve a goal. The agent runs for a full episode, which will be the period that the agent is active. An episode could represent a game, for example a game of tic-tac-toe.

We will, in this section, consider only episodic environments, which means that there eventually will be a terminal absorbing state in which the Markov process stops. This means that we will have a point in time T, where the Markov process stops. If the process describes an intraday stock market, the terminal state could be the state at the time at which the stock market closes. If the state represents the positions in a game. Then the terminal state could be a position in which the game is over.

The actions that the agent can take may depend on the current state that the agent are in, and consists of all the possible actions at time $t = 0, 1, ..., T$, $a_t \in \mathcal{A}(s)$. We define the transition probabilities given actions in a MDP as

$$p(s'|s, a) = \mathbb{P}(S_t = s' \mid S_{t-1} = s, A_{t-1} = a), \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A}(s). \tag{2}$$

where the only difference from an ordinary Markov process is that transition probability also depends on the chosen action a. Given a state s and an action a at time t we will with certainty end be in a state again at time $t + 1$, i.e.

$$\sum_{s' \in S} p(s' \mid s, a) = 1, \ \forall s \in \mathcal{S}, a \in \mathcal{A}(s),$$

Since we only need to include the preceding state and action, we need the last state to include all information about the changes to the environment that affects the agent; otherwise we would not have the Markov property. If the agent would act differently whether it had information about all previous states and actions taken or just the last state and action, then we could not have the Markov property. Therefore we have to be careful when defining how the states are represented.

Note that the action space is dependent on the given state, since the actions that the agent are able to take can depend on the state. Though to spare notation we will from now denote the action space as $\mathcal{A}$. To describe how the states and actions can interact, consider a game of tic-tac-toe where the agent plays randomly with the given probabilities:

	1	2	3
1	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
2	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
3	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

When the game starts it is possible for the agent to place a piece wherever it wants, but after the first move the environment changes, lets say the first piece is placed in the upper left corner (1,1):

	1	2	3
1	×		
2			
3			

Then the agent has the action space of all positions except for the upper left corner (1,1). In the tic tac toe example, a terminal state is either a state with three X's or O's in a row/diagonal or where all places in the matrix are filled.

2.3 Rewards, Value Functions and Policies

In Markov decision problems rewards are used to evaluate the value of being in different states, or the value of selecting different actions given a state. For instance in the tic tac toe problem, a winning state having 3 pieces in a row is an optimal state and should be rewarded accordingly. We define the reward function $r(s, a)$, which defines the expected future reward being in state s taking action a.

$$\begin{aligned} r(s,a) &= \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] \\ &= \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a). \end{aligned} \tag{3}$$

where R_t is a stochastic variable representing the reward at time t and $\mathcal{R}$ is the space of all possible rewards and

$$p(s', r \mid s, a) = \mathbb{P}(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a)$$

is the transition probability from state s, by taking action a to state s' receiving reward r. We can write the transition formulation with rewards with regards to the transition including rewards as follows

$$p(s' \mid s, a) = \sum_{r \in \mathcal{R}} p(s', r \mid s, a). \tag{4}$$

Often we are not very interested in the immediate reward, but rather the future cumulative reward. For example in tic tac toe one could imagine a reward provided for connecting two pieces, but we would care much more about the future reward received when connecting three. Another obvious example is modelling returns in the financial markets, where investors often have a long investing horizon. We define the cumulative future rewards from time t until T as

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{5}$$

where $0 \leq \gamma < 1$ is a discount factor to ensure convergence. This is especially helpful, when an episode of the MDP is long. Note that the cumulative rewards has a recursive nature

$$\begin{aligned} G_t &= R_{t+1} + \gamma G_{t+1} \\ &= R_{t+1} + \gamma R_{t+2} + \gamma^2 G_{t+2} \\ &\vdots \end{aligned}$$

In the tic-tac-toe example the agent played randomly, however this is surely not the best strategy. To define how the agent acts we need a smart way to map from a given state

to a set of probabilities, from which the agent will choose an action to take in order to maximize its expected return. This mapping is defined as the *policy*, π. We define π to be a probability function that specifies the probability that we will take action $A_t = a$ given the state we are in $S_t = s$

$$\pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s).$$

Thus for a given policy the transition probability of going from state s to s' is given by

$$\mathbb{P}(S_t = s \mid S_{t-1} = s, \pi) = \sum_{a \in \mathcal{A}} \pi(a \mid s) p(s' \mid s, a)$$

where the only difference is that we take the probability weighted expectation over the action space.

To evaluate policies and their future reward Richard Bellman 1957 suggests the *state-value function*, $v_\pi(s)$. This function is the expectation of the future cumulative rewards given the current state following policy π:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_t = s] \end{aligned}$$

The first term is straight forward and follows from probability weighting the actions in eq. (3) by our policy:

$$\mathbb{E}_\pi[R_{t+1} \mid S_t = s] = \sum_{r, s', a} r \pi(a \mid s) p(s', r \mid a, s).$$

As for the second, see that by applying the law of total expectation we can further condition on the next state and thus get

$$\begin{aligned} \gamma \mathbb{E}_\pi[G_{t+1} \mid S_t = s] &= \gamma \mathbb{E}_\pi \Big[\mathbb{E}\left[G_{t+1} \mid S_{t+1} = s'\right] \;\Big|\; S_t = s \Big] \\ &= \sum_{r, s', a} \pi(a \mid s) p(s', r \mid a, s) \gamma \mathbb{E}\left[G_{t+1} \mid S_{t+1} = s'\right] \\ &= \sum_{r, s', a} \pi(a \mid s) p(s', r \mid a, s) \gamma v_\pi(s'). \end{aligned}$$

Finally we get

$$v_\pi(s) = \sum_{s', r, a} \pi(a \mid s) p(s', r \mid s, a)[r + \gamma v_\pi(s')], \qquad \forall\, s \in \mathcal{S} \tag{6}$$

Equation 6 is known as the bellman equation for $v_\pi(s)$. Note its recursive nature ($v_\pi(s)$ is a function of $v_\pi(s')$).

Instead of evaluating the value of a state, another intuitive objective is to evaluate the actions given a state. We let the *action-value function* express the expected return given a state s, taking action a, and thereafter following the policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

following the same steps as when deriving the state-value function we get

$$\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} \mid S_t = s, A_t = a] + \mathbb{E}_\pi[\gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r \mid s, a) r + \sum_{s',r} p(s', r \mid s, a) \gamma v_\pi(s') \\
&= \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')]. \qquad (7)
\end{aligned}$$

Clearly the value functions have a close relationship. We can write $v_\pi(s)$ as an expectation of $q_\pi(s,a)$ over the action space

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}_\pi\Big[\mathbb{E}_{\pi'}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \;\Big|\; S_t = s\Big] \\
&= \mathbb{E}_{\pi'}[q_\pi(S_t, A_t) \mid S_t = s] \qquad (8)
\end{aligned}$$

and by taking a probability weighted average over the action space we get

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) q_\pi(s,a) \qquad (9)$$

The State Distribution

We define the expected discounted number of time steps spent in s starting in state S_0, be given by

$$\begin{aligned}
\eta(s) &= \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t \mathbb{1}_{\{S_t = s\}} \mid S_0 = s_0\Big] \\
&= \sum_{t=0}^{\infty} \gamma^t \mathbb{P}_\pi(S_t = s \mid S_0 = s_0) \\
&= \sum_{t=0}^{\infty} \gamma^t p_t(s \mid s_0, \pi) \qquad (10)
\end{aligned}$$

where $p_t(s \mid s_0, \pi)$ is the t-step transition probability following π. Recall that γ is the discount factor. It can be seen as a form of termination, which eventually occur. And thus η(s) is the discount weighted expected number of time steps visiting s, under policy

π, starting in state S_0. We define the state distribution of s under π as the expected fraction of time spent in s following π given by

$$\mu_\pi(s) = \frac{\eta(s)}{\sum_{s \in \mathcal{S}} \eta(s)}.$$

This is also known as *the on-policy distribution*. Recall that the state space is assumed finite.

2.4 Putting it all together

Finally we combine all the elements in the tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$, which defines the Markov Decision Process.

Definition 2.2 (Markov Decision Process). A Markov Decision Process is a tuple $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$

- $\mathcal{S}$ is a set of states.
- $\mathcal{A}$ is a set of actions.
- p is a transition probability function.
- r is a reward function.
- γ is a discount factor.

Figure 1 illustrates the MDP dynamics between the agent and the environment:

observe → take action → receive reward+observe → take action → receive reward+observe....

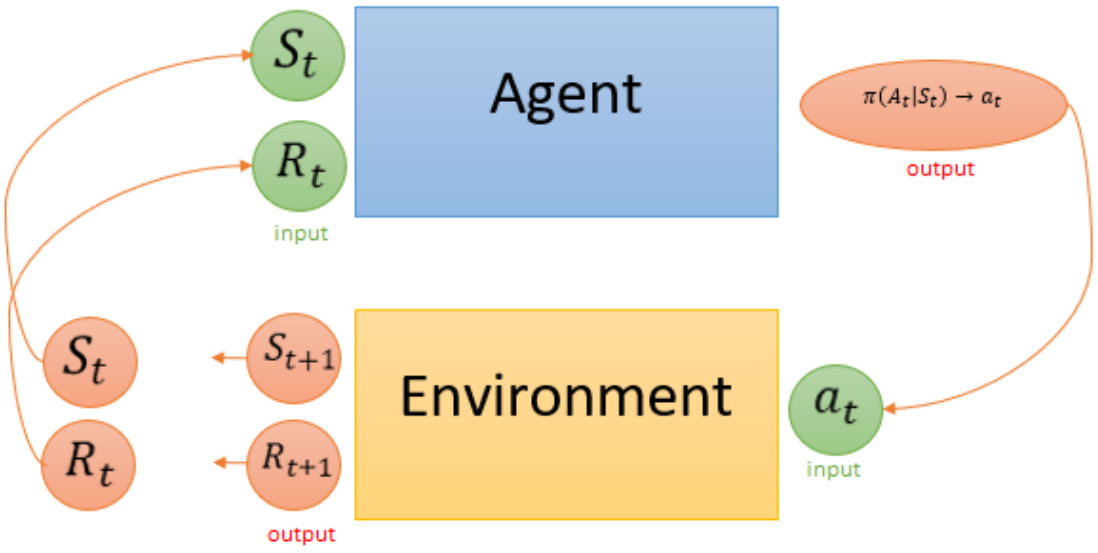


Figure 1: *Illustration of MDP dynamics. Green illustrates input, and red illustrates output. Agent observe and interacts through actions which affect the environment, which provides feedback in rewards, and updates the state, which the agent observe ....*

3 Solution Methods

The goal of RL is for the agent to achieve the maximum possible reward during a given time frame. This is done by finding an *optimal policy* maximizing the value functions. Bellman explain the definition in his principle of optimality.

PRINCIPAL OF OPTIMALITY. (Bellman 1957) *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

A policy π is considered a better policy than π' when:

$$v_{\pi'}(s) \leq v_{\pi}(s), \forall s \in \mathcal{S}$$

And all optimal policies $\pi_*(a \mid s)$ satisfies

$$v_{\pi^*}(s) = \max_{\pi} v_{\pi}(s)$$

and

$$q_{\pi_*}(s, a) = \max_{\pi} q_{\pi}(s, a)$$

In this section we present three fundamental approaches to solve the Bellman Equation (eq. 6) in order to find optimal policies.

3.1 Dynamic Programming

Dynamic programming, also introduced by Richard Bellman, is a method to find solutions to the recursive value function by splitting the problem into sub problems, which is done by approximating the value functions through updated expectations. The process assumes that the problem is well defined (full knowledge of the MDP). The idea is the same as in the shortest path algorithm: split the problem into sub problems and solve them one at a time.

The first objective when finding an optimal policy is being able to estimate the state-value function given a policy. The simplest method to do so is the *Iterative Policy Evaluation* algorithm, shown in algorithm 1. It simply computes and updates the value function for each step in time and by adding up reward. As $k \to \infty$ the algorithm converges to the true $v_{\pi}(s)$. The convergence is guaranteed and easily reasoned by the discount factor γ: the more iterations into the future the harder discounting through γ and thus the less the value function is affected by the reward.

Algorithm 1: Iterative Policy Evaluation

Input:
- π: policy
- θ: accuracy threshold parameter
- $\hat{v}_0(s) = 0$

Output: Estimate of value function vector $\hat{v}_\pi(s),\ \forall s \in \mathcal{S}$.

Initialize:
$k = 0$
repeat
  $\Delta(s) \leftarrow 0,\ \forall s \in \mathcal{S}$
  for $s \in \mathcal{S}$ **do**
    $\hat{v}_{k+1}(s) \leftarrow \sum_{s',r,a} \pi(a \mid s) p(s', r \mid s, a)[r + \gamma \hat{v}_k(s')]$
    $\Delta(s) \leftarrow \max(\Delta, |\hat{v}_k(s)_{k+1} - \hat{v}_k(s)|)$
  end
  $k \leftarrow k + 1$
until $\Delta(s) < \theta,\ \forall s \in \mathcal{S}$;

In theory we are now able to find an optimal policy brute forcing through all possible policies and finding the one maximizing our value function. Though a simpler way is to update our policy in each iteration choosing the policy maximizing the probability of entering the state with the highest value. This method is called *Policy Improvement* and specifically updates the policy as the greedy policy π'

$$\pi'(s) \leftarrow \arg\max_a q_\pi(s, a),$$

where $\pi'(s) = a$ denotes a deterministic policy, meaning that policy π' takes action a with probability equal to 1 given state s. Its value is at least as large as if any other policies were followed, for all s, over one step:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s).$$

Thus it improves the state-value function

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) = \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + ... \mid S_t = s] = v_{\pi'}(s) \end{aligned}$$

Combining the policy evaluation and improvement methods we can now construct the *policy iteration* algorithm, shown in algorithm 2 and converges to the optimal policy π_*.

Algorithm 2: Policy Iteration

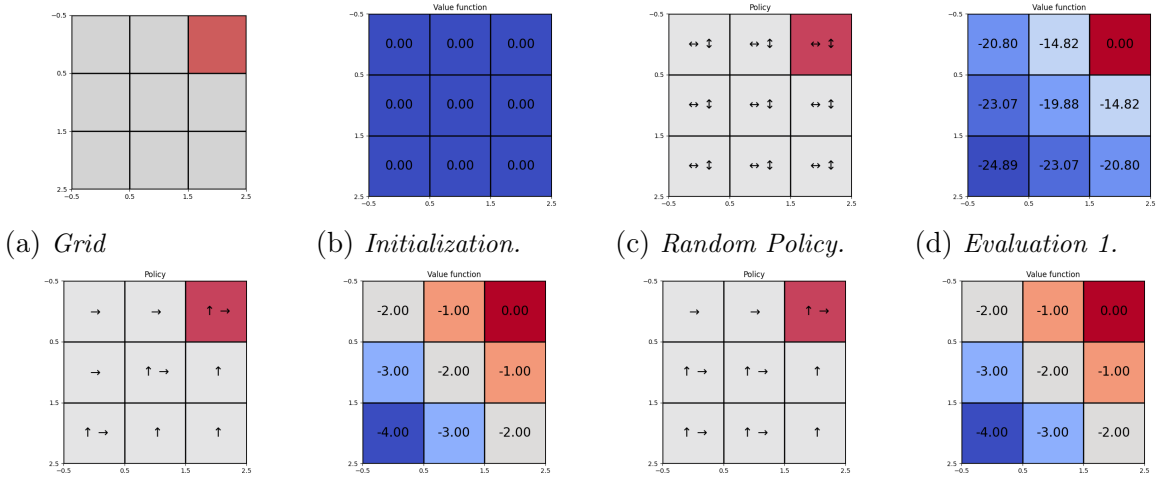
Input: π^0: policy, $v^0_{\pi^0}$: value function
Output: Optimal policy π_*, state-value function v_*

1. Policy Evaluation
e.g. algorithm 1

2. Policy Improvement
Policy_ stable = True
for $s \in \mathcal{S}$ **do**
  $\pi_{\text{old}} \leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a q_\pi(s, a)$
  if $v_{\pi_{old}}(s) \neq v_\pi(s)$ **then**
    Policy_ stable = False
  end
end
if *Policy_ stable* **then**
  Return $v_\pi(s) \approx v_*, \;\; \pi \approx \pi_*$
else
  Go to 1.
end

Figure 2, illustrates the policy iteration algorithm solving a simple grid-world example. It has a cost of 1 to take a step from a given state (square) to another. The top right corner is a terminal state, which ends (wins) the game. We start by setting up the grid (a) and evaluates all states as 0 (b). A random policy is then chosen (c) and being evaluated (d). Policy is improved (e) and evaluated (f), same again (g) and (h), with no change to value function, so algorithm has converged and stops.

Another method for finding optimal policies is to work directly on the value function and maximize it with respect to the possible actions. This is known as *value iteration* and shown in algorithm 3, which iteratively maximizes the Bellman equation and stops when the difference in state-value function between two iterations is small, returning the policy achieving the latest state-value function. The key difference to policy iteration is expressed in the wording. Policy iteration evaluates policies and the improve upon these, while value iteration evaluates the value of each action from a state and then updates the corresponding state-value function.



(a) *Grid* (b) *Initialization.* (c) *Random Policy.* (d) *Evaluation 1.*

(e) *Improvement 1.* (f) *Evaluation 2.* (g) *Improvement 2.* (h) *Evaluation 3.*

Figure 2: *Grid world example. We start by setting up the grid (a) and evaluates all states as 0 (b). A random policy is then chosen (c) and being evaluated (d). Policy is improved (e) and evaluated (f), same again (g) and (h), with no change to value function, so algorithm has converged and stops. The squares are colored using a heat map, such the the red colors presents high value states, and blue represents low.*

Algorithm 3: Value Iteration

Input: $v(s)$: value function, θ: accuracy threshold parameter.
Output: $\pi \approx \pi_*$

repeat
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v_{k+1}(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_k(s')]$
    $\Delta \leftarrow \max(\Delta, |v_{k+1}(s) - v_k(s')|)$
  end
  $k \leftarrow k + 1$
until $\Delta < \theta$;

3.2 Monte Carlo methods

Sometimes the environment is not known before the agent moves into it, which makes it so that we cannot use Dynamic Programming. Here Monte Carlo methods offer a range of methods to estimate the value functions and find the optimal policies, without knowing the environment in advance. These algorithms rely on random sampling, which in RL frameworks are sampling of states, actions and rewards. This can either be through live experience, where the agent is running in real time or in a simulated setup. In Monte Carlo methods we do not need full knowledge of the MDP, but we need a model that can generate samples, from which returns from different states can be calculated to evaluate a given a policy π. The Monte Carlo method is simply an iterative mean update after

each simulated episode. Let $G_k(s)$ be the cumulative return of the k-th visit to state s, we then write the Monte Carlo update as

$$\begin{aligned}\hat{v}_\pi(s)_k &\leftarrow \frac{1}{k}\sum_{j=0}^{k} G_j(s) \\ &= \frac{1}{k}\Big(G_k(s) + \sum_{j=0}^{k-1} G_j(s)\Big) \\ &= \hat{v}_\pi(s)_{k-1} + \frac{1}{k}\Big(G_k(s) - \hat{v}_\pi(s)_{k-1}\Big).\end{aligned}$$

Note that we in the last equation subtract a fraction of the previous state-value estimate and add a corresponding fraction of the new observed value of the same state. The estimates we get at the different states are independent and we can therefore start estimating the value functions from any given starting state. The general Monte Carlo algorithm for estimating the value function is illustrated in Algorithm 4.

Algorithm 4: Monte Carlo Algorithm to estimate v_π

Input:
- π: policy
- $\hat{v}_\pi(s)$: state-value function
- $n(s)$: vector to count number of times we have been in s.

Output: Approximate state-value function $\hat{v}_\pi(s)$.

Initialize:
$n(s) = 0,\ \forall s$
repeat
  Simulate an episode using π
  for $s \in \mathcal{S}$ **do**
    $n(s) \leftarrow n(s) + 1$
    $\hat{v}_\pi(s) \leftarrow \hat{v}_\pi(s) + \frac{1}{n(s)}\left(G(s) - \hat{v}_\pi(s)\right)$
  end
until *Desired number of episodes has been run*;

Monte Carlo methods allows us to estimate state-value functions and is seen as 'model-free' because we do not model the underlying transition probabilities of the MDP. Regarding the action-value function we can sample state-action pairs and thus estimate the action-value function, $\hat{q}_\pi(s, a)$. We can then make a deterministic policy that chooses the action that leads to the highest future cumulative reward. Though when doing so, we are facing some issues because we are following a deterministic policy, and thus we are not going to experience all the state-action pairs. Therefore we need to find a way such that our policy sometimes allow us to pick the less-favored actions to make sure that we actually find the optimal policy. This dilemma between choosing high value actions and testing unknown or low value actions is known as the *exploration exploitation trade-off*. To deal with this we split RL algorithms into two families; *on-policy* and *off-policy*. On-policy is a method where we improve the policy that we also use in the episodes to decide which actions to take, where off-policy improves another policy that we do not use through the episodes.

In on-policy methods we ensure that

$$\pi(a \mid s) > 0, \forall s \in \mathcal{S}, a \in \mathcal{A},$$

which allows us to choose all the given actions in a given state, although there should be a higher probability to choose the action with the most expected reward than the others. One method is using $\epsilon-$greedy policies, where we choose a non-optimal action with probability ϵ. Assuming that there only is one action a^* yielding the highest expected reward (if there is more, a random of those can be chosen), we have

$$\pi(a \mid s) = \frac{\epsilon}{|\mathcal{A}| - 1}, \forall s \in \mathcal{S},\ a \in \mathcal{A} \backslash a^*.$$

The greedy action then has

$$\pi(a^* \mid s) = 1 - \epsilon, \forall s \in \mathcal{S}$$

Since we are assigning probability mass to the actions that are not considered the most favorable actions, these $\epsilon-$greedy policies will be worse the more we increase ϵ. Despite from being a constant, ϵ can also defined as a function of the number of actions taken, such that it goes to 0 when the number of actions taken goes to ∞. An example of an on-policy algorithm to estimate the action-value function can be seen in Algorithm 5.

Algorithm 5: Monte Carlo Algorithm to estimate π and $\hat{q}(s, a)$ on-policy

Input: π: initial policy, Q: action-value function, R: return array, $n(s, a)$ vector to count number of times we have been in s and taken action a

Output: Policy $\pi(a \mid s)$ and action-value function $\hat{q}(s, a)$.

repeat
  Simulate an episode, E, using π
  `/* Action-value function` $\hat{q}(s, a)$ `update */`
  for s, a *in* E **do**
    $G =$ cummulated return after state s and action a
    $n(s, a) += 1$
    $\hat{q}(s, a) = \hat{q}(s, a) + \frac{1}{n(s,a)} (G - \hat{q}(s, a))$
  end
  `/* Policy` $\pi(a \mid s)$ `update */`
  for s *in* E **do**
    $a^* = \underset{a}{\text{argmax}}\ \hat{q}(s, a)$
    for *For all* $a \in \mathcal{A}(s)$ **do**
      $$\pi(a \mid s) = \begin{cases} 1 - \epsilon & \text{if } a = a* \\ \frac{\epsilon}{|\mathcal{A}|-1} & \text{if } a \neq a^* \end{cases}$$
    end
  end
until;

In the off-policy methods we use two policies: the one we know from earlier, π, which we

will call the *target-policy* and a *behaviour-policy*, b. The behaviour-policy is the policy we use during episodes and use to estimate the target-policy. Because we use b to estimate π we need to make sure that all the states and actions that we encounter in π is also encountered in b. Here we can keep b ϵ-greedy while having π as a deterministic policy which simply chooses the optimal actions.

To estimate the value function under π we encounter a problem, because all the observed rewards are achieved following b. I.e. we wish to calculate

$$\hat{v}_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s],$$

but only know

$$\hat{v}_b(s) = \mathbb{E}_b[G_t \mid S_t = s],$$

because G_t is achieved from following b. Luckily we can use a method called *importance sampling*, which is a method where we estimate the expected returns following π relative to following b. In a general setting where p and q are two distributions, the idea is that

$$\begin{aligned}
\mathbb{E}_p[f(X)] &= \sum_{i=1}^{N} p(x_i) f(x_i) \\
&= \sum_{i=1}^{N} q(x_i) \frac{p(x_i)}{q(x_i)} f(x_i) \\
&= \mathbb{E}_q\Big[\frac{p(X)}{q(X)} f(X)\Big].
\end{aligned}$$

so that using Monte Carlo we can compute

$$\mathbb{E}_q\Big[\frac{p(X)}{q(X)} f(X)\Big] = \frac{1}{N} \sum_{i=1}^{N} \frac{p(x_i)}{q(x_i)} f(x_i), \quad x_i \sim q(x). \tag{11}$$

The ratio between the two distributions q and p is called the *importance sampling ratio*. Back to the RL setting we let S_t be the state at time t. To calculate the probability of a given sequence of state-action pairs from time t to T, we can multiply our target policy π, with the transition probabilities:

$$\prod_{k=t}^{T-1} \pi(A_k \mid S_k) p(S_{k+1} \mid S_k, A_k),$$

which is dependent on the transition probabilities. However the ratio between the probabilities given the two policies are independent on the transition probabilities:

$$\begin{aligned}
\rho_t^T &= \frac{\prod_{k=t}^{T-1} \pi(A_k \mid S_k) p(S_{k+1} \mid S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k \mid S_k) p(S_{k+1} \mid S_k, A_k)} \\
&= \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.
\end{aligned}$$

because they cancel out. ρ_t^T is the importance sampling ratio. It represents the ratio between the probability that we get a sequence of state-action pairs given the two different policies. It tells us the likelihood of selecting an action a following π relative to following b. To calculate the state-value function we define a set $\mathcal{J}(s)$ that keeps track of all the time steps, t, that we have entered a given state s. Furthermore we let $T(t)$ keep track of the time step of termination after being at time t. Using these we can estimate the state-value function following π by multiplying the reward following b with the importance sampling ratio:

$$\hat{v}_\pi(s) = \frac{\sum_{t\in\mathcal{J}(s)} \rho_t^{T(t)} G_t}{|\mathcal{J}(s)|},$$

where G_t is the return we gained from t to $T(t)$ following b, and $|\mathcal{J}(s)|$ is the number of times we have been in s (equal to the number of time steps we have entered s). This way of estimating $\hat{v}$ is called the *ordinary importance sampling* method. It follows from eq. (11) that it is an unbiased estimate. Though it has potentially very large variances because there is no bound on the importance sampling ratio. Instead of computing the average by dividing with the number of times we have been in s, a solution is to calculate the weighted average with respect to the sampling ratios

$$\hat{v}_\pi(s) = \frac{\sum_{t\in\mathcal{J}(s)} \rho_t^{T(t)} G_t}{\sum_{t\in\mathcal{J}(s)} \rho_t^{T(t)}}.$$

This adds bias to our estimate, but lowers the variance.

3.3 Temporal Difference

The RL problem deviates from classical ML problems in the fact that temporality matters. Temporality refers to time or the sequence in which we observe a MDP. Temporal difference (TD) is a method, which mixtures Dynamic programming and Monte Carlo. It utilizes the resampling aspect of Monte Carlo and the value function estimation of Dynamic Programming. Just as with Monte Carlo we in TD use an incremental mean update, though instead of estimating the mean after a full sampled episode we estimate (bootstrap) the expected future reward in an episode on an ongoing basis. TD agent's approach to learning during episodes is known as *online learning*, and its ability to do so is one of the reasons why RL has gained its popularity. In general we see the TD problem as an update with respect to a fraction α of a *TD error*. The simplest version is a one-step look ahead update known as $TD(0)$:

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha \underbrace{\left[\underbrace{R_{t+1} + \gamma\hat{v}(S_{t+1})}_{\text{TD target}} - \hat{v}(S_t)\right]}_{\text{TD error}}.$$

Thus the *TD target* in TD(0) is an estimate of the future value function at time $t+1$. We will refer to the TD error by δ_t, i.e

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha\delta_t,$$

where

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}) - \hat{v}(S_t)$$

By learning online and estimating $\hat{v}(S_t)$ continuously we add bias to our estimates, unless our estimate of the value function is the actual true value function, whereas Monte Carlo updates after full episodes creates unbiased estimates. By only updating with a single step into the future the noise of our updates becomes much smaller and thus the variances of our estimates becomes smaller. Instead of only looking one step ahead we could also look n steps ahead before updating our estimate and this way find a balance in the trade-off between bias and variance. Even though we add bias to our estimate the TD(0) algorithm still converges to the true value function (S. R. Sutton and Barto 2018).

An intuitive alternative approach is estimating the action value function. Using TD(0) this is known as the on-policy *Sarsa* algorithm:

$$\hat{q}_\pi(S_t, A_t) = \hat{q}_\pi(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \hat{q}_\pi(S_{t+1}, A_{t+1}) - \hat{q}_\pi(S_t, A_t) \right].$$

The sequence of random variables appearing in the update: $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$, is what the algorithm is named after.

One of the most famous RL algorithms is *Q-learning*. Q-learning is an off-policy TD algorithm, which approximates the action value function by estimating q_* one step ahead:

$$\hat{q}_\pi(S_t, A_t) = \hat{q}_\pi(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a (\hat{q}_\pi(S_{t+1}, a)) - \hat{q}_\pi(S_t, A_t) \right]$$

3.4 Ergodic Markov Decision Processes

We have considered cases where MDP's comes in episodes, and shown that if the episodes are long, using discounting helps. But when MDP's are continuous, e.g. we play a game without an ending it turns out there is a more efficient way of modelling rewards: *the average reward setting*. This could for instance represent a financial market, which never closes, e.g. the Foreign Exchange Market. In the average reward setting we evaluate rewards as the difference between the instantaneous reward and the average reward:

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + \dots$$

where $r(\pi)$ is the average reward of the followed policy π, which is given by

$$\begin{aligned} r(\pi) &= \lim_{h \to \infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}_\pi[R_t \mid s_0] \\ &= \lim_{t \to \infty} \mathbb{E}_\pi[R_t \mid S_0] \\ &= \sum_s \mu_\pi(s) \sum_{s',r,a} \pi(a \mid s) p(s', r \mid s, a) r \end{aligned} \tag{12}$$

where

$$\mu_\pi(s) = \lim_{t \to \infty} \mathbb{P}_\pi(S_t = s)$$

is the steady state or stationary distribution following π. The derivation holds if the MDP is *ergodic*. In an ergodic MDP a stationary distribution exists independent of S_0. I.e. the probability of transitioning from a state to another in x steps converges to a distribution, which is independent of the starting distribution and does not change as long as the same policy is being followed.

In the derivation of the state-value function in the average reward setting, we use the same steps as deriving the bellman equation earlier, eq. (6), where the only difference is the subtraction of $r(\pi)$ and the removal of the discount factor γ:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r - r(\pi) + v_\pi(s')], \qquad \forall\, s \in \mathcal{S}$$

Likewise our TD targets gets adjusted. In the average reward setting we have

$$\delta = R_{t+1} - \hat{r}(\pi) + \hat{v}(S_{t+1}) - \hat{v}(S_t)$$

Finally the main reason for choosing the average reward setting, is because using discounted rewards in a continuous setting simply happens to be a scalar of the average reward. This is shown in *the futility of discounting in continuing problems.*

The Futillity of Discounting in Continuing Problems. (S. R. Sutton and Barto 2018) *Discounting can be saved by choosing an objective that sums discounted values over the distribution with which states occur under the policy.*

Argument. Let $J(\pi)$ denote the performance measured in reward following π.

$$\begin{aligned}
J(\pi) &= \sum_s \mu_\pi(s) v_\pi(s) \\
&= \sum_s \mu_\pi(s) \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')] && \text{(Bellman, eq.6)} \\
&= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \gamma v_\pi(s') && \text{(def. } r(\pi), \text{eq. 12)} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi(s') \sum_s \mu_\pi(s) \sum_a \pi(a \mid s) p(s' \mid s, a) && \text{(def. } p(s' \mid s, a))
\end{aligned}$$

and since $\sum_s \mu_\pi(s) \sum_a \pi(a \mid s) p(s' \mid s, a) = \mu_\pi(s')$, we get

$$\begin{aligned}
&= r(\pi) + \gamma \sum_{s'} v_\pi(s') \mu_\pi(s') \\
&= r(\pi) + \gamma J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 r(\pi) + ... \\
&= \frac{1}{1-\gamma} r(\pi) && \text{(sum geo. series)}
\end{aligned}$$

Because the rewards in the discounted setting just becomes scalar of those in the average setting, the discount factor has no influence.

4 Generalized Solution Methods

So far we have focused on finding solutions to the value functions for each and every unique (state, action)-pair provided in a given MDP. In practice the state and action spaces are often large, sometimes continuous and therefore infinite. Thus the problems will be computationally difficult or even impossible to solve. The solution is to find a way of generalizing our estimate of the value function, even to scenarios not encountered in the MDP before. In this section we show how this is done by using function approximators.

4.1 Function approximation

We let $\hat{f}$ denote a model or function approximating another unknown function f. There are endless ways of approximating functions, some common known are linear models, decision trees and neural networks. We will focus on parametric methods, where the model takes vector of parameters as input and utilizes them to output an approximation of the desired unknown function.

4.1.1 Value Function approximation

In RL, two key unknown functions are the value functions $v_\pi(s)$ and $q_\pi(s,a)$. Let $v_\pi(s)$ and $q_\pi(s,a)$ represent the true state- and action-value functions. Then the idea is to find approximators $\hat{v}_\pi(s,\mathbf{w})$ and $\hat{q}_\pi(s,a,\mathbf{w})$ such that

$$\hat{v}_\pi(s,\mathbf{w}) \approx v_\pi(s) \qquad \wedge \qquad \hat{q}_\pi(s,a,\mathbf{w}) \approx q_\pi(s,a).$$

Here $\mathbf{w} \in \mathbb{R}^d$ stands for weights and represents an unknown parameter vector estimated using data from the given state (and chosen action for $\hat{q}_\pi(s,a,\mathbf{w})$). From the data provided in a state, we extract a state feature vector:

$$\mathbf{x}(s) = \begin{bmatrix} \mathbf{x}_1(s) \\ \mathbf{x}_2(s) \\ \vdots \\ \mathbf{x}_{\|\mathbf{x}\|}(s) \end{bmatrix},$$

where in the linear case $\|\mathbf{x}\| = d$. The feature vector is basically a design matrix with one row dependent on the current sate. This way of looking at the problem does not change anything dramatically, actually all the methods introduced in section 3 can be modelled this way with the feature vector having as many features as the MDP has states. In this case, when modeling $\hat{v}_\pi(s,\mathbf{w})$, the features would be binary each representing a state, such that each state could potentially gain their unique value - just as in section 3. I.e

the feature vector would be of the form:

$$\mathbf{x}(S = s_j) = \begin{bmatrix} \mathbb{1}\{S = s_1\} \\ \vdots \\ \mathbb{1}\{S = s_{j-1}\} \\ \mathbb{1}\{S = s_j\} \\ \mathbb{1}\{S = s_{j+1}\} \\ \vdots \\ \mathbb{1}\{S = s_{|\mathbb{S}|}\} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

and the function approximator could be a linear model

$$\hat{v}_\pi(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}.$$

For obvious reason, all-ready mentioned, we do not desire to include features for each state in most practical applications. The aim is to select a set of features and find a function approximator $f(s, \mathbf{w})$ of our value functions that generalizes well, while describing enough information about the individual states to distinguish them from each other.

4.1.2 Policy approximation

Until now we have solely presented methods to solve MDPs by estimating the true value functions and then make decisions based on these. Policy methods is another class of RL algorithms, which attempts to learn the optimal policy directly. This is done by making a parameterized function of the policy, such that

$$\pi(a \mid s, \boldsymbol{\theta}) = \pi_{\boldsymbol{\theta}}(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}),$$

which again could be any function, though often neural networks are a popular choice. $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ represents the parameter vector of the learned policy, while we keep $\mathbf{w}$ for the weights of the learned value functions. We keep the parameter vectors separated because the more advanced methods, which we present in 4.4, utilizes function approximations for both policy and value functions to enhance performance and convergence.

In discrete action spaces the *softmax function* is often used to output a probability equivalent to the exponential weight

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_a e^{h(s,a,\boldsymbol{\theta})}}$$

where $h(s, a, \theta)$ is some function used to approximate the optimal policy, e.g. linear model or neural network.

One clear advantage that policy methods bring is its possibility learn stochastic policies. In a game like rock-paper-scissors the only optimal policy is a random uniform distribution between rock, paper and scissors. Any strategy, which leans towards a specific action will easily be learned and exploited. Another advantage is that policy methods works well in continuous action spaces, which we will show in section 4.3.3. A disadvantage on the other hand is that policy methods often converge to local instead of global optimum. Another disadvantage is that evaluating a policy is often inefficient and with high variance, though in section 4.4 we introduce some methods to help with this issue.

4.2 Deep Reinforcement Learning

Deep reinforcement learning is a popular subgroup of RL algorithms, where the function approximator is a *deep neural network*. In part II and III of this thesis, where we test RL algorithms for trading purposes, deep neural networks are our function approximators of choice. We will only focus on feed-forward neural networks, even though network types such as Convolutional neural networks and Recurrent neural networks also have its use cases in RL.

Neural networks attempts to replicate some of the mechanisms in the brain of human beings, by mathematically connecting data points like the network of neurons in the brain, and is one of the building blocks of modern artificial intelligence. Like any other statistical model neural networks are used as function approximators attempting to map an input to an output (e.g. $X \to Y$). Actually neural networks are build upon the linear model and a traditional feed forward neural network has the output form/layer:

$$f(X) = \beta_0 + \beta^\top Z,$$

where

$$Z \in \mathbb{R}^M, \qquad Z_m = \sigma(\alpha_{0m} + \alpha_m^\top X), \ \forall m \in 1, ...M.$$

Z represents a *hidden layer* in the neural network with elements Z_m, called neurons. The *activation function* $\sigma(x)$ maps x to a given range depending on the chosen function, and can be different from layer to layer. Usually it maps an input to output in a limited range, e.g. $x \in \mathbb{R} \to Y \in]-1, 1[$. For example the sign, sigmoid or ReLU functions:

$$\sigma(x) = \text{sign}\,(x) \qquad \vee \qquad \sigma(x) = \frac{1}{1+e^{-x}} \qquad \vee \qquad \sigma(x) = \max(0, x)$$

In case the output has a specific form, for instance a probability or a non-negative value, it is a common practice to apply an activation function in the output-layer, such that

$$f(X) = \sigma(\beta_0 + \beta^\top Z).$$

To further develop a neural network a common extension is to add additional 'hidden layers' ($Z's$) and feed them into each other, replacing X after first iteration, i.e:

$$f(X) = \beta_0 + \beta^\top Z^D,$$

where

$$\begin{aligned} Z_m^D &= \sigma_D(\alpha_{D0m} + \alpha_{Dm}^\top Z^{D-1}), && \forall m \in 1, ...M_D \\ Z_m^1 &= \sigma_1(\alpha_{10m} + \alpha_{1m}^\top X), && \forall m \in 1, ...M_1. \end{aligned}$$

A neural network with more than one hidden layer ($D > 1$) is called a *deep neural network* and is often referred to as deep learning. We will generally refer to neural works covering both regular neural networks, and deep neural networks. Figure 3 illustrates a

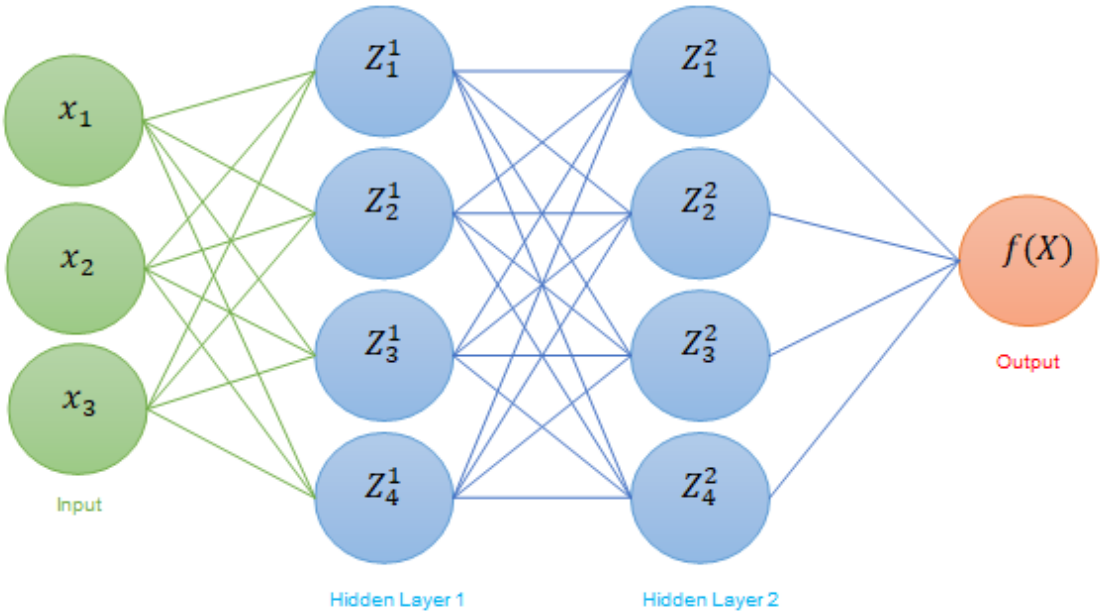


Figure 3: *Illustration of a neural network with two hidden layers. The green circles represent the inputted feature vector and the blue circles represent the neurons of the network. The 2 times 4 stacked set of neurons represents two hidden layers. The final red circles represents the output layer.*

deep neural network with input $X \in R^3$, two hidden layers $Z^1, Z^2 \in R^4$ and an output $f(X) \in R$. We will only work with *dense* layers, which are layers that are fully connected to each other, but there are other type of networks that work with layers that are not fully connected.

One thing to keep in mind, when using neural networks is that due to its complex structure, it is often impossible to interpret the relationship between the inputs and outputs. It is a "black box" model. Thus if the modeller has a priori knowledge about how the inputs and outputs are related, or if it is important for the modeller to gain information about the relation. Then it is advised to use a model which reflects, or can reflect this relation, rather than using neural networks. Or for the first case at least attempt to do so before testing neural networks. Note that because of this, when we use neural networks later, we will not be analyzing variable importance, when estimating $v_\pi(s, \mathbf{w}), q_\pi(s, a, \mathbf{w})$ or $\pi_{\boldsymbol{\theta}}(a \mid s)$. This would have been possible if e.g. linear models were used. This is a sacrifice, which often is necessary, because the linear models are not complex enough to capture the structure of the value and policy functions. As a consequence of complexity neural networks are prone to overfitting, but there are various ways that we can try to prevent this. We can use shrinkage methods, here two popular ones are the L1- and L2-Regularization, also known as Lasso and Ridge. L2 regularization is a method where we punish the loss function with the sum of the squared weights, θ, where θ is our parameter vector

$$\hat{\mathcal{L}}(\theta) = \frac{\lambda}{2} \sum_{w \in \theta} w^2 + \mathcal{L}(\theta),$$

where λ determines the amount of regularization that we use. This form of regularization pushes the weights towards zero. On the other hand we have the L1 regularization that allow the weights to become zero. This is done by penalizing the sum of absolute weights:

$$\hat{\mathcal{L}}(\theta) = \lambda \sum_{w \in \theta} |w| + \mathcal{L}(\theta).$$

Another famous and very powerful method to prevent overfitting is *dropout* (Srivastava et al. 2014). Dropout is a very simple approach where we assign a probability, p_{dropout}, to each neuron that it will be omitted. This can either be done to specific layer or to all layers in the network. Formally the dropout procedure is done by sampling M_l (where l represents the given layer) number of Bernoulli random variables with probability p_{dropout} and multiply these with the weights in the given layer:

$$B \sim \text{Bernoulli}(p_{\text{dropout}})$$
$$\alpha = \alpha \odot B.$$

Note that this is only done during training, and when actually predicting we utilize the full neural network.

Figure 4 shows an example neural network, where dropout has dropped out 3 neurons marked with dark blue.

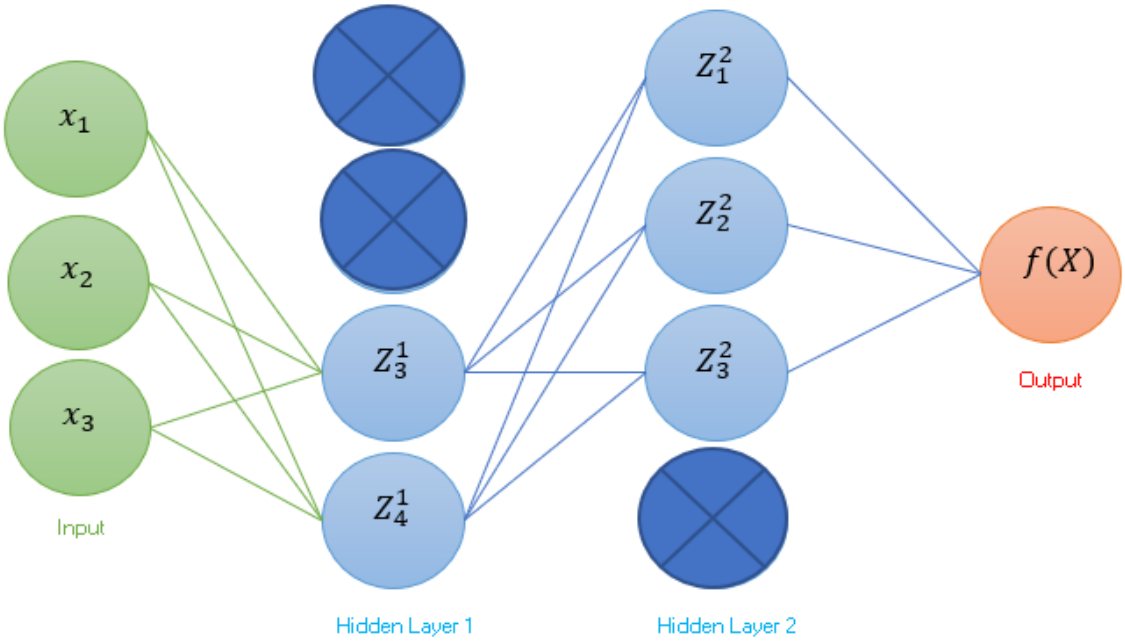


Figure 4: *Illustration of how a neural network could be with dropout. The dark blue circles with crosses represents the neurons that are dropped out.*

4.3 Estimation

Just like any other statistical problem we would like to estimate the parameters $\mathbf{w}_*$, such that $\hat{f}(X, \mathbf{w}_*)$ is as close to the true function $f(X)$ as possible, where 'close' is defined by an objective function, often a loss-function measuring the distance from $\hat{f}(X, \mathbf{w}_*)$ to $f(X)$.

4.3.1 The Weight Vector

When estimating the state-value function a natural objective is to minimize the loss function *mean squared value error* by following a policy. The mean squared value error is

$$\begin{aligned}\overline{\mathbf{VE}}(\mathbf{w}) &= \mathbb{E}_\pi\Big[\Big(v_\pi(S) - \hat{v}_\pi(S, \mathbf{w})\Big)^2\Big] \\ &= \sum_{s\in\mathcal{S}} \mu_\pi(s)[v_\pi(s) - \hat{v}_\pi(s, \mathbf{w})]^2.\end{aligned}$$

Recall that $\mu_\pi(s) \geq 0$, $\sum_s \mu_\pi(s) = 1$ is a state distribution under π, which in practice usually is the fraction of time spent in s. The distribution is used to weight the different states according to their frequency in the minimization problem. $v_\pi(s)$ represents the true state-value function, which we obviously do not know. Therefore we substitute in approximations of it in its place. Using Monte Carlo methods we run through an entire episode and therefore have knowledge of the future cumulative reward and in TD(0) we use the TD target. In Monte Carlo we use the *mean square return error*:

$$\begin{aligned}\overline{\mathbf{RE}}(\mathbf{w}) &= \mathbb{E}_\pi[(G_t - \hat{v}_\pi(S_t, \mathbf{w}))^2] \\ &= \sum_{s\in\mathcal{S}} \mu(s)\Big(G_t - \hat{v}_\pi(s, \mathbf{w})\Big)^2 \\ &= \sum_{s\in\mathcal{S}} \mu(s)\Big([G_t - v_\pi(s)] + [v_\pi(s) - \hat{v}_\pi(s, \mathbf{w})]\Big)^2 \\ &= \sum_{s\in\mathcal{S}} \mu(s)\Big(G_t - v_\pi(s)]^2 + [v_\pi(s) - \hat{v}_\pi(s, \mathbf{w})]^2 + 2(G_t - v_\pi(s))(v_\pi(s) - \hat{v}_\pi(s, \mathbf{w}))\Big) \\ &= \sum_{s\in\mathcal{S}} \mu(s)\Big([G_t - v_\pi(s)]^2 + [v_\pi(s) - \hat{v}_\pi(s, \mathbf{w})]^2\Big)\end{aligned}$$

The double product term is zero, because

$$\mathbb{E}_\pi\left[\Big((G_t - v_\pi(S_t)\Big)\Big(v_\pi(S_t) - \hat{v}(S_t, \mathbf{w})\Big)\right] = \mathbb{E}_\pi\Big[(\mathbb{E}_\pi[G_t - v_\pi(S_t) \mid S_t])(v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}))\Big]$$

where the right hand side is zero because $\mathbb{E}_\pi[G_t|S_t] = v_\pi(S_t)$. Thus we finally get

$$\overline{\mathbf{RE}}(\mathbf{w}) = \overline{\mathbf{VE}}(\mathbf{w}) + \mathbb{E}_\pi[(G_t - v_\pi(S_t))^2].$$

In TD(0) we use the *mean square TD error*:

$$\begin{aligned}\overline{\mathbf{TDE}}(\mathbf{w}) &= \sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}_\pi\Big[\Big(\underbrace{R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)}_{\text{TD target}} - \hat{v}(S_t, \mathbf{w}_t)\Big)^2 \mid S_t = s\Big] \\ &= \mathbb{E}_\pi\Big[\Big(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\Big)^2\Big],\end{aligned}$$

if training on-policy and

$$= \mathbb{E}_b\Big[\rho_t\Big(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\Big)^2\Big],$$

if training off-policy, where b represents the behavior policy and ρ is the importance sampling ratio. Note that the parameter $\mathbf{w}_t$ is time dependent this is because TD(0) is fully online and trains within the episodes.

Whenever we get a new observation we would like update our parameters iteratively. There are many methods to do so which utilizes second order derivatives and advanced techniques, but in practice *stochastic gradient descent* (SGD) is a popular and reliable choice. With $\overline{\mathbf{VE}}$ as objective function standard gradient descent performs the iterative update

$$\begin{aligned}\mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t - \alpha\frac{1}{2}\nabla_{\mathbf{w}}\overline{\mathbf{VE}} \\ &= \mathbf{w}_t + \alpha\mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(S, \mathbf{w})]\end{aligned}$$

and stochastic gradient descent samples the update

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)]\ \nabla_{\mathbf{w}}\hat{v}(s_t, \mathbf{w}_t),$$

where α is the *learning rate* or step size. The lower α the better convergence properties, but the slower the learning and vice versa. SGD updates like this makes it possible to learn fully online, i.e. we learn immediately from each action taken, in an efficient way. Though instead of updating the weights after each time step a common practice is to use *mini batch gradient descent*, which at every m steps in time samples a batch $\mathcal{B}$ from memory $\mathcal{M}$ to perform gradient descent upon. The memory

$$\mathcal{M} = \Big\{\{s_1, v_\pi(s_1)\}, \{s_2, v_\pi(s_2)\},\ ...,\ \{s_t, v_\pi(s_t)\}\Big\}$$

represents all data points an agent has experienced at a given time point. At time $t+1$ in the context of estimating a state-value function we get

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha\frac{1}{\|\mathcal{B}\|}\sum_{\{s, v_\pi(s)\}\in\mathcal{B}}[v_\pi(s) - \hat{v}(s, \mathbf{w}_t)]\ \nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w}_t), \qquad \mathcal{B} \subseteq \mathcal{M}.$$

Updating in mini batches saves a lot of computing power compared to training on the full updated data frequently. It is often also a preferred solution over learning fully online, because modern implementations can parallelize the gradient updates. An advantage of both SGD and mini batch gradient descent is that they are less prone to get stuck in local minimums. This is a consequence of the noise added to the direction in the sampling process.

Sometimes $\mathcal{B}$ is restricted to the most recent experiences. If samples are drawn from the past and replayed (trained upon), this is called *experience replay.*Through time our function approximators change, and thus replaying a data point with our new updated functions will also provide a new representation of the old state. Often gaining new experiences can be expensive, why learning multiple times from previous experiences can be efficient.

4.3.2 Policy Gradient Methods

Let us consider the performance measure $J(\pi_{\boldsymbol{\theta}}) = J(\boldsymbol{\theta})$ of a policy $\pi_{\boldsymbol{\theta}}$. In an episodic environment with the same starting state s_0 a natural choice is the state-value function of s_0 following $\pi_{\boldsymbol{\theta}}$:

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

To estimate $\boldsymbol{\theta}$, we use gradient descent methods again, and with $J(\boldsymbol{\theta})$ as the objective function, we update our parameters as follows.

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}$$

where $\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}$ is an estimate of the performance measure with expected value approximately equal to the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$. Notice that the minus sign in the update has become a plus, previously we wished to minimize a loss function, whereas we now desire to maximize a performance function. All RL methods that uses policy updates as in the equation above is called *policy gradient methods*.

When the function approximator for $\pi_{\boldsymbol{\theta}}$ is differentiable with respect to $\boldsymbol{\theta}$ and we know the gradient $\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)$ we can calculate the gradient analytically using likelihood ratios:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) &= \pi_{\boldsymbol{\theta}}(a \mid s)\frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)} \\ &= \pi_{\boldsymbol{\theta}}(a \mid s)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s).\end{aligned}$$

where $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)$ is the score function.

We now have a performance measure $J(\boldsymbol{\theta})$ and a parameterized policy function $\pi_{\boldsymbol{\theta}}(a \mid s)$ with a gradient we can calculate. But

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) &= \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s_0) \\ &= \nabla_{\boldsymbol{\theta}} \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s, a), \quad \text{(eq., 9)}\end{aligned}$$

which depends on the state distribution $\mu_{\pi_{\boldsymbol{\theta}}}(s)$. And we do not know the effect of the policy changes on $\mu_{\pi_{\boldsymbol{\theta}}}(s)$. This is where the *Policy Gradient Theorem* (R. Sutton et al. 1999) comes in handy. It utilizes likelihood ratios and creates a new formulation for $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)$, which does not include derivatives of $\mu\pi_{\boldsymbol{\theta}}(s)$.

Theorem 4.1. *(Policy Gradient Theorem). Assuming a finite state space $\mathcal{S}$. Then for any differentiable policy $\pi_{\boldsymbol{\theta}}(a \mid s)$, the policy gradient is*

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\, q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t)].$$

Proof. From eq. (9) we have that $v_\pi(s) = \mathbb{E}_\pi[q_\pi(s, A_t)] = \sum_{a \in \mathcal{A}} \pi(a \mid s) q_\pi(s, a)$ and thus

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) &= \nabla_{\boldsymbol{\theta}}\Big[\sum_a \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s, a)\Big], \\ &= \sum_a \Big[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi_{\boldsymbol{\theta}}(a \mid s) \nabla_{\boldsymbol{\theta}} q_{\pi_{\boldsymbol{\theta}}}(s, a)\Big]\end{aligned}$$

using the product rule. From eq. (7) we get

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) &= \sum_a \Big[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a) + \pi_{\boldsymbol{\theta}}(a \mid s) \nabla_{\boldsymbol{\theta}} \mathbb{E}\left[R_{t+1} + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1}) \mid S_t = s, A_t = a\right]\Big] \\ &= \sum_a \Big[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a) + \pi_{\boldsymbol{\theta}}(a \mid s) \nabla_{\boldsymbol{\theta}} \sum_{s',r} p(s', r \mid s, a)(r + \gamma v_{\pi_{\boldsymbol{\theta}}}(s'))\Big]\end{aligned}$$

because $p(s', r \mid s, a)r$ is independent of $\boldsymbol{\theta}$ and given the definition of $p(s' \mid s, a)$, eq.(4) we get

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) &= \sum_a \Big[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a) + \pi_{\boldsymbol{\theta}}(a \mid s) \nabla_{\boldsymbol{\theta}} \sum_{s'} p(s' \mid s, a) \gamma v_{\pi_{\boldsymbol{\theta}}}(s')\Big] \\ &= \sum_a \Big[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a) + \pi_{\boldsymbol{\theta}}(a \mid s) \sum_{s'} p(s' \mid s, a) \nabla_{\boldsymbol{\theta}} \gamma v_{\pi_{\boldsymbol{\theta}}}(s')\Big],\end{aligned}$$

from which we see a recursion similar to the one from the bellman equation, eq. (6). To ease notation we let

$$b(s) = \sum_a \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a),$$

and the n-step transition probability from state s to s' under policy $\pi_{\boldsymbol{\theta}}$ be denoted as

$$p_n(s' \mid s, \pi_{\boldsymbol{\theta}}) = \mathbb{P}_{\pi_{\boldsymbol{\theta}}}(S_{t+n} = s' \mid S_t = s)$$

such that

$$\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) = b(s) + \sum_{s'} p_1(s, s', \pi_{\boldsymbol{\theta}}) \gamma \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s')$$

and by inserting $\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s')$

$$\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) = b(s) + \sum_{s'} \Big[p_1(s, s', \pi_{\boldsymbol{\theta}}) \gamma \Big[b(s') + \sum_{s''} p_1(s', s'', \pi_{\boldsymbol{\theta}}) \gamma \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s'')\Big]\Big],$$

which we can write using the Chapman-Kolmogorov equation, eq. (1)

$$\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) = b(s) + \sum_{s'} \Big[p_1(s, s', \pi_{\boldsymbol{\theta}}) \gamma b(s') + \sum_{s''} p_2(s, s'', \pi_{\boldsymbol{\theta}}) \gamma^2 \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s'')\Big],$$

inserting again

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) = b(s) + \sum_{s'} \Big[&p_1(s, s', \pi_{\boldsymbol{\theta}}) \gamma b(s') + \sum_{s''} \Big[p_2(s, s'', \pi_{\boldsymbol{\theta}}) \gamma^2 b(s'') \\ &+ \sum_{s'''} p_3(s, s''', \pi_{\boldsymbol{\theta}}) \gamma^3 \nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s'')\Big]\Big].\end{aligned}$$

Note that this recursion continues and the longer we do so, the higher the exponent of γ. Thus eventually the term including $\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}$ will be eliminated because $0 \leq \gamma < 1$. This leads to

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} v_{\pi_{\boldsymbol{\theta}}}(s) &= \sum_{t=0}^{\infty} \sum_{s'} p_t(s' \mid s, \pi_{\boldsymbol{\theta}}) \gamma^t b(s') \\
&= \sum_{t=0}^{\infty} \sum_{s'} p_t(s' \mid s, \pi_{\boldsymbol{\theta}}) \gamma^t \sum_{a} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s') q_{\pi_{\boldsymbol{\theta}}}(s', a),
\end{aligned}$$

which applies for all s. Now recall that the performance measure is with respect to the first state s_0:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}}(s_0) \\
&= \sum_{s'} \sum_{t=0}^{\infty} p_t(s' \mid s_0, \pi_{\boldsymbol{\theta}}) \gamma^t \sum_{a} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s') q_{\pi_{\boldsymbol{\theta}}}(s', a).
\end{aligned}$$

Now recall that $\eta(s') = \sum_{t=0}^{\infty} \gamma^t p_t(s' \mid s_0, \pi_{\boldsymbol{\theta}})$, eq. (10) is the expected number of time steps in s, such that

$$\nabla J(\boldsymbol{\theta}) = \sum_{s'} \eta(s') \sum_{a} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s') q_{\pi_{\boldsymbol{\theta}}}(s', a).$$

Dividing and multiplying by the total number of time steps $\sum_s \eta(s)$:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \sum_{s} \eta(s) \sum_{s'} \frac{\eta(s')}{\sum_s \eta(s)} \sum_{a} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s') q_{\boldsymbol{\theta}}(s', a) \\
&\propto \sum_{s'} \mu(s') \sum_{a} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s') q_{\pi_{\boldsymbol{\theta}}}(s', a),
\end{aligned}$$

where $\propto$ means that it is proportional to. The constant of proportionality is $\sum_s \eta(s)$. Using likelihood ratios we finally get

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_{s'} \mu(s') \sum_{a} \pi_{\boldsymbol{\theta}}(a \mid s') \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s')}{\pi_{\boldsymbol{\theta}}(a \mid s')} q_{\pi_{\boldsymbol{\theta}}}(s', a) \\
&\propto \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t)]
\end{aligned}$$

□

The theorem says that we can take a step in the correct gradient direction of the performance measure, with respect to our policy parameters. We do this by adjusting our policy such that we do more of the actions that provide high rewards and less of the actions which provide low rewards, a quite intuitive result. The result leads us to our first policy gradient algorithm REINFORCE (Williams 1992). The REINFORCE algorithm

uses Monte Carlo methods to run through episodes using $\hat{v}_\pi(s_t) = G_t$ as an unbiased estimates of $q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t)$ followed by a policy update combining SGD with the policy gradient theorem:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) G_t.$$

The full algorithm is shown in algorithm 6.

Algorithm 6: REINFORCE

Input:
- Differentiable policy parameterization $\pi_{\boldsymbol{\theta}}(a \mid s)$.
- learning rate $\alpha \in [0, 1]$.
- Initial policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$.

Output: Policy parameter $\boldsymbol{\theta}$.

```
for each episode following π_θ do
    for t = 0, 1, ...T − 1 do
        θ ← θ + α∇_θ log π_θ(a_t | s_t) G_t
    end
end
```

4.3.3 Continuous Action Spaces

When we are in the case of continuous action spaces or in a case where we have a very large action space, the most effective way is not to use our policy to evaluate probabilities for each of the actions. Instead we want our policy to learn statistics regarding a distribution of our actions. Such a distribution could be the multivariate normal distribution, where we need to estimate the mean, $\mu(s, \boldsymbol{\theta}_\mu) \in \mathbb{R}^{d_a}$ and $\Sigma(s, \boldsymbol{\theta}_\Sigma) \in \mathbb{R}^{d_a \times d_a}$, such that $a \sim N\Big(\mu(s, \boldsymbol{\theta}_\mu), \Sigma(s, \boldsymbol{\theta}_\Sigma)\Big)$, i.e. the policy is given by

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{1}{\sqrt{(2\pi)^{d_a}|\Sigma(s, \boldsymbol{\theta}_\Sigma)|}} \exp\Big(-\frac{1}{2}(a - \mu(s, \boldsymbol{\theta}_\mu))^\top \Sigma(s, \boldsymbol{\theta}_\Sigma)^{-1}(a - \mu(s, \boldsymbol{\theta}_\mu)\Big),$$

where $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\Sigma]^\top$. It does seem very appealing to model the policy as a multivariate normal distribution, but in practice estimating the variance parameters makes the problem a lot more difficult for the agents, while not providing a lot of gain in performance. The reason why we include the variance term is to provide exploration. It is less important to find optimal exploration than optimal actions. Thus in practice we assume the actions to be uncorrelated and therefore let Σ be a diagonal matrix, with constant elements chosen by the modeller. To solve the problem we then need to calculate the score function of $\mu(s, \boldsymbol{\theta}_\mu)$, which is

$$\nabla_{\boldsymbol{\theta}_\mu} \log \pi_{\boldsymbol{\theta}}(a \mid s) = \left(\frac{\partial \mu(s, \boldsymbol{\theta}_\mu)}{\partial \boldsymbol{\theta}_\mu}\right)^\top \Sigma^{-1}\Big(a - \mu(s, \boldsymbol{\theta}_\mu)\Big)$$

where $\frac{\partial \mu(s, \boldsymbol{\theta}_\mu)}{\partial \boldsymbol{\theta}_\mu}$ is a Jacobian matrix.

4.4 Actor-Critic Methods

The REINFORCE method has a problem with high variance in its estimates, which is due to the Monte-Carlo sampling that makes the trajectories of rewards and log-probabilities vary a lot. In theory one could increase the sample size, but in practice this is not a computationally efficient solution. This variance makes the gradients unstable, but luckily *actor-critic methods* provides tools to ease the issue by utilizing function approximators for both policy and value functions. The "critic" estimates the value-function and the "actor" updates the policy in the direction we get from the critic. The idea is that we in actor-critic methods not only update the policy in regards to the observed reward, but also in regards to our expectation to the rewards. Hence the critic refers to the function approximator of the action-value function

$$\hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) \approx q_{\pi_\theta}(s, a),$$

where $\mathbf{w}$ is the parameters in the action-value function and $\boldsymbol{\theta}$ is the parameters used to set the policy $\pi_{\boldsymbol{\theta}}$. Actor-critic methods uses the approximate policy gradient

$$\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})} \approx \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w}) \right],$$

such that

$$\Delta \boldsymbol{\theta} = \alpha \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}).$$

The *compatible function approximation theorem* (Theorem 4.2) states that we can replace the true q-function with an approximation in the policy gradient and still ensure that the true gradient is preserved. An approximation of $q(s, a)$ is thus compatible with the policy $\pi_\theta(s, a)$ in the policy gradient if it meets the criteria given in Theorem 4.2.

Theorem 4.2. *(Compatible Function Approximation). A function approximator, $\hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w})$, is compatible with a policy, $\pi_{\boldsymbol{\theta}}$ if the value function approximator is compatible to the policy:*

$$\nabla_{\mathbf{w}} \hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)$$

and $\mathbf{w}$ minimizes the MSE

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[(q_{\pi_\theta}(S_t, A_t) - \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w}))^2 \right]$$

and thereby ensuring that the gradient is exact:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w}) \right]$$

Proof. If $\mathbf{w}$ is chosen to minimize the MSE then the gradient of the MSE w.r.t $\mathbf{w}$ must be zero:

$$\nabla_{\mathbf{w}} \text{MSE} = \mathbb{E} \left[2 \left(q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t) - \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w}) \right] = 0.$$

We have that

$$\nabla_{\mathbf{w}} \hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) = \nabla_{\boldsymbol{\theta}} \log(\pi_{\boldsymbol{\theta}}(a \mid s))$$

and by substituting it into the MSE gradient we get

$$\nabla_{\mathbf{w}}\text{MSE} = \mathbb{E}\left[2\left(q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t) - \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})\right)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\right] = 0$$

expanding and ignoring the factor 2

$$\begin{aligned} 0 &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) - \hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)] \\ &= \mathbb{E}\left[q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\right] - \mathbb{E}\left[\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\right]. \end{aligned}$$

Because $\nabla_{\mathbf{w}}\text{MSE} = 0$ the two expectations on the right hand side are equal. Thus we can substitute $\hat{q}_{\pi_{\boldsymbol{\theta}}}$ in for q and still follow the true gradient:

$$\mathbb{E}\left[q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\right] = \mathbb{E}\left[\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\right].$$

□

Due to the high variance of the action value function, we wish to subtract a baseline from it to reduce the variance. This needs to be done such that it does not change the direction of our gradient step. A good baseline is our state-value function, which then means we will compare how good a specific action is given the mean value of the corresponding state. By subtracting the state-value function from the action-value function we create what is called the *advantage* function

$$Adv_{\pi_{\boldsymbol{\theta}}}(s, a) = q_{\pi_{\boldsymbol{\theta}}}(s, a) - v_{\pi_{\boldsymbol{\theta}}}(s).$$

Note that it is important that the baseline function is independent of a since we then do not add any bias to the gradient. As we will see, this does not change the expectation. We let $\mathbf{v}$ denote the parameter vector for estimating $v_{\pi_{\boldsymbol{\theta}}}$, in order to distinguish between the parameters used for estimating $v_{\pi_{\boldsymbol{\theta}}}$ and $q_{\pi_{\boldsymbol{\theta}}}$ in following derivations. Substituting in the advantage function gives

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)(\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t, \mathbf{v}))], \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log(\pi_{\boldsymbol{\theta}}(A_t \mid S_t))\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})] - \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t, \mathbf{v})], \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})] - \sum_s \mu(s) \sum_a \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s)\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v}), \end{aligned}$$

using that both $\mu(s)$ and $\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v})$ is independent of a:

$$= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})] - \sum_s \mu_{\pi_{\boldsymbol{\theta}}}(s)\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v})\nabla_{\boldsymbol{\theta}} \sum_a \pi_{\boldsymbol{\theta}}(a \mid s),$$

where $\mu_{\pi_{\boldsymbol{\theta}}}(s)$ is the on-policy distribution following $\pi_{\boldsymbol{\theta}}$. Using that $\sum_a \pi_{\boldsymbol{\theta}}(a \mid s) = 1$ since it is a sum over all the probabilities, which must sum to 1 we get

$$\begin{aligned} &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log(\pi_{\boldsymbol{\theta}}(A_t \mid S_t))\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})] - \sum_s \mu(s)\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v})\nabla_{\boldsymbol{\theta}} \sum_a 1, \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log(\pi_{\boldsymbol{\theta}}(A_t \mid S_t))\hat{q}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t, \mathbf{w})] - \sum_s \mu(s)\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v}) \cdot 0, \end{aligned}$$

To do this we let the critic approximate both value functions and compute the advantage from these:

$$\begin{aligned}\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v}) &\approx v_{\pi_{\boldsymbol{\theta}}}(s), \\ \hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) &\approx q_{\pi_{\boldsymbol{\theta}}}(s, a), \\ \hat{Adv}_{\pi_\theta}(s, a) &= \hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v}).\end{aligned}$$

Thus to reduce variance we can replace $\hat{q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w})$ by the advantage function in our estimated gradient, providing

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \hat{Adv}_{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \right].$$

However we can do this in a smarter fashion by using TD(0). Recall the TD error:

$$\delta_{\pi_{\boldsymbol{\theta}}} = R_t + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1}) - v_{\pi_{\boldsymbol{\theta}}}(S_t),$$

which is an unbiased sample estimate of the advantage function:

$$\begin{aligned}\mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\delta_{\pi_{\boldsymbol{\theta}}} \mid S_t = s, A_t = a] &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[R_t + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1}) - v_{\pi_{\boldsymbol{\theta}}}(S_t) \mid S_t = s, A_t = a \right] \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[R_t + \gamma v_{\pi_{\boldsymbol{\theta}}}(S_{t+1}) \mid S_t = s, A_t = a \right] - v_{\pi_{\boldsymbol{\theta}}}(s) \\ &= q_{\pi_{\boldsymbol{\theta}}}(s, a) - v_{\pi_{\boldsymbol{\theta}}}(s) \\ &= Adv_{\pi_{\boldsymbol{\theta}}}(s, a),\end{aligned}$$

why we substitute it into our gradient replacing the advantage function

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t) \delta_{\pi_{\boldsymbol{\theta}}}].$$

In practice we use an approximation of the state-value function, and only need to estimate one set of parameters, $\mathbf{v}$, instead of estimating both $\mathbf{v}$ and $\mathbf{w}$:

$$\delta_v = r + \gamma \hat{v}_{\pi_{\boldsymbol{\theta}}}(S_{t+1}, \mathbf{v}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(S_t, \mathbf{v}).$$

We call actor critic policy gradient methods, which uses the advantage function for advantage-actor-critic (A2C) algorithms. The A2C algorithm using TD(0) as an estimate of the advantage function is show in Algorithm 7, and a visualization of its interactions between the actor, critic and the environment is shown in figure 5.

Algorithm 7: Advantage-Actor-Critic (A2C) with TD(0) step (online)

Input:

- Differentiable parameterized policy $\pi_{\boldsymbol{\theta}}(a \mid s, \boldsymbol{\theta})$.
- Differentiable parameterized state-value function $\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v})$.
- learning rate $\alpha_{\boldsymbol{\theta}}, \alpha_{\mathbf{v}} \in]0, 1]$.
- Initial policy parameter and state-value parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}, \mathbf{v} \in \mathbb{R}^{d}$.

Output: Policy π_θ

for *each episode* **do**
  $I \leftarrow \mathbf{1}$
  for $t = 0, 1, ...T-1$ **do**
    Take action $a \sim \pi_{\boldsymbol{\theta}}(A_t \mid s)$, observe s', r
    $\delta \leftarrow r + \gamma \hat{v}_{\pi_{\boldsymbol{\theta}}}(s', \mathbf{v}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v})$
    $\mathbf{v} \leftarrow \mathbf{v} + \alpha_{\mathbf{v}} \delta \nabla \hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \mathbf{v})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} I \delta \nabla \log \pi_{\boldsymbol{\theta}}(a \mid s)$
    $I \leftarrow I\gamma$
    $s \leftarrow s'$
  end
end

5.

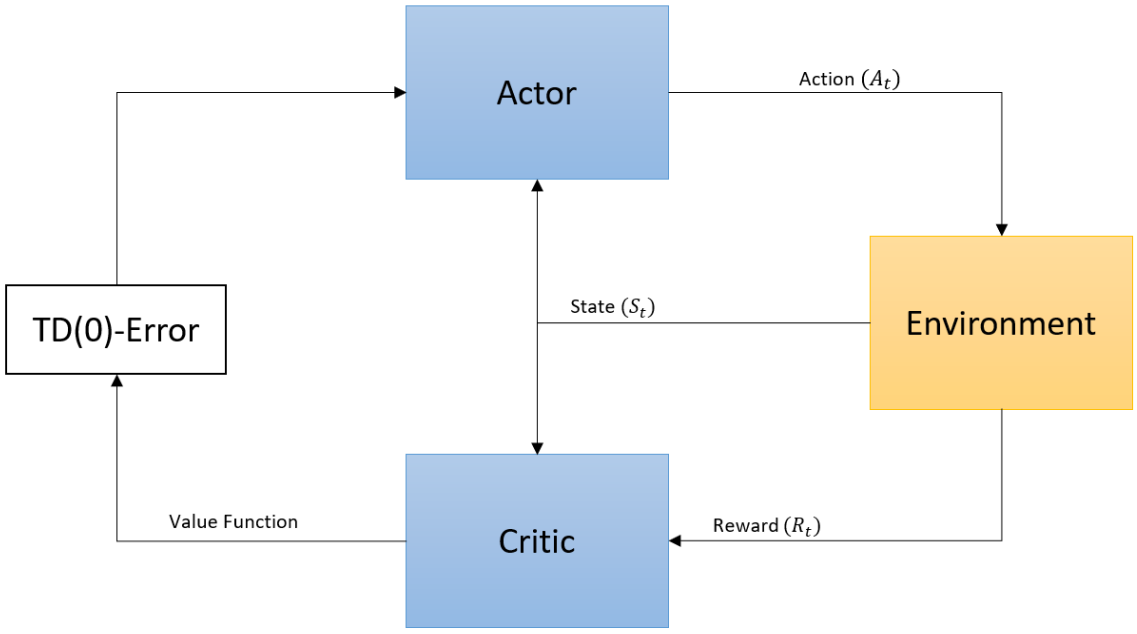


Figure 5: *A representation of the A2C algorithm: Advantage - Actor - Critic*

Part II
Portfolio Optimization with Reinforcement Learning

In this part we build upon the study made by Petter N. Kolm and Gordon Ritter in 2019 (Kolm and Ritter 2019), where they create an RL agent that trades a mean-reverting process around a predefined equilibrium price. Their environment enforces an alpha strategy, by selling above the equilibrium price and buying below. We will contribute by implementing a RL agent that trades an index, which is simulated by a AR(1)+GARCH(1,1) process, which is known to reflect real market dynamics well. We will estimate the parameters from daily S&P500 prices observed from 2010 to 2020. This way we incorporate real data, and produce a simulation which reflect the dynamics of the real stock market better. Furthermore we will back test the agents performance on the period from 2020-01-01 to 2022-04-01.

5 Background

We will start by considering the classical *utility maximization problem*: How should one individual allocate her money in order to maximize her utility? Consider the expected future utility at time T:

$$\mathbb{E}\left[u(w_T)\right] = \mathbb{E}\left[u\left(w_0 + \sum_{t=1}^{T}[w_t - w_{t-1}]\right)\right]$$

where u is the utility function and w_t is the individuals wealth at time t. Because the future changes in wealth are unknown, we often let the maximization problem take risk into account. We call individuals that care about risk for risk-averse. There are multiple ways to model risk, but we will consider the general case, in which risk is measured in variance of change in wealth.

The changes in wealth are assumed independent, such that $w_t - w_{t-1} \perp\!\!\!\perp w_k - w_{k-1},\ k \neq t$. Thus the variance in wealth at time T can be written as:

$$\mathrm{Var}[w_T] = \sum_{t=1}^{T} \mathrm{Var}[w_t - w_{t-1}].$$

Now consider the risk-averse mean-variance utility function (Chamberlain 1983)

$$\mathbb{E}\left[u(w_T)\right] = \mathbb{E}\left[w_T\right] - \frac{\kappa}{2}\mathrm{Var}[w_T],$$

κ controls the investors sensitivity to risk, and is often called the risk-aversion parameter. In this case we define risk as the variance in the change in wealth. If κ is zero the investor

is risk-neutral, i.e. ignores risk and only cares about maximum wealth. As κ becomes large the investor will tend toward was is called the minimum variance strategy. The importance of risk becomes large and the investor will only care to minimize risk. In RL the individual (an agent), will seek to maximize its utility (cumulative future reward), by adjusting its policy. This leads to the maximization problem of interest:

$$\max_{\pi} \mathbb{E}\left[u(w_T)\right] = \max_{\pi} \left\{ \mathbb{E}_{\pi}[w_T] - \frac{\kappa}{2} \mathrm{Var}_{\pi}[w_T] \right\}. \tag{13}$$

As presented in Part I we seek to maximize the expectation of the cumulative future reward:

$$\max_{\pi} \mathbb{E}_{\pi}[G_t],$$
$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k,$$

where R_t represents the reward the agent received at time t and γ the discount factor. In the RL setting the change in utility is expressed in form of rewards, i.e. we need to to define a reward function resembling eq. (13). To this we let the reward at time t be expressed as

$$R_t = (w_t - w_{t-1}) - \frac{\kappa}{2} \left([w_t - w_{t-1}] - \hat{\mu} \right)^2, \qquad \hat{\mu} = \mathbb{E}\left[w_t - w_{t-1} \right]$$

Taking the sample average provides

$$\begin{aligned} \frac{1}{T} \sum_{t=1}^{T} R_t &= \frac{1}{T} \sum_{t=1}^{T} \left((w_t - w_{t-1}) - \frac{\kappa}{2} \left([w_t - w_{t-1}] - \hat{\mu} \right)^2 \right) \\ &= \underbrace{\frac{1}{T} \sum_{t=1}^{T} ((w_t - w_{t-1})}_{\approx \mathbb{E}[w_T]} - \frac{\kappa}{2} \underbrace{\frac{1}{T} \sum_{t=1}^{T} \left([w_t - w_{t-1}] - \hat{\mu} \right)^2}_{\approx \mathrm{Var}[w_T]} \end{aligned}$$

which by law of large numbers converge as T becomes large. Inserting this into the cumulative reward function, gives us an expression of the desired form

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} \left((w_t - w_{t-1}) - \frac{\kappa}{2} \left([w_t - w_{t-1}] - \hat{\mu} \right)^2 \right),$$

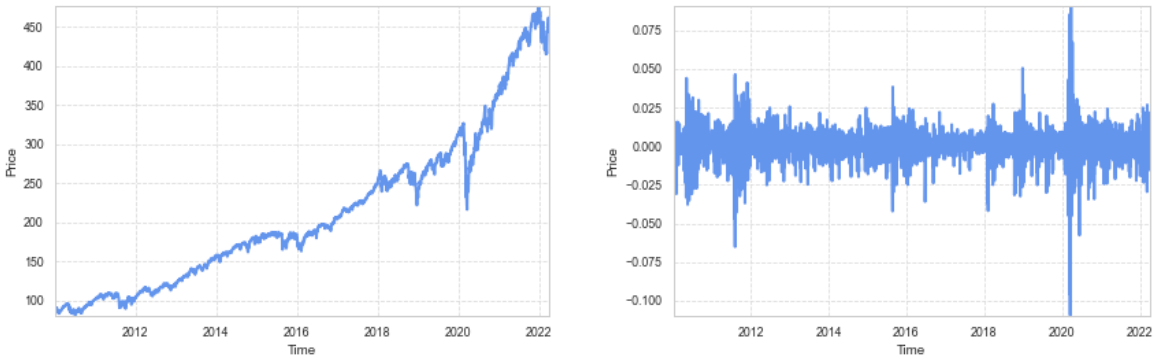
except for the constant w_0 which is irrelevant for the optimization problem and the discount factor γ, which represents the same in finance and RL: The cost of receiving utility (reward) in the future as opposed to the present. Because we do not have an estimate of $\hat{\mu}$, at initialization, a natural solution is to instantiate the biased estimate $\hat{\mu} = 0$, during the first learning period. When we have enough observations such that we are somewhat confident in our approximation of the value function, we let $\hat{\mu}$ be the sample average.

6 Modelling the Stock-Market

In this experiment, we are studying the applications of RL to perform asset trading in an index. For this purpose, we assume that the agent's actions won't affect the future development of the stock index price. And in order not to let the agent train upon the same observed returns repeatedly throughout the learning process. We wish to simulate new data with similar characteristics as the original stock index. The better these simulations reflect the true dynamics of the stock index, the better the results will be from our trained model.

6.1 Data

The data we will use in this study is *The Standard and Poor's 500* (S&P 500) stock market index, which tracks the performance of 500 of the largest companies on United States stock exchanges. The period is from the beginning of 2010 until and including 2022-04-01. The prices and returns is shown in figure 6 (a) and (b). We will use the data from 2010 till 2019-12-31 to simulate and train on, and the data from 2020-01-01 till 2022-04-01 for testing.



(a) *S&P 500 Prices* (b) *S&P 500 Returns*

Figure 6: *Returns and prices of the S&P 500 stock market index from 2010-01-01 until and including 2022-04-01.*

6.2 Stylized Facts

When studying asset prices and returns it is an empirical fact that they possess certain statistical properties (Cont 2001). These properties are known as the *stylized facts* of the financial markets. In our simulation we will attempt to reflect 3 key stylized facts: *absence of autocorrelation*, *heavy tails* and *volatility clustering*.

To avoid confusion between returns of assets and rewards of agents we let the return at time t be denoted as RE_t and defined as

$$\mathrm{RE}_t = \frac{\mathrm{P}_t}{\mathrm{P}_{t-1}} - 1.$$

where P_t is the price of the traded asset at time t.

6.2.1 Absence of Autocorrelation

The linear autocorrelation of returns in financial markets are insignificant, when time scales are larger than 20 minutes, which is obviously the case when modelling daily returns. This fact is known as the absence of autocorrelation. Autocorrelation refers to correlation between a time series and a lagged version of itself. To measure the autocorrelation, for a given lag we use the sample autocorrelation function (acf)

$$\rho(\tau) = \frac{\sum_{t=1}^{n-\tau}(\mathrm{RE}_t - \bar{\mathrm{RE}})(\mathrm{RE}_{t+\tau} - \bar{\mathrm{RE}})}{\sum_{t=1}^{n}(\mathrm{RE}_t - \bar{\mathrm{RE}})^2}$$

where $\bar{\mathrm{RE}}$ is the sample mean of RE. We expect that $\rho(\tau) \approx 0$. If this is not the case it implies that past returns reflect the future performance, which could be utilized to create statistical arbitrage strategies. The autocorrelation function of the S&P 500 index from 2010 until 2020 is shown in figure 7, and illustrates that it is close to zero for all lags. Though a bit low for lag 5 and 25 at 95% confidence interval.

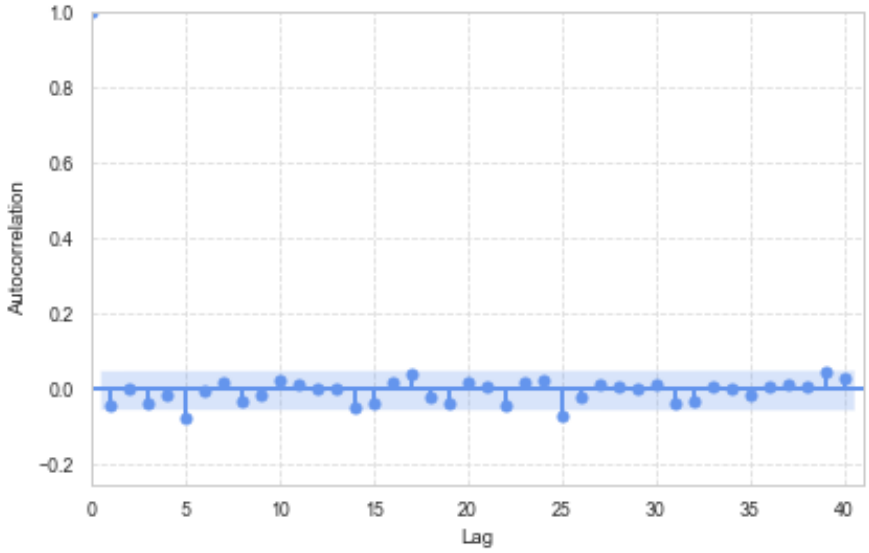


Figure 7: *Autocorrelation of the S&P 500 returns from 2010 until 2020.*

6.2.2 Heavy Tails

The second stylized fact is that the pdf of returns in financial time series have heavy tails, compared to the normal distribution. A natural way to measure "heaviness" of the tails of a distribution is through the fourth standardized moment, also known as kurtosis. The sample kurtosis is

$$\mathrm{Kurt} = \frac{1}{n-1}\sum_{t=1}^{n}\frac{(\mathrm{RE}_t - \bar{\mathrm{RE}})^4}{s^4}$$

where s the sample standard deviation of the returns. The kurtosis gets high in distributions with a lot of centered observations, as well as a lot of observations in the tails (heavy tails). The larger kurtosis the heavier the tails. The normal distribution has a kurtosis of 3, and thus a kurtosis larger than 3 reflects heavy tails. Figure 8 illustrates a distribution comparison between the selected S&P 500 returns and the normal distribution. It clearly shows that with a kurtosis of 7.4 the S&P 500 returns has heavier tails. The figure also shows that the return distribution has a skewness (third standardized moment) of -0.4. It is also a common statistic that financial data has negative skewness, as opposed to the normal distribution which has a skewness of 0. Though the general divergence from the normal distribution is not as clear as with kurtosis.

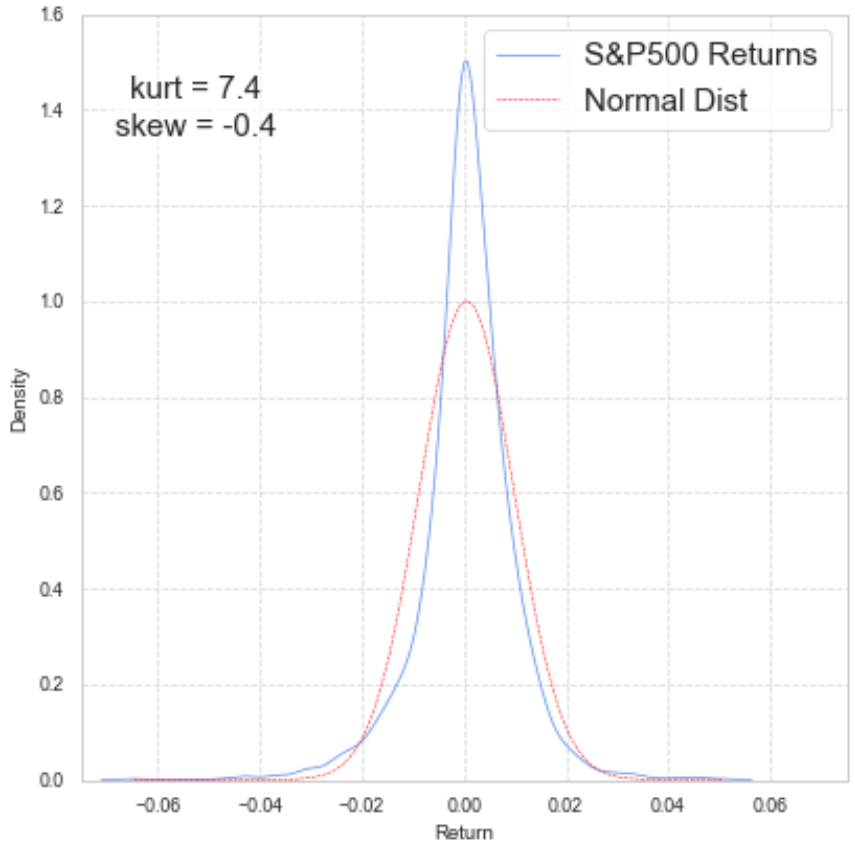


Figure 8: *Distribution comparison of S&P 500 returns and the normal distribution. It shows that the kurtosis and skewness of the returns are respectively 7.4 and -0.4, compared the 3 and 0 of the normal distribution.*

6.2.3 Volatility Clustering

The last stylized fact we will touch upon is volatility clustering, which states that volatility tends to cluster. This states that there is in fact autocorrelation in financial markets, not in the actual returns themselves, but in the size of the returns. The volatility clustering of S&P 500 returns is clearly shown in figure 6 (b). A natural method to measure volatility

clustering is through the autocorrelation of squared returns. This is this is shown in figure 9.

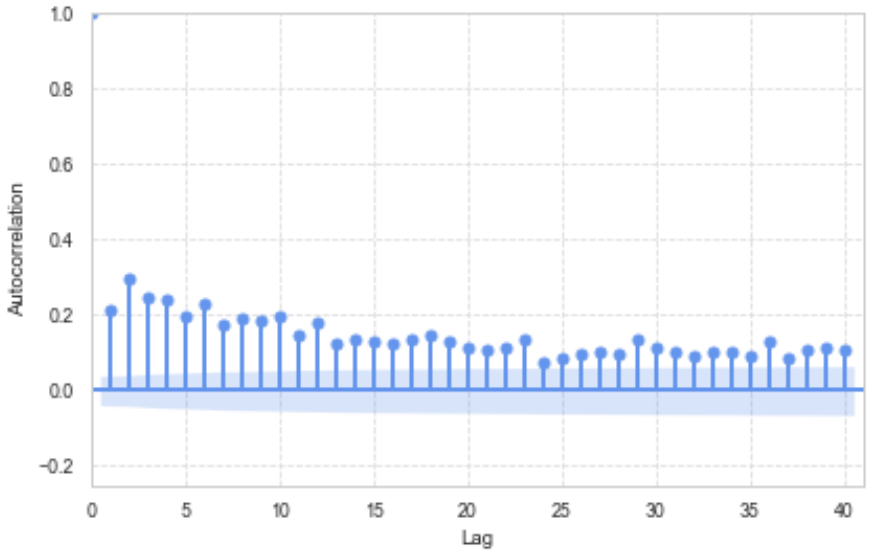


Figure 9: *Autocorrelation of the S&P 500 squared returns from 2010 until 2020. Illustrating the tendency to volatility clustering.*

6.3 AR + GARCH

To simulate the S&P 500 we use a combination of an Auto Regressive (AR) process and a Generalized Autoregressive Conditional Heteroskedasticity (GARCH) process. These processes are known to reflect the stylized facts well.

An AR(p) is a process which is modelled by a weighted average of the last p observations. The series $Y_1, ..., Y_n$ is an AR process if the following holds:

$$Y_t - \mu = \phi_1(Y_{t-1} - \mu) + \cdots + \phi_p(Y_{t-p} - \mu) + \epsilon_t,$$

where μ is the mean of Y, ϕ is the parameter vector of weights and ϵ is weak white noise; $\epsilon_t \sim \text{WN}(0, \sigma_\epsilon^2)$. For an AR(1) process we have that if $|\phi| < 1$ then Y is a stationary process, if $|\phi| = 1$ then it is a random walk and if $|\phi| > 1$ then it has explosive behaviour. Generally we say that stock returns are stationary while stock prices are non-stationary. For an AR(1) process with $|\phi| < 1$ then the following properties holds:

$$\begin{aligned}
\mathbb{E}[Y_t] &= \mu, \quad \forall t, \\
\text{Var}[Y_t] &= \frac{\sigma_\epsilon^2}{1-\phi^2}, \quad \forall t, \\
\text{Cov}[Y_t, Y_{t+h}] &= \phi^{|h|}\frac{\sigma_\epsilon^2}{1-\phi^2}, \quad \forall t, \forall h, \\
\text{Corr}[Y_t, Y_{t+h}] &= \phi^{|h|}, \quad \forall t, \forall h
\end{aligned}$$

To estimate the values for μ, ϕ and σ_ϵ one can use Maximum Likelihood with conditional least squares, where we know that in the AR(1) $Y_1, ..., Y_n$ is Markov

$$\mathbb{P}(Y_t|Y_{t-1}, ..., Y_1) = \mathbb{P}(Y_t|Y_{t-1}).$$

The joint density for $Y_1, ..., Y_n$ is

$$f_{Y_1,...,Y_n}(y_1, ..., y_n; \boldsymbol{\theta}) = f_{Y_1}(y_1; \boldsymbol{\theta}) \prod_{t=2}^{n} f_{Y_1|Y_{t-1}}(y_t|y_{t-1}; \boldsymbol{\theta}),$$

where $\boldsymbol{\theta}$ is our parameters $(\mu, \phi, \sigma_\epsilon)$. Assuming that the errors are following a Gaussian white noise, we have that $Y_1 \sim N\left(\mu, \frac{\sigma_\epsilon^2}{1-\phi^2}\right)$ and the only random component of Y_t is the noise-term ϵ_t when Y_{t-1} is given. We think of Y_1 as being deterministic and write the conditional likelihood as

$$f_{Y_n,...,Y_2|Y_1}(y_n, ..., y_2|y_1; \boldsymbol{\theta}) = \prod_{t=2}^{n} f_{Y_t|Y_{t-1}}(y_t|y_{t-1}; \boldsymbol{\theta})$$

we then maximize the following log-likelihood w.r.t to $\mu, \phi, \sigma_\epsilon$

$$\ell(\boldsymbol{\theta}) = \log\left[\left(\frac{1}{\sqrt{2\pi}\sigma_\epsilon}\right)^{n-1}\right] \sum_{t=2}^{n} -\frac{(Y_t - (\mu + \phi(Y_{t-1} - \mu)))^2}{2\sigma_\epsilon^2}.$$

One issue arises when modelling the financial markets, since they are known to experience volatility clustering. Because this we can not only use an AR process since it has constant conditional variance. However combining this AR process with a GARCH process allows us to account for these clusters of volatility. We make an AR + GARCH process by replacing the noise term in the AR process with a GARCH process. The GARCH(p,q) model (Bollerslev 1986) :

$$\begin{aligned}
a_t &= \sigma_t \epsilon_t, \\
\sigma_t &= \sqrt{\omega + \sum_{i=1}^{p} \alpha_i a_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2}, \\
p &\geq 0, q > 0, \\
\alpha_0 &> 0, \alpha_i \geq 0, i = 1, ..., q, \\
\beta_j &\geq 0 \; j = 1, ..., p.
\end{aligned}$$

where ω represents a constant volatility, α measures how much a volatility burst today continues through into the next period's volatility, and $\alpha + \beta$ measures the rate at which this effect dies over time. When $p = 0$ we have an ARCH(q) process. To get an idea about how this process works we take a look at the ARCH(1) process:

$$a_t = \epsilon_t \sqrt{\omega + \alpha a_{t-1}^2}, \quad \epsilon_t \sim N(0, 1).$$

Taking a look at the variance of a_t shows that this process allows these volatility clusters

$$\text{Var}(a_t|a_{t-1},...) = \mathbb{E}\left[\epsilon_t^2(\omega + \alpha a_{t-1}^2)|a_{t-1},...\right] - \mathbb{E}\left[\epsilon_t\sqrt{\omega + \alpha a_{t-1}^2}|a_{t-1},...\right]^2,$$

since ϵ_t and a_{t-1} is independent we can split the last expectation up

$$\text{Var}(a_t|a_{t-1},...) = \mathbb{E}\left[\epsilon_t^2(\omega + \alpha a_{t-1}^2)|a_{t-1},...\right] - \left(\mathbb{E}\left[\epsilon_t|a_{t-1},...\right]\mathbb{E}\left[\sqrt{\omega + \alpha a_{t-1}^2}|a_{t-1},...\right]\right)^2,$$

having $\mathbb{E}\left[\epsilon_t\right] = 0$ removes the last part

$$\text{Var}(a_t|a_{t-1},...) = \mathbb{E}\left[\epsilon_t^2(\omega + \alpha a_{t-1}^2)|a_{t-1},...\right],$$

using that ω and α are constants and a_{t-1} is known at time t we can take them out from the expectation

$$\text{Var}(a_t|a_{t-1},...) = (\omega + \alpha a_{t-1}^2)\mathbb{E}\left[\epsilon_t^2|a_{t-1},...\right],$$

ϵ_t independent on a_{t-1} and $\mathbb{E}\left[\epsilon_t^2\right] = \text{Var}[\epsilon_t] - \mathbb{E}\left[\epsilon_t\right]^2 = \text{Var}[\epsilon_t] = 1$ we get the following

$$\text{Var}(a_t|a_{t-1},...) = \omega + \alpha a_{t-1}^2.$$

We see that if a_{t-1} is large then $\text{Var}[a_t|a_{t-1},...]$ will also be proportionally larger than usual and vice versa. This can continue for some time, but having $\alpha < 1$ ensures finite variance. The unconditional variance of a_t is $\gamma_a(0)$, can be seen to be positive only for $\alpha < 1$:

$$\begin{aligned}\gamma_a(0) &= \mathbb{E}\left[a_t^2\right] \\ &= \mathbb{E}\left[\omega + \alpha a_{t-1}^2\right]\underbrace{\mathbb{E}\left[\epsilon_t^2\right]}_{=1} \\ &= \omega + \alpha\mathbb{E}\left[a_{t-1}^2\right]\end{aligned}$$

because the expectation and variance of a_t and a_{t-1} is not conditioned on the previous processes, they can be set equal providing:

$$\mathbb{E}\left[a_{t-1}^2\right] = \mathbb{E}\left[a_t^2\right] = \gamma_a(0)$$

and thus

$$\begin{aligned}\gamma_a(0) &= \omega + \alpha\gamma_a(0) \\ &= \frac{\omega}{1-\alpha}\end{aligned}$$

we therefore need to have $\alpha < 1$ to ensure a positive unconditional variance. Apart from the ARCH part of the process we have

$$\sum_{j=1}^{q}\beta_j\sigma_{t-j}^2$$

which feeds the past q values of σ into the process and allows more persistent volatility. The unconditional variance of the full GARCH process is (in the GARCH(1,1) case):

$$\gamma_a(0) = \frac{\omega}{1-\alpha-\beta},$$

thus we now need $\alpha + \beta < 1$.
The full AR+GARCH model is then

$$\begin{aligned} Y_t &= c + \sum_{i=1}^{p} [\phi_{t-i} Y_{t-i}] + a_t \\ c &= \mu \left(1 - \sum_{i=1}^{p} \phi_i\right) \\ a_t &= \sigma_t \epsilon_t, \\ \sigma_t &= \sqrt{\omega + \sum_{i=1}^{p} \alpha_i a_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2} \end{aligned}$$

So far we have only considered the case where the noise, ϵ_t, is Gaussian, but since stock returns have heavy tails this is often not the best distribution to use. Thus we instead model the process such that ϵ_t is an i.i.d white noise that follows a heavy tailed distribution. To model a skew t-distribution we use a method introduced by Bruce E. Hansen (Hansen 1994). He uses a t-distribution that is normalized to unit variance:

$$f_\eta(z) = \left[\frac{\Gamma\left(\frac{\eta+1}{2}\right)}{(\pi(\eta-2))^{\frac{1}{2}}\Gamma\left(\frac{\eta}{2}\right)}\right] \cdot \left[1 + \frac{z^2}{(\eta-2)}\right]^{-(\eta+1)/2},$$

where $2 < \eta < \infty$ controls heaviness of the tails. To further allow for skewness (which is often observed in financial markets), Hansen suggests the following method:

$$f_\eta(z,\lambda) = \begin{cases} bc\left[1 + \frac{1}{\eta-2}\left(\frac{bz+a}{1-\lambda}\right)^2\right]^{-(\eta+1)/2} & z < -a/b \\ bc\left[1 + \frac{1}{\eta-2}\left(\frac{bz+a}{1+\lambda}\right)^2\right]^{-(\eta+1)/2} & z > -a/b \end{cases},$$

where

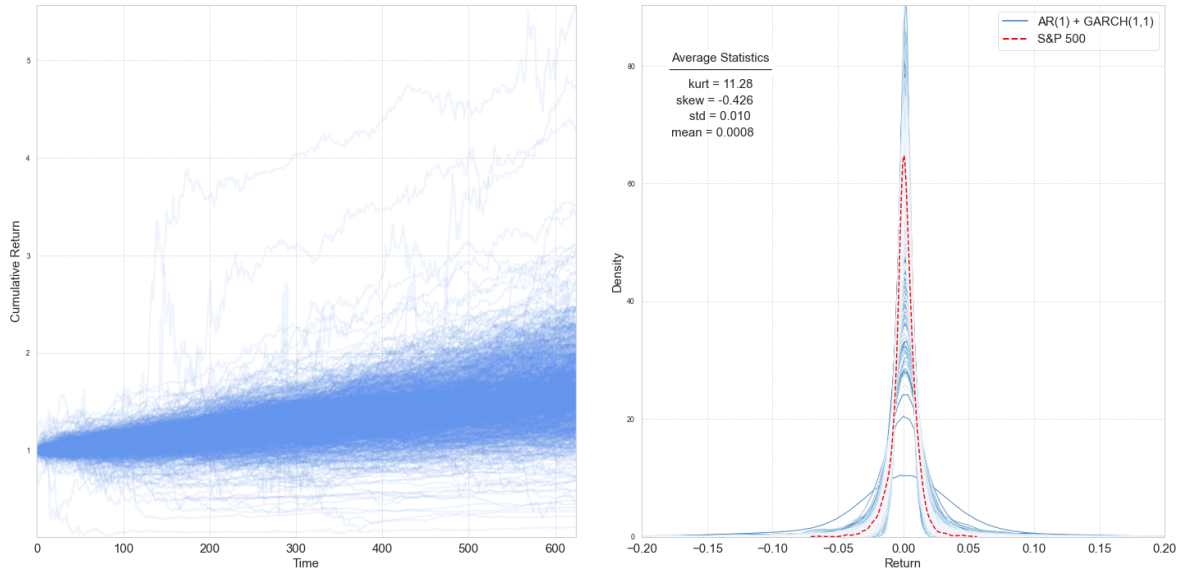
$$\begin{aligned} a &= 4\lambda c\left(\frac{\eta-1}{\eta-1}\right), \\ b^2 &= 1 + 3\lambda^2 - \alpha^2, \\ c &= \frac{\Gamma\left(\frac{\eta+1}{2}\right)}{(\pi(\eta-2))^{\frac{1}{2}}\Gamma\left(\frac{\eta}{2}\right)}, \end{aligned}$$

and $2 < \eta < \infty$ and $-1 < \lambda < 1$ is the skew-parameter. If $\lambda = 0$, we have the normal t distribution, and when $\lambda < 0$ it is left-skewed and for $\lambda > 0$ it is right-skewed.

6.4 Simulating stock prices with AR+GARCH

In our study we have decided to use an AR(1)+GARCH(1,1) model, which is considered the most widely model used to model financial time series (Ruppert and S. Matteson 2015). We simulate pseudo S&P 500 paths by through a AR(1) + GARCH(1,1) model with parameters estimated on the test data set. The noise ϵ_t is sampled using a skewed t-distribution. We train the agent on these simulated paths where each represents one episode. By letting the agent train on the different paths, we expose it to a lot of different scenarios compared to if we just trained it on the actual observed data. This will make the agent less prone to overfitting and therefore better at generalizing out of sample.

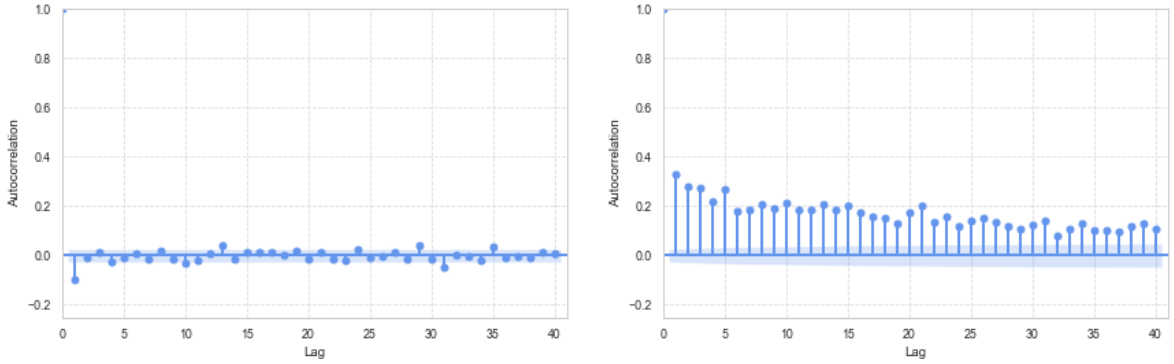
Figure 10 (a) shows 1000 examples of simulated paths of stock prices, from the AR(1)+GARCH(1,1) model. And Figure 10 (b) shows densities of the corresponding returns, as well as average statistics of each path of returns. We observe, a sample average kurtosis of 11, a skewness of -0.4, which roughly matches the real data. The kurtosis is a bit higher. Figure 11 shows the autocorrelation and squared autocorrelation of returns as an average over the 1000 paths. In Figure 11 (a), we see that most of the lags show no autocorrelation of returns, however, there is autocorrelation in the first lag, which is due to the negative AR(1) parameter. And in figure 11 (b), we see a slow decay in autocorrelation as expected.



(a) *Simulated S&P 500 prices*

(b) *Densities of simulated S&P 500 returns*

Figure 10: *100 AR(1) + GARCH(1,1) simulations of S&P-500 using data from 2010-01-01 to 2020-01-0 . figure (a) shows paths of stock prices and figure (b) return densities compared to the true S&P 500 density from the period, including key average statistics over the simulations.*



(a) *Acf of returns* (b) *Acf of squared returns*

Figure 11: *Autocorrelation function of returns (left) and squared returns (right), calculated as an average over 1000 simulated paths. We observe autocorrelation on the first lag of returns, which is due to the negative AR(1) parameter, while the rest are approximately 0. And we see a slow decay in autocorrelation of squared returns, implying volatility clustering, as expected.*

Table 1: *Parameter estimates for AR(1)+GARCH(1,1) and skewed t-distribution. The parameters are estimated on returns * 100, for better convergence properties in optimization.*

Parameter	S&P-500 Estimate
μ	0.083536
ϕ_1	-0.065907
ω	0.023387
α_1	0.169049
β_1	0.815012
η	5.403227
λ	-0.109379

7 REINFORCE Agent

7.1 The environment

The environments state space is deviating a bit from that of Peter and Kolm. They use the price as input, which makes perfectly sense since they want to model a mean-reverting process, however, this is not the case in this study. Due to the random nature of our simulations there are a lot of different paths and thus a lot of different price ranges for each path which complicates our value-function approximation. To get the input in a more stable format we use returns instead of prices.
The state space consists of the last k returns and the currency position size at time t:

$$\begin{aligned} S_t &= \{\mathrm{RE}_t, ..., \mathrm{RE}_{t-k+1}, h_t\} \\ \mathrm{RE}_t &= \text{Simulated return from } t-1 \text{ to } t, \in \mathbb{R} \\ h_t &= \text{Agents invested position size at time } t, \in \mathbb{Z} \end{aligned}$$

where k is a constant deciding how many time-steps before time t we want the agent to have information about. To model the state space we have decided to consider it continuous due to the continuous nature of the inputs we have in the state. Even though it in theory is possible to discretize the inputs we would not be able to do this in a computationally feasible way, since a matrix containing those states would have a way too large dimension.

When running the environment we start off by simulating a path using the AR(1)+GARCH(1,1) model, which consists of NP observations. The first k time steps in the can be seen as an initial burn-period. Thus when the agent observes the market (simulation) for the first time it will be at time k. We haven chosen an episode length of 525 trading days which is roughly equal to two and a half years of daily returns, thus $NP = 525 + k$ where k is decided when setting the agent up.

7.2 The Agent

We let the agent be modelled using REINFORCE (algorithm 6), and represent an investor controlling a portfolio of one asset, the simulated stock index. The agent then at each time point, has the possibility to increase, decrease or do nothing about its position. Up to a max position size of $h^{\max}$. We let $h_t \in \mathbb{Z}$ denote the agents position in the traded index at time t. $h_t \in \mathbb{Z}$ thus represents the exposure to the index and will be negative if the agent has a short position. We then have the portfolio value given by

$$v^{pf} = \mathrm{nav}_t + \mathrm{cash}_t,$$

where $\mathrm{nav}_t = h_t \mathrm{P}_t$ is the net asset value at time t, and cash_t is the corresponding cash position. Furthermore let

$$\delta h_t = |h_t - h_{t-1}|$$

denote the shares traded from $t-1$ to t, or absolute change in holdings. We assume that the buy/sell-period, where the trades are executed, always happen just before a new

time point. This way the change in position from $t-1$ to t will not be reflected in the corresponding period return. Because in reality markets have spreads and transaction costs, we include TC_t, which represents the difference between the market price, and the prices that the agent would be able to realize a trade for plus transaction cost at time t.

$$\mathrm{TC}_t = \delta h_t(\tilde{\mathrm{P}}_t - \mathrm{P}_t), \quad \tilde{\mathrm{P}}_t = \begin{cases} \mathrm{P}_t(1-\mathrm{cost}) & \text{if selling} \\ \mathrm{P}_t(1+\mathrm{cost}) & \text{if buying} \end{cases}$$

where $\tilde{\mathrm{P}}_t$ represents the effective buy (sell) price of the agent determined by the cost parameter. This leads to the change in portfolio value (wealth)

$$\delta v_t^{pf} = h_{t-1}\mathrm{P}_{t-1}\mathrm{RE}_t - \mathrm{TC}_t$$

Recall that in REINFORCE we use Monte Carlo and estimate $\hat{q}_{\pi_{\boldsymbol{\theta}}}(s,a)$ for our policy gradient step with G_t, which in our utility context, as shown in section 5 is given by

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} \left((\delta v_t^{pf}) - \frac{\kappa}{2} \left([\delta v_t^{pf}] - \hat{\mu} \right)^2 \right),$$

where $\hat{\mu}$ is the sample average change in portfolio value, γ the discount factor and κ the risk-aversion parameter.

We let the agents action space be defined by the possibilities of buying or selling 0, 1 or 2 assets in the market, which can be done at each time step. We have made the action space in this way since we want the agent to be able to get in and out of the market at different speeds according to what it observes in the state space. For this problem we have implemented the REINFORCE algorithm with a dense feed-forward neural network as function approximator for the policy. To decide the number of layers and hidden units, we have tested different combinations of the numbers of layers and number of hidden units. We have generally tested combinations between 2 and 6 hidden layers, and $\{2^3, 2^4, ..., 2^{10}$ number of hidden units. When deciding the neural network architecture in a RL setup, there is not a method that is as straight forward as when we do it in a regression/classification problem. Since the actions that the agent takes depend on the predictions of the network, we need to make sure that it gets to try out a lot of different actions at the different states. To do this we kept track of the following during training:

- The actions that the agent took during each episode
- Distribution of actions taken during each episode
- Position size during each episode
- The agents reward and loss during each episode

When evaluating these observations we wanted the agent to start off with a lot of exploration and slowly converging to a strategy.

Regarding the selection of activation functions we generally prefer nonlinear ones. If we just pick a linear activation function, the network will also have a linear relationship

between layers, and thus will not be able to detect nonlinear relations. To make sure the network can find nonlinear relations between neurons we will use the ReLU activation function. It has become a popular choice, both because it is computationally efficient and does not suffer (to the same degree as tanh and sigmoid) from vanishing gradients (Basodi et al. 2020). The output-layer is using the softmax activation function since the output is probabilities.

The final neural network we have decided to use has 4 hidden layers with number of hidden units: 512, 256, 128 and, 64 respectively.

To mitigate overfitting each layer is further applied with a dropout probability of 0.1, plus a L2 regularization on the objective function, with a weight of $1e-04$. The neural networks layers is sketched in Figure 12.

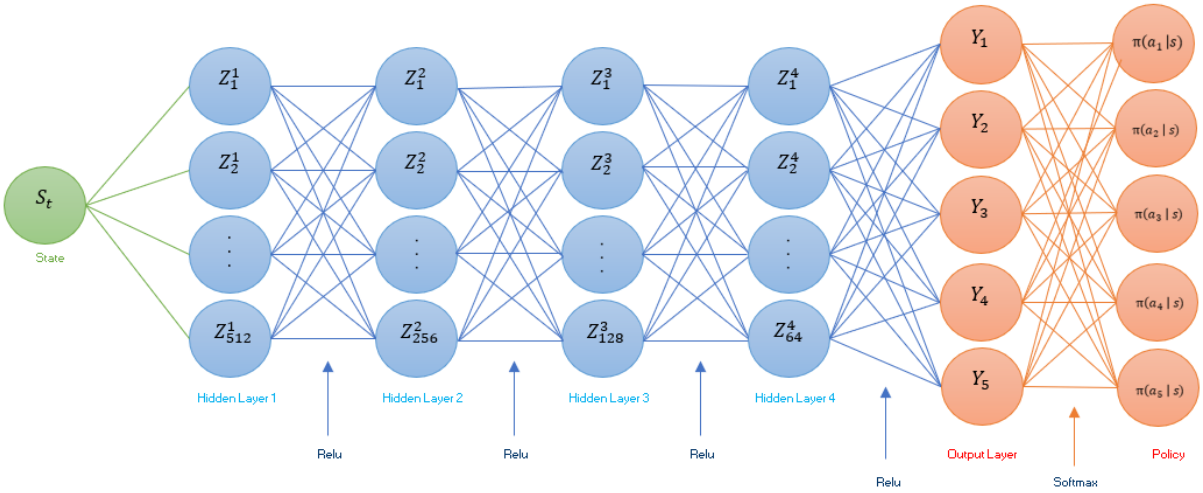


Figure 12: *Illustration of the neural network used in the REINFORCE agent in Part II. It takes an observed state as input, extracts a feature vector, which is fed into 4 hidden layers, then an output layer and finally a softmax layer transforming the output to probabilities.*

The simulation of the environment can be seen in Algorithm 8, where "Env.step(a)" takes in an action a, calculates the reward and returns the next state as well as the reward gained by the agent. The "Agent.take_action(state)" takes in a state where the neural network is used to predict the probabilities for the 5 different possible actions, from which the action is sampled.

To prevent the agent from taking actions that would make its position exceed $h^{\max}$, Huang and Ontañón 2020 proposes four different methods to tackle the problem. One of them is *invalid action masking*: This is a very simple method where we exclude the invalid actions by redistributing their probability mass to the legal actions by changing our output function, in this case the softmax function, in the states where we have illegal

Algorithm 8: Stock Market Simulation

Input:
- REINFORCE Agent: Agent
- Environment class of stock market: Env
- Initial action vector: **a**
- Number of episodes: NE
- Number of steps in episode: SE

Output: Actions, Rewards

```
episode = 1
repeat
    Env.simulate_path
    a = 0
    repeat
        state, reward = Env.step(a)
        a = Agent.take_action(state)
    until until step ≥ SE;
    Train REINFORCE
    episode+ = 1
until until episode ≥ NE;
```

actions, such that our policy is:

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{e^{f_{\boldsymbol{\theta}}(s,a)}}{\sum_{b \in \text{LegalMoves}} e^{f_{\boldsymbol{\theta}}(s,b)}},$$

where f represents the neural network output-layer with parameters $\boldsymbol{\theta}$.

To make sure that the agent keeps trying new actions during training and thereby exploring the environment, we have used a combination of an ϵ-greedy action selection as well as using the softmax as explained above. During the first 100 episodes we ensured that it explored a lot of different states by setting $\epsilon = 0.8$, after these we set $\epsilon = 0.01$

8 Results Part II

We split this results section into three subsections. First, we will show the observations during the training period on the AR(1) + GARCH(1,1) model. Then we will present test results on the test data set. And lastly, we will enforce an obvious statistical arbitrage strategy and show that the agent can learn and exploit it.

8.1 Train Results

The final hyper parameters used in when training the REINFORCE model can be seen in Table 2. Figure 13 shows the actions and corresponding returns (top graphs) and position sizes and corresponding prices (bottom graphs) of four example training episodes on the data simulated by the AR(1) + GARCH(1,1) model. Beginning with episode 50 (far left),

Table 2: *Hyper parameter values for the REINFORCE agent*

Parameter	Value	Explanation
κ	$1e-03$	Variance term multiplier
α	$1e-06$	Learning rate of the NN
cost	0.001	Cost weight
ϵ	0.01	Exploration
k	10	Time steps to include in the state
p_{dropout}	0.1	Percentage of neurons to drop
γ	0.99	Discount factor
λ_{L2}	$1e-04$	L2 regularization weight
$h^{\max}$	10	Maximum exposure

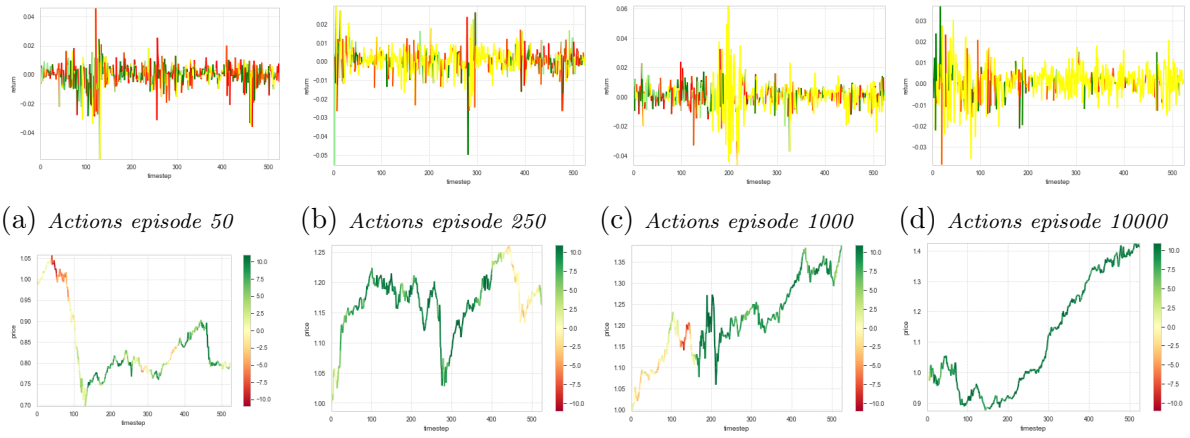
and ending at episode 10000 (far right). From the top graphs we see how the agent has changed its strategy during training from taking somewhat random actions into a buy and hold strategy. This insinuates that the agent have identified the positive drift in the market, but given up on learning the random fluctuations in the short term.

Even after 10000 episodes, there are still periods where the agent does not have a position size of 10 in the market. This is mainly a consequence of our enforced exploration. Moreover we see that the agent has drastically lowered its number of trades per episode, which is influenced both by the trading cost as well as the fact that the agent has shifted towards this buy-and-hold setup. Generally, this behavior is likely to be due to the fact that the S&P-500 index has increased over the period, why the parameters that we have estimated for the AR(1)+GARCH(1,1) model also most of the time have generated paths that providing positive returns, which can be seen in Figure 10.

In table 3 we see the distribution of actions of the same four episodes. We see that agent goes from an almost uniformly distributed policy, to a policy where it in most of the time points had no trades executed. The weight has mainly been moved from the action buy 1 and sell 2 to do nothing, while buy 2 for instance still has a high frequency. The reason why action sell 1 has a high frequency in the last training episode could be explained by the fact that when the agent has a max position, its policy will only distribute probabilities over the sell actions and do nothing action due to the invalid action masking.

Table 3: *Number of actions taken during train episodes*

Action	Episode 50	Episode 250	Episode 1000	Episode 10000
Sell 2	169	63	70	32
Sell 1	96	98	89	78
Do nothing	95	201	205	336
Buy 1	83	91	84	5
Buy 2	82	72	77	74



(a) *Actions episode 50* (b) *Actions episode 250* (c) *Actions episode 1000* (d) *Actions episode 10000*

(e) *Position episode 50* (f) *Position episode 250* (g) *Position episode 1000* (h) *Position episode 10000*

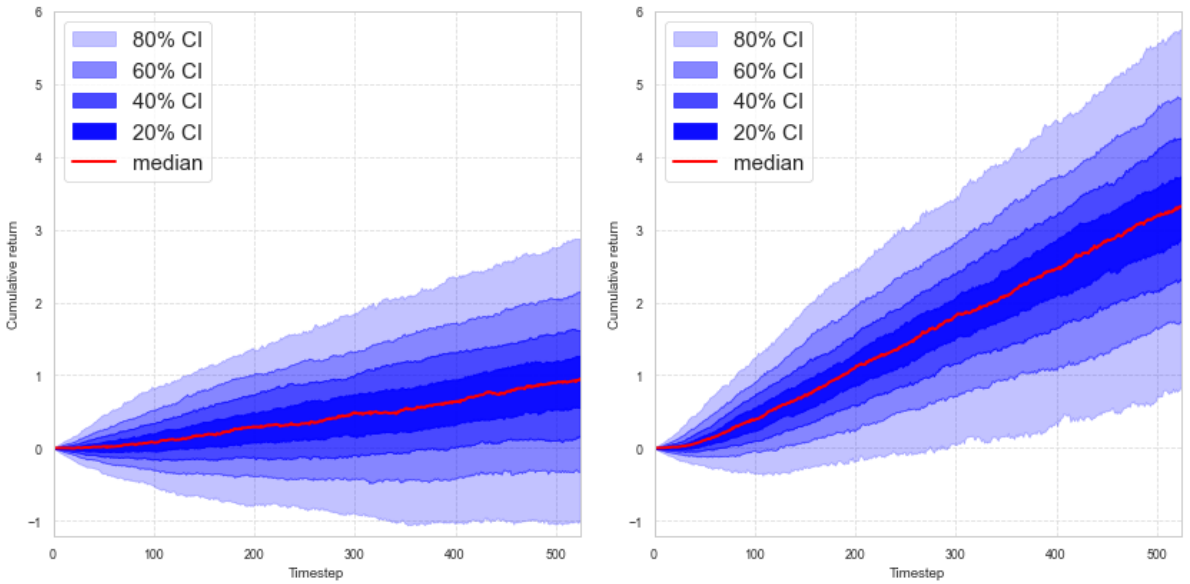
Figure 13: *Training development on the AR(1)+GARCH(1,1) model. We see the REINFORCE agents actions and return from episode 50 (far left), 250, 1000 and 10000 (far right). The agent had the following hyper parameter values:* $\kappa = 1\text{e}{-3}$ *and* $\gamma = 0.99$*. The top graphs shows the actions that the agent have taken at each time step where red indicates shorting, yellow is neutral and green is long and corresponding returns. The bottom graphs shows the agents position at each time step on top of the cumulative price development. The plots are normalized such that the price starts at 1.*

To get a more general idea of how the agent have improved during training we take a look at Figure 14. It shows a fan chart of the distribution of cumulative returns during the first 1000 episodes (a) and during the last 1000 episodes (b). We see that the agent clearly have gotten a higher median reward during the last part of the training than in the beginning, while the variance does not change a lot. This behaviour is due to the agents reward function, prioritizing return over reducing risk, increasing the value of κ would decrease the variance but also decrease the expected return. Nevertheless we tested the agent with different values of κ and found that, $\kappa = 1\text{e} - 03$ seemed like a good choice. If κ was much higher the agent would be too risk sensitive to trade.

8.2 Test Results

When running the final test we set $\epsilon = 0$, and let the agent run for the full period from 2020-01-01 to 2022-04-10. Figure 15 shows the final results where the agent traded the S&P-500 index. Overall it gained a positive return and it acted much like we saw from the test data, where it primarily kept a long position.

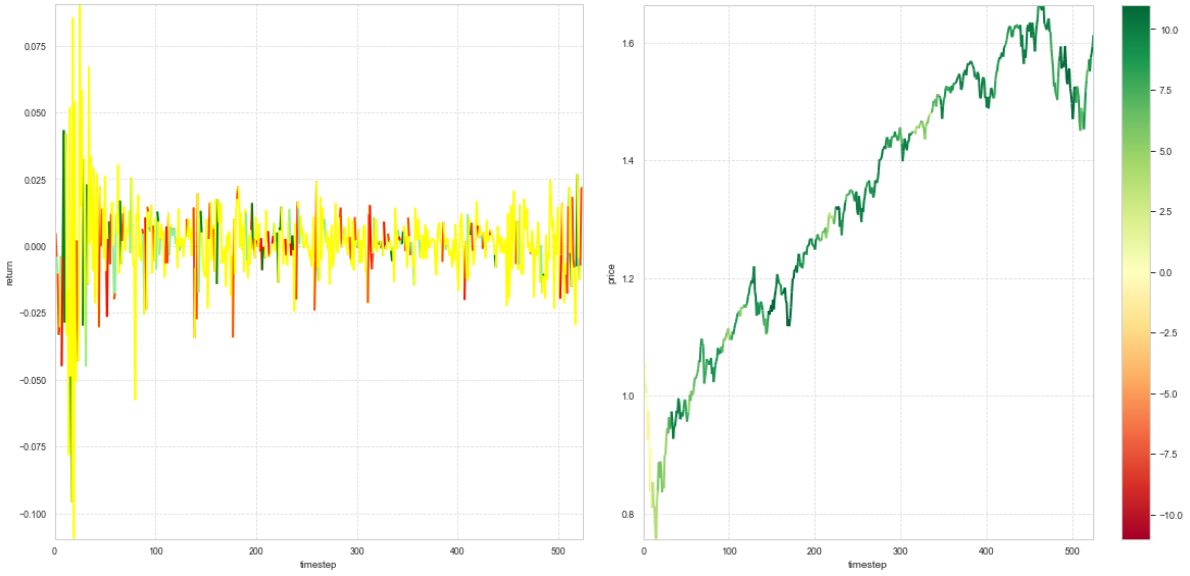
In Table 4 we have compared the REINFORCE agent to a simple buy and hold strategy, where we see that they perform very similarly. We do see that the REINFORCE agents return over the period is a bit lower, but likewise is its risk levels measured in standard deviation, value at risk and conditional value at risk. Overall the two strategies perform very similarly.



(a) *Percentiles of cumulative returns during the first 1000 episodes*

(b) *Percentiles of cumulative returns during the last 1000 episodes*

Figure 14: *Fan charts showing the distribution of cumulative returns training on the AR(1) + GARCH(1,1) model with, for the first (left figure) and last (right figure) 1000 episodes. The blue layers represents confidence intervals represented by the 20, 40, 60 and 80th percentiles of the cumulative returns. We clearly see an increase in return from the first episodes to the last.*



(a) *Actions taken during the test episode*

(b) *Position size during the test episode*

Figure 15: *Plots generated from running the agent on the observed S&P-500 data from 2020-01-01 to 2022-04-01. The left figure shows the actions that the agent took during the episode and the right figure shows the position that the agent was holding during the episode. Further the plots are normalized such that the price starts at 1.*

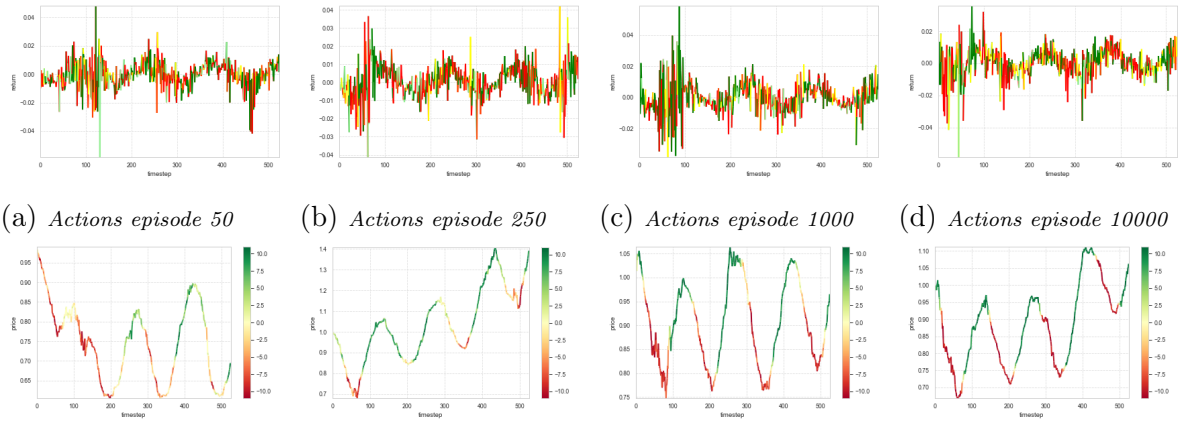
Table 4: *Performance measures during the test period on S&P-500 data from 2020-01-01 to 2022-04-01*

	REINFORCE	buy and hold
Return	4.208074	4.623759
sd	0.149935	0.156679
$\text{VaR}_{0.95}$	0.254030	0.265855
$\text{CVaR}_{0.95}$	0.316682	0.331325

8.3 Sinus Curve Experiment

To further test whether the agent can learn other patterns we have combined the simulated prices with a sinus curve, in order to enforce a price pattern, which can be exploited. We want to test whether the agent can learn an efficient strategy in a market where we know that there is statistical arbitrage opportunities.

In Figure 16 we see example training episodes through time, where the agents actions (top graphs) and position size (bottom graphs) is plotted on top of the returns and cumulative returns respectively. We see that the agent struggle in the beginning (left graphs), while learning an almost perfect strategy in the final episodes (right graphs). However, it still needs some improvement on exactly when to get in and out of the market. Because of the stochasticity in the returns, this is a very hard task to learn.



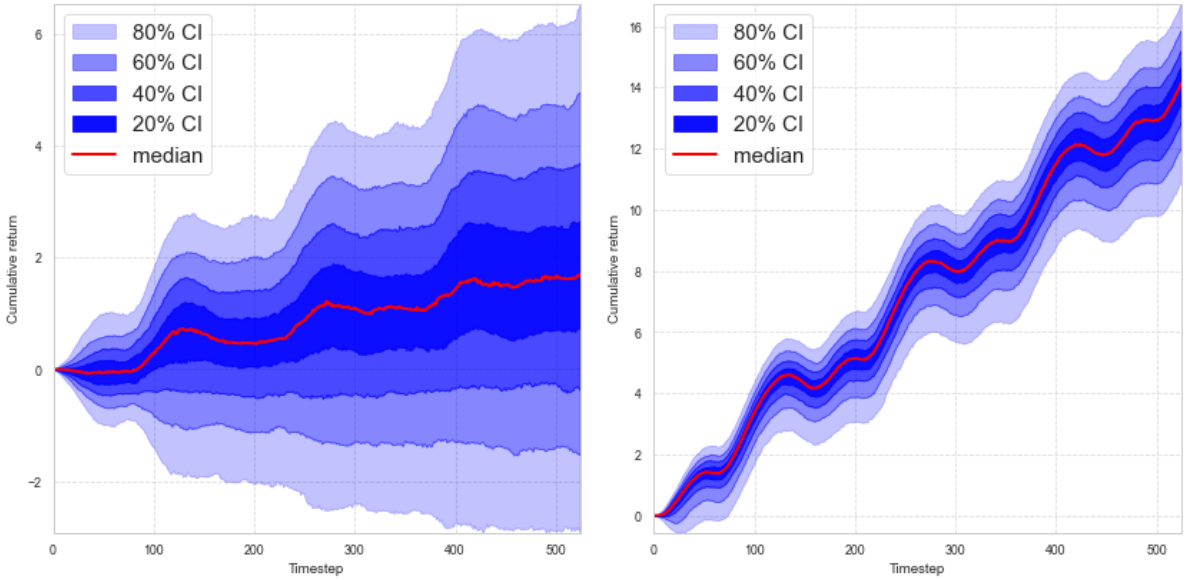
(a) *Actions episode 50* (b) *Actions episode 250* (c) *Actions episode 1000* (d) *Actions episode 10000*

(e) *Position episode 50* (f) *Position episode 250* (g) *Position episode 1000* (h) *Position episode 10000*

Figure 16: *Training development on the AR(1)+GARCH(1,1) model with enforced sinus curve. We see the REINFORCE agents actions and returns from episode 50 (far left), 250, 1000 and 10000 (far right). The top graphs shows the actions that the agent have taken at each time step where red indicates shorting, yellow is neutral and green is long and corresponding returns. The bottom graphs shows the agents position at each time step on top of the cumulative price development. The plots are normalized such that the price starts at 1.*

In Figure 17 we again see fan charts showing the distributions of cumulative returns

during the initial 1000 (left figure) and last 1000 (right figure) training episodes. We see that the agent has indeed learned a systematic strategy that provides a high return. Furthermore we notice that the returns are not only higher but also more stable. Hence the agent has both learned to increase its return and decrease its variance in returns at the same time.



(a) *First 1000 episodes* (b) *Last 1000 episodes*

Figure 17: *Fan charts showing the distribution of cumulative returns training on the AR(1) + GARCH(1,1) model with enforced sinus curve, for the first (left figure) and last (right figure) 1000 episodes. The blue layers represents confidence intervals represented by the 20, 40, 60 and 80th percentiles of the cumulative returns. We clearly see an increase in return and a decrease in variation.*

This 'market' is obviously unrealistic, but it serves as a proof of concept that the agent can identify and exploit patterns in a market.

9 Discussion Part II

In this study, we simulated market prices with an AR(1)+GARCH(1,1) model because it is a simple yet widely used model to model financial time series. However, it could be worthwhile to do some testing on the performance of the REINFORCE agent when using a more sophisticated model. Such a model could either be a AR+GARCH model with more lags or an extension such to the GARCH model, such as EGARCH. Using a more sophisticated model could lead to more realistic simulations of the S&P-500 index, but one must be careful, since we still need to be able to generalize.

The agent we implemented was the REINFORCE agent, which proved to create a strategy that performed on par with a buy and hold strategy when trading the S&P-500 index out of sample.

To build on the environment, one could expand it with a risk-neutral asset with a positive interest rate or even one or multiple other assets that the agent could invest in. This would require a rework of the agent's actions, where one solution could be to expand the action space to a continuous one with weights assigned to each asset.

10 Conclusion Part II

In this experiment, we showed how the REINFORCE algorithm can be used to create a risk-averse systematic trading strategy. We trained the agent on simulations from a AR(1) + GARCH(1,1) model with parameters estimated on the S&P 500 index from 2010 until 2020. During the training period, we saw that its strategy converged to a buy and hold strategy: the classic long-term investment strategy you find on page one in most finance books. This is not a surprising result since we trained it using a AR + GARCH model, which has random increments through time. It reflects that the agent has learned the long term drift in market prices, and given up predicting the random small changes through time. A test on the following period 2020 until 2022-04-01, showed that the agent performed similarly to a fixed buy and hold strategy, with a return of 4.2, slightly lower than the return of 4.6 achieved by the buy and hold strategy. As a compensation it achieved a lower standard deviation of returns at 0.15 vs 0.16 by the buy and hold strategy. Furthermore as a proof of concept we enforced a sinus curve on the simulated price and showed that the agent was able to learn and exploit the statistical arbitrage strategy, which it entailed.

Part III
Market Making As a Multi-Agent Reinforcement Learning Problem

11 Background

In recent years technology and systematic trading strategies have become more and more dominating in the financial markets. This is true for all market participants, but especially for market makers, and high frequency trading firms. For theses firms, as in any other area within finance, managing risk and optimizing return subject to risk is crucial.But in contrast to long term investing, their risk exposures lie within the small intraday changes in the financial markets. They do have an impact on these changes because they are constantly interacting within them and setting the prices. Because actions in the markets provoke reactions from other market participants and thus affect the future development of the market, intertemporal choice is essentially at the heart of decision making for any market participant. And thus, with RLs recent success at solving problems of intertemporal choice, RL has become a popular area of interest for optimizing systematic trading strategies in the financial markets. A key problem though, is how to simulate these financial markets in order to back test and evaluate these intraday strategies. In this study we create an environment, where it is possible to test the performance of systematic intraday trading strategies and optimize them with RL agents.

Through historical simulation, it is possible to evaluate actions made by agents at specific time points, but the effect these actions have on the future development of the market will not be reflected therein. For that reason historical simulation is simply not an efficient way to evaluate intraday systematic trading strategies. In agent-based models (ABMs), financial markets can be simulated through the interactions of agents representing the market participants, who act according to their own strategies or reward functions. This concept reflect the dynamics of the true financial markets quite well. We can for example relax on assumptions often made when modelling markets, such as no market impact and that the market is frictionless. Though, a dilemma is how to get the agents to reflect the characteristics of real market participants, i.e. create the agents such that their behavior reflect the behavior of the real market participants. A natural way to measure this reflection is by comparing descriptive statistics of the real markets and the pseudo market from the ABM simulation. Recall that the descriptive statistics of the financial markets are known as *the stylized facts* (Cont 2001). We will use these to calibrate our ABM to reflect the true markets as close as possible.

Firstly we present an ABM representation of a financial stock market and the actors/agents within it. We then evaluate and calibrate our ABM to a real financial market using the stylized facts (ibid.). Afterwards we present the RL algorithm built to optimize market making and finally we present the results achieved and conclude based on these.

11.1 Related Work

An example within finance, where ABMs have been widely used is in the replication of the the famous flash crash in 2010, where the American stock market dropped by 9% within minutes, an extreme scenario never seen before. In this case ABMs are used to understand which dynamics, actors, signals and actions that may have caused the crash. See for example (Vuorenmaa and Wang 2014). It is likely that ABMs will also be used to replicate the recent European flash crash 3rd of may 2022.

With the recent success of RL as a tool to solve dynamic decision problems, e.g. the first go-computer to ever beat the world champion (Silver et al. 2017), it has also seen increase of interest for researches within finance. Due to the structure of RL, ABMs are a natural choice to simulate financial environments to train RL agents within. Recent research include multiple articles from J.P. Morgan AI research (Vyetrenko et al. 2019; Ardon et al. 2021 and Ganesh et al. 2020), studying the combination of RL and ABMs ability to replicate realistic market dynamics and optimal hedging strategies. In the first and second paper they present the objective of applying ABMs with RL agents to replicate stylized facts by calibrating to real data through the reward functions these RL agents. In the third they train RL agents to perform market making in an environment with prices simulated through a statistical model. Other interesting recent papers within the field are also studying realistic market simulation, specifically the dynamics of the order book (Karpe et al. 2020; Maeda et al. 2020). In a recent presentation Thomas Spooner from J.P. Morgan AI Research says that they currently are focusing a lot of their research on combing ABMs and RL to analyze trading strategies (Spooner 2021). In this part of the thesis we will do exactly that.

12 The Agent-Based Model Simulation Framework

An ABM is a method which simulates a complex environment composed of multiple agents, which interact inside an environment. Agents in this context are usually similar to the agents presented in part I, but without the ability to learn. They observe states and takes actions based on these states, which affects the environment and thus the future states. Agents thus also affect each other, i.e. an action by one agent may affect future actions by other agents. In ABMs, the agents usually follow a fixed policy/strategy and do not apply RL to estimate parameters attempting to improve their performance within the environment.

Using ABMs enables researchers to built, experiment with and analyze scenarios, which may or may not have happened in reality. Within finance the obvious objective is modelling the financial markets. Especially when it comes to modelling the underlying dynamics of high frequency data, i.e. time series with small time intervals e.g. 1 min, 30 sec or even 1 sec between data points. When modelling financial markets with ABMs, the agents represents the actors in the market, e.g. institutional investors, market makers, hedge funds, retail investors etc., and the environment represents the financial market itself.

12.1 The Order Book

Before going into details about how the environment and the agents are constructed, we will briefly go through how trades are being executed on stock exchanges through the order book.

The order book is a key component at the heart of the financial markets, controlling what goes in and out of a given exchange. It holds all current orders listed on the exchange. The orders contain information, such as the price, volume, and whether the trader wishes to buy or sell. Thus the order book also includes an overview of the volume of buy and sell orders currently existing in the market and their corresponding prices. Bid prices refer to buy orders and ask prices to sell orders. The bid price and ask price often refer to the best bid and best ask price. They are respectively defined as the best price to sell at and the best buy price to buy at in the current market, i.e., the highest buy price and the lowest sell price. The gap between the bid and ask price is called *bid-ask spread*.

If two traders agree on a specific price, i.e a trader wish to buy at a given price and another wish to sell at the same price or lower, the order book will match the traders resulting in a trade. If there are two orders in the market with the same price, the order which entered the market first will get priority. This is known as *price-time priority*.

There are two main order types: a *limit order* and a *market order*. With limit orders the trader places an order to trade at a given limit or better; for instance to buy for a particular price or lower, or to sell for a particular price or higher. The limit order remains in the market until it is matched or the trader withdraws it off the market. with market orders the trader places an order to trade at the current best price, i.e to buy at the ask price or to sell at the bid price. Thus it will trade immediately (as long as there are any orders of the opposing site in the order book). A dilemma with this choice is that the trader will not know the actual trading price. Prices can change within nanoseconds and thus a price which appear on the screen can change just as the trader places her market order. With the limit order, there is a limit to how 'bad' a trade can be.

In figure 18 we illustrate an example order book. It shows how the order book is built up and how the best bid and ask prices are interpreted as well as the bid-ask spread.

12.2 The Environment

As mentioned the environment is representing a financial market. For simplicity we let the market be a stock exchange where a single stock, Stock A, is traded and can be either bought or sold. Furthermore we let the environment be episodic, where an episode can be seen as a full day of trading. An episode is run over 500 time points at which the agents will observe a state and be able to place orders.

We let N^{agents} be the number of agents in the model, i.e. the number of actors in the market. Furthermore we let the $\mathbf{a}_t^{(i)} \in \mathbb{R}^4$ be the action taken by agent i at time t,

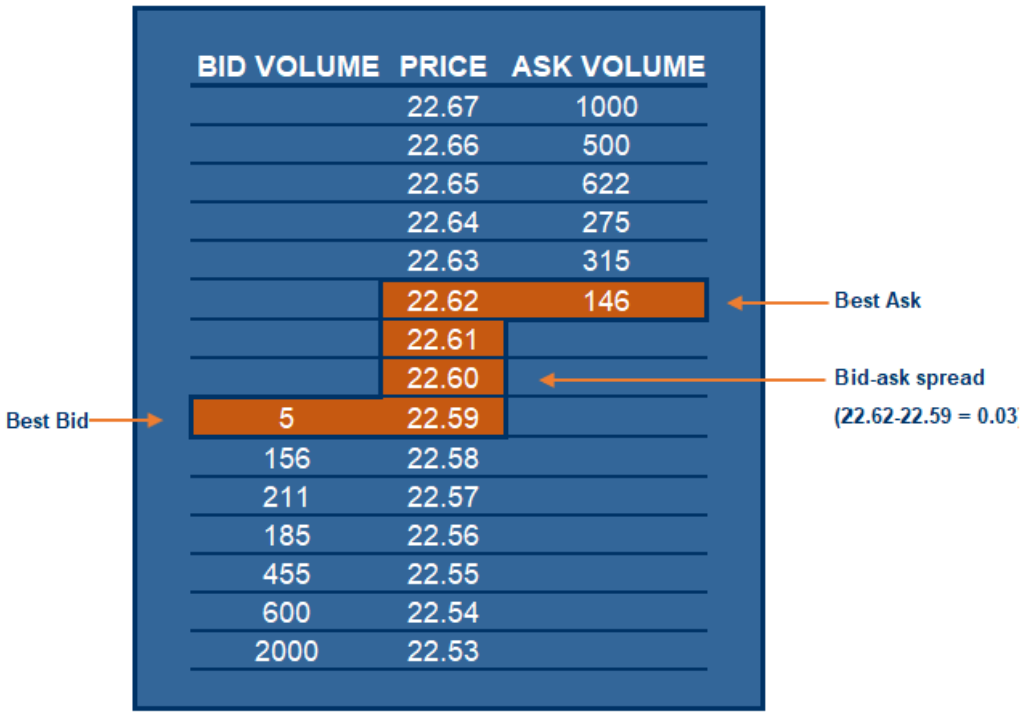


Figure 18: *Example of a filled order book, with prices ranging from 22.53 to 22.67. In the example the best bid price is 22.59 with a volume of 5. The best ask price is 22.62 with a volume of 146. And the bid-ask spread is 0.03.*

defined as

$$\mathbf{a}_t^{(i)} = \left[\mathbf{bp}_t^{(i)} \quad \mathbf{sp}_t^{(i)} \quad \mathbf{bv}_t^{(i)} \quad \mathbf{sv}_t^{(i)}\right],$$

where $\mathbf{bp}_t \in \mathbb{R}_+^{N^{agents}}$, $\mathbf{sp}_t \in \mathbb{R}_+^{N^{agents}}$, $\mathbf{bv}_t \in \mathbb{N}_0^{N^{agents}}$, $\mathbf{sv}_t \in \mathbb{N}_0^{N^{agents}}$ are vectors holding the actions of each agent at time t and respectively represent buy price, sell price, buy volume and sell volume. Note that we assume no negative buy or sell prices.

12.2.1 Order Matching

The action of an agent at time t defines the agents order to the market at time t. Therefore when we refer to the order of an agent, we directly refer to the composition of the agents action.

In the real market, order placement happens on a continuing basis, but for practical purposes we have approximated it by discrete time steps in which agents can submit orders (actions). At each time step t the agents orders are being submitted to the order book, from which the trades are being matched. The orders placed by the agents are limit orders, which gets updated after each time step. If a buy order is submitted with a limit a lot higher (lower if selling) than the current market price, the order will effectively work as a market order. To compensate for the lag of a continuous market and reflect the fact that orders does not hit the market at the exact same time, we let each agent have a latency, which changes through time. The agents orders then hit the market in

a sequence from lowest to highest corresponding to the agents latency. This way if two agents has similar prices, the agent with the lowest latency has the highest probability of getting its orders executed. This also opens up to the possibility of letting specific types of traders consistently have lower latency than others, which is the case in the real world.

Note that we will at each time step have multiple orders being matched. All these matches we interpret as our pseudo high frequency data set. Thus even though we only simulate 500 time steps, we could easily, with enough agents and volumes in the market, observe a lot more than 500 prices.

12.2.2 State Definition

Notice that due to the structure of the environment, the agents are only able to observe the market after a full order book matching. For this reason we need a reference market price reflecting the price after a sequence of matches in the order book. Letting $\mathbf{mp}_t$ and $\mathbf{mv}_t$ denote vectors with elements respectively equal to the matched prices and corresponding volumes at time step t. We then define P_t as the volume weighted median trade price at time t. That is the median in the cumulative distribution function

$$F_t(p) = \frac{\sum_{j=0}^{N^{trades}} \mathbf{mv}_{t,j} \mathbb{1}_{\{p \leq \mathbf{mp}_{t,j}\}}}{\sum_{j=0}^{N^{trades}} \mathbf{mv}_{t,j}},$$

where ${N^{trades}}_t$ denote the number of trades that happened at time t. The distribution is discrete and thus have either zero or infinitely many solutions. Thus, we in practice use a linear interpolation between the jumps to enforce a unique output. Our reason for using the median instead of an average is related to the fact that we desire the environment to reflect real market dynamics. The issue we find is that the average method easily drag prices too far in either direction, because a few trades happen a bit far from the mid price. We have observed similar results if the reference price was the last traded price at the time step. Of course, this is also affected by other decisions, such as the agent's attributes and parameters. To further enrich the state representation we define the following metrics

$$\mathrm{TMV}_t = \sum_{j}^{{N^{trades}}_t} \mathbf{mv}_{t,j} \qquad \text{(Total Matched Volume)}$$

$$\mathrm{TBV}_t = \sum_{i}^{{N^{agents}}_t} \mathbf{bv}_t^{(i)} \qquad \text{(Total Buy Volume)}$$

$$\mathrm{TSV}_t = \sum_{i}^{{N^{agents}}_t} \mathbf{sv}_t^{(i)} \qquad \text{(Total Sell Volume)}$$

$$\mathrm{MBP}_t = \frac{1}{\mathrm{TBV}_t} \sum_{i=1}^{{N^{agents}}_t} \mathbf{bp}_t^{(i)}\, \mathbf{bv}_t^{(i)} \qquad \text{(Mean Buy Price)}$$

$$\mathrm{MSP}_t = \frac{1}{\mathrm{TSV}_t} \sum_{i=1}^{{N^{agents}}_t} \mathbf{sp}_t^{(i)}\, \mathbf{sv}_t^{(i)} \qquad \text{(Mean Sell Price)},$$

from which we together with the reference market price define the current state of the environment at time t:

$$S_t = \{\mathrm{P}_0,\ \mathrm{P}_1, ...,\ \mathrm{P}_{t-1},\ \mathrm{TMV}_{t-1},\ \mathrm{MBP}_{t-1},\ \mathrm{MSP}_{t-1},\ \mathrm{TBV}_{t-1},\ \mathrm{TSV}_{t-1}\}.$$

All the agents observe the state S_t and based on the observation they select actions $\mathbf{a}_t$ according to their policies.

As well as the state, the environment includes the parameters *slippage* and *fee*, which respectively represent a slippage cost used to calculate profit and loss at given time points, as well as a market fee for buying and selling on the exchange. The slippage refers to the loss associated with liquidating a position. It is reflected in the difference between the current market price and the actual trade price, e.g. due to market spread. If the order is large, it could be due to the order moving the market in the unfavorable direction.

12.2.3 Simulating The Environment

We now have all the ingredients to compose and simulate the ABM of our stock exchange. The simulation works as follows: At each time point the agents, which we present later in more detail, submit their actions; the buy and sell prices as well as corresponding volumes. When prices have been submitted, the exchange matches the buyers and sellers through a pseudo *order book* and send back information to the agents as to whether they have made any trades, and if so at which prices and volumes. Afterward, the state gets updated according to the latest matches, and the agents will be able to update their actions, and the environment continues in this circular manner until desired. The structure of the simulation algorithm is shown in algorithm 9.

Algorithm 9: ABM Simulation

Input:
- List of agent objects: agents
- Environment class of financial market: Env
- Initial state: state_0
- Initial 4 dimensional action vector: $\mathbf{a} \in \mathbb{R}^{N^{agents} \times 4}$
- Number of episodes: NE

Output: Observed Financial Market

episode = 1
repeat
  /* Initialize env and agents orders */
  env = Env(state_0)
  agents_temp ← agents
  /* Run one full episode */
  for $i \in N^{agents}$ **do**
    $\mathbf{a}_i = \text{agents\_temp}_i.update\_orders(\text{state}_0)$
  end
  for $t \in 0, 1, ..., T$ **do**
    state ← env.$step(\mathbf{a})$
    for $i \in N^{agents}$ **do**
      $\mathbf{a}_i \leftarrow \text{agents\_temp}_i.update\_orders(\text{state})$
    end
  end
  episode+ = 1
until *until episode* $\geq$ *NE*;

12.3 The Agents

In the ABM we have split the actors of the market into 4 classes (5 if we include the RL agent): Random, Investor, Trend Follower and Market Maker. Each class has their own strategy/policy, which they follow when placing orders.

12.3.1 The Random Agent

The first agent class is *the random agent*, which given in the name, represents random actors who places noisy orders to the market. This group of agents represents the retail investors, who 'randomly' decides whether they want to enter (long or short) the market of stock A. The action of a random agent at time t is given by

$$\mathrm{a}_t^{(\mathrm{R})} = \begin{bmatrix} \mathrm{bp}_t^{(\mathrm{R})} & \mathrm{sp}_t^{(\mathrm{R})} & \mathrm{bv}_t^{(\mathrm{R})} & \mathrm{sv}_t^{(\mathrm{R})} \end{bmatrix}.$$

$\text{bp}_t^{(\text{R})}$ and $\text{sp}_t^{(\text{R})}$ are respectively the buy price and sell price and calculated as

$$\text{bp}_t^{(\text{R})} = \max\left\{\bar{P}_t^{\text{R}}\Big(1 - U(lb^{(\text{R})}, ub^{(\text{R})})\Big), 0\right\}$$
$$\text{sp}_t^{(\text{R})} = \max\left\{\bar{P}_t^{\text{R}}\Big(1 + U(lb^{(\text{R})}, ub^{(\text{R})})\Big), 0\right\},$$

where

$$\bar{P}_t^{(\text{R})} = \text{P}_{t-1}\Big(1 + N(0, \sigma^{(\text{R})})\Big).$$

$U(lb^{(\text{R})}, ub^{(\text{R})})$ is a uniform distribution, which creates a spread around the random agents mid price, $\bar{P}_{t-1}^{(\text{R})}$. $lb^{(\text{R})}$ and $ub^{(\text{R})}$ represent parameters for lower-and upper bounds. The distribution is subtracted when buying and added when selling, which secures that the agent does not place orders irrationally, where the buy price is higher than the sell price. $\bar{P}_t^{(\text{R})}$ is a random mid price centered around the market price calculated at each time step and with randomness defined by the parameter $\sigma^{(\text{R})}$.

The volumes $\text{bv}_t^{(\text{R})}$, $\text{sv}_t^{(\text{R})}$ are drawn from a binomial distribution, i.e

$$\text{bv}_t^{(\text{R})} = B(n^{(\text{R})}, p_b^{(\text{R})})$$
$$\text{sv}_t^{(\text{R})} = B(n^{(\text{R})}, p_s^{(\text{R})}).$$

The parameters can be tweaked to move the market in a specific direction. For example increasing $p_s^{(\text{R})}$ relative to $p_b^{(\text{R})}$ would increase the volume of sell orders compared to buy orders and thus drive the market in a negative direction.

12.3.2 The Investor Agent

Another typical actor that we observe in the financial markets is institutional investors like pension funds, hedge funds and large asset managers. We will refer to this group of actors as investors and attempt to model them through *the investor agent* class. Institutions like pension funds often have a policy, which denies them from taking on negative position (no short selling), whereas hedge funds often use short selling in their strategies. When investors enter the market it is usually with the intent of taking on a large position (positive or negative) to speculate on the development of the asset over a longer horizon. If the investor went on to place their full order, they would likely clear the entire order book instantaneously and move the market heavily in an unfavourable direction. Thus making it more expensive to fulfill their desired position. Thus, investors usually enter a market using so called execution strategies, which seek to get a position while influencing the market price as little as possible. A typical execution strategy for investors, and also the one our investor agent is applying, is the so called *time slicing* strategy (Durbin 2010). Here the investor places small orders over time, for example with a fixed time sequence or with random time increments. When an investor is taking on a position (buying up or selling out), the same investor can not enter the market again, i.e. start a new sequence of orders. To model this behavior, we let the investor agent have an intensity parameter $\lambda^{(\text{I})}$, which defines the probability that the investor will enter the market, given that they are not already taking on a position.

We let the probability that an investor agent enters the market be modelled through the binomial distribution, i.e:

$$\mathbb{P}\Big(\text{Enter market} \mid \text{Not currently taking on position}\Big) = B\Big(1, \lambda^{(\mathrm{I})}\Big). \tag{14}$$

The investor also has a boolean argument CanShort, which determines if the investor can short sell or not. If the investor can short, it will attempt to short sell just after it attempts to go long, both with success rate given by eq. (14). Note that if the investor at time t starts buying, it will not attempt to sell as well.

When an investor is in the market at time t it will output an action of the same form as the other agents. The action of an investor agent at time t is given by

$$\mathrm{a}_t^{(\mathrm{I})} = \begin{bmatrix} \mathrm{bp}_t^{(\mathrm{I})} & \mathrm{sp}_t^{(\mathrm{I})} & \mathrm{bv}_t^{(\mathrm{I})} & \mathrm{sv}_t^{(\mathrm{I})} \end{bmatrix}.$$

The prices are calculated as

$$\mathrm{bp}_t^{(\mathrm{I})} = \max\Big\{\mathrm{P}_{t-1}\Big(1 - m_b^{(\mathrm{I})}\Big), 0\Big\} \tag{15}$$

$$\mathrm{sp}_t^{(\mathrm{I})} = \max\Big\{\mathrm{P}_{t-1}\Big(1 + m_s^{(\mathrm{I})}\Big), 0\Big\}, \tag{16}$$

where $m_b^{(\mathrm{I})} \in [0,1]$ and $m_s^{(\mathrm{I})} \in [0,1]$ which are buying and selling margin parameters respectively. They define a margin, as to how far in the unfavorable direction from the last observed market price the investor is willing to trade.

Since the investor is using time slicing, we have the parameters $n_{bo}^{(\mathrm{I})}$ and $n_{so}^{(\mathrm{I})}$, which sets the number of orders the investor will place in a row when it is in the market. The volumes of each order is given by the volume actions $\mathrm{bv}_t^{(\mathrm{I})}$ and $\mathrm{sv}_t^{(\mathrm{I})}$, which are constant and given by

$$\mathrm{bv}_t^{(\mathrm{I})} = v_b^{(\mathrm{I})}$$
$$\mathrm{sv}_t^{(\mathrm{I})} = v_s^{(\mathrm{I})}$$

Thus if an investor enters the market to buy at time t, with $n_{bo} = 10$ and $\mathrm{bv}_t^{(\mathrm{I})} = 5$. Then the investor will place 10 buy orders each with a volume of 5 at time $t, t+1, ..., t+9$ with prices given by eq. (15). The execution algorithm for investor agents, in a simplified version where it is only possible to buy stocks, is shown in algorithm 10.

Algorithm 10: Investor Execution Algorithm (no selling)

Input:
- can short sell: CanShort = False
- Intensity: $\lambda^{(\mathrm{I})}$
- Buy margin: $m_b^{(\mathrm{I})}$
- Number of buy orders: n_{bo}
- Buy volume: $volume_b^{(\mathrm{I})}$

1 **Initialize**
2 $is\_buying = 0$
3 $orders\_in\_queue = 0$
4 **for** $t \in 0, 1, ..., T$ **do**
5 **if** $is\_buying == 1$ **then**
6 $orders\_in\_queue \leftarrow orders\_in\_queue - 1$
7 $\mathrm{bp}_t^{(\mathrm{I})} = \max\left\{\mathrm{P}_{t-1}\left(1 - m_b^{(\mathrm{I})}\right), 0\right\}$
8 $\mathrm{a}_t^I = \left[\mathrm{bp}_t^{(\mathrm{I})} \quad NaN \quad volume_b^{(\mathrm{I})} \quad NaN\right]$
9 **if** $orders\_in\_queue == 0$ **then**
10 $is\_buying = 0$
11 **end**
12 **else**
13 $is\_buying = B(1, \lambda^{(\mathrm{I})})$
14 **if** $is\_buying == 1$ **then**
15 $orders\_in\_queue = n_{bo}$
16 Go to line 5.
17 **end**
18 **end**
19 **end**

12.3.3 The Trend Agent

The third agent in our model is *the trend agent*, which represents predictors in the financial markets. Predictors are actors, which attempts to predict the short or mid term development of the asset, often through statistical analysis. The trend agent can go both long and short, and has a strategy based on trends or momentum in the market. The trend is defined by the relation between two moving averages

$$\mathrm{Trend}_t = \frac{\frac{1}{ma_1^{(\mathrm{T})}} \sum_{i=t-ma_1^{(\mathrm{T})}}^{t-1} \mathrm{P}_i}{\frac{1}{ma_2^{(\mathrm{T})}} \sum_{i=t-ma_2^{(\mathrm{T})}}^{t-1} \mathrm{P}_i},$$

where $ma_1^{(\mathrm{T})}$ and $ma_2^{(\mathrm{T})}$ $(ma_1^{(\mathrm{T})} < ma_2^{(\mathrm{T})})$ defines the length of the two moving averages.

If Trend ≥ 1 the trend agent wants to be long in the stock, and if Trend < 1 it wants

to be short. The position size (positive if long and negative if short) is given by a target position parameter $th^{(\mathrm{T})}$. This target position reflects the position size that the trend agent wants to have in the market; either long or short. The agent will buy or sell stocks until its absolute position size is equal to $th^{(\mathrm{T})}$.

$$\mathrm{a}_t^{(\mathrm{T})} = \begin{bmatrix} \mathrm{bp}_t^{(\mathrm{T})} & \mathrm{sp}_t^{(\mathrm{T})} & \mathrm{bv}_t^{(\mathrm{T})} & \mathrm{sv}_t^{(\mathrm{T})} \end{bmatrix},$$

where the buy and sell volumes are given by an execution algorithm, which is designed such that the order volume becomes the difference between the desired position size, $th^{(T)}$, and the agents current position size. Thus letting h_t define the current position size we get:

$$\mathrm{bv}_t^{(\mathrm{T})} = \mathbb{1}_{\{\mathrm{Trend}>1\}}(th^{(\mathrm{T})} - h_t)$$
$$\mathrm{sv}_t^{(\mathrm{T})} = \mathbb{1}_{\{\mathrm{Trend}<1\}}(th^{(\mathrm{T})} + h_t)$$

This way the agent will never get a position larger than desired, and it will often be able to change position quickly since the target position $th^{(T)}$ is not meant to be very large. The prices are calculated as

$$\mathrm{bp}_t^{(\mathrm{T})} = \max\left\{\mathrm{P}_{t-1}\left(1 - m^{(\mathrm{T})}\right), 0\right\} \tag{17}$$
$$\mathrm{sp}_t^{(\mathrm{T})} = \max\left\{\mathrm{P}_{t-1}\left(1 + m^{(\mathrm{T})}\right), 0\right\}, \tag{18}$$

Where $m^{(\mathrm{T})} \in [0, 1]$ is a margin parameter, just like the one investors have, which determines the agents willingness to trade in the unfavorable direction of the market price.

12.3.4 The Market Maker

The last predefined agent type in the market is the market maker. Market makers are liquidity providers, which means that they add liquidity to the order book. Opposite the investor agent and trend agents, which desires to get positions and thus take away liquidity from the order book. Market makers earn their money by constantly having both buy and sell orders in the market, but with a spread around a target mid price. The idea is to have a 'high' sell price and a 'low' buy price and then constantly trade at the two prices providing a small steady return. The aim is to keep a market neutral position, i.e a position close to 0, also opposite the trend and investor agents. This is usually a fairly low risk strategy, but it is obviously sensitive to large price moves. If for instance the price increases heavily over a small period, market makers will likely sell at their sell price, but since the price will follow up they will not get their buy orders executed. A common market maker strategy is *lean your market*, which corrects the market makers mid price in the direction, which increases the probability of trading towards a market neutral position. At time t the action of a market agent is defined as

$$\mathrm{a}_t^{(\mathrm{M})} = \begin{bmatrix} \mathrm{bp}_t^{(\mathrm{M})} & \mathrm{sp}_t^{(\mathrm{M})} & \mathrm{bv}_t^{(\mathrm{M})} & \mathrm{sv}_t^{(\mathrm{M})} \end{bmatrix}.$$

The volumes are constant and equal at each time step

$$\mathrm{bv}_t^{(\mathrm{M})} = \mathrm{sv}_t^{(\mathrm{M})} = v^{(\mathrm{M})},$$

and the prices are set as

$$\mathrm{bp}_t^{(\mathrm{M})} = \max\left\{\bar{P}_t^{(\mathrm{M})} - \frac{\zeta_t^{(\mathrm{M})}}{2}, 0\right\}$$
$$\mathrm{sp}_t^{(\mathrm{M})} = \max\left\{\bar{P}_t^{(\mathrm{M})} + \frac{\zeta_t^{(\mathrm{M})}}{2}, 0\right\}.$$

$\bar{P}_t^{(\mathrm{M})}$ and $\zeta_t^{(\mathrm{M})}$ are functions respectively representing the market makers mid price and spread, which are calculated as

$$\bar{P}_t^{(\mathrm{M})}(\mathrm{P}_{t-1}, h_t^{(\mathrm{M})}) = \mathrm{P}_{t-1}\left(1 - \gamma_1^{(\mathrm{M})}\, h_t^{(\mathrm{M})}\right)$$
$$\zeta_t^{(\mathrm{M})}(\sigma_t^{(\mathrm{M})}) = \sigma_{t-1}^{(\mathrm{M})}\, \gamma_2^{(\mathrm{M})} + c^{(\mathrm{M})},$$

where $c^{(\mathrm{M})}$ is a constant base-spread, $h_t^{(\mathrm{M})}$ is the agents current position. $\gamma_1^{(\mathrm{M})} \in [0, 1]$ and $\gamma_2^{(\mathrm{M})} \in \mathbb{R}_+$ are sensitivity parameters controlling the market makers level of risk aversion. $\gamma_1^{(\mathrm{M})}$ determines mid price sensitivity to position size, which comes from the market leaning strategy. The effect is that if the position is positive the mid price decreases, which increases the likelihood of selling and decreases the likelihood of buying, motivating a market neutral position. $\gamma_2^{(\mathrm{M})}$ determines the spread sensitivity to volatility. Thus the agent will create larger spreads and protect itself against large price changes when the volatility in the market is high. $\sigma_t^{(\mathrm{M})}$ is an estimate of the volatility calculated as the unbiased sample standard deviation of the prices over the last $n_\sigma^{(\mathrm{M})}$ time steps:

$$\sigma_t^{(\mathrm{M})} = \sqrt{\frac{\sum_{i=t-n_\sigma^{(\mathrm{M})}}^{t-1}\left(\mathrm{P}_i - \bar{\mathrm{P}}\right)^2}{n_\sigma^{(\mathrm{M})} - 1}}, \qquad \bar{\mathrm{P}} = \frac{1}{n_\sigma^{(\mathrm{M})}} \sum_{i=t-n_\sigma^{(\mathrm{M})}}^{t-1} \mathrm{P}_i.$$

12.3.5 Common Agent Attributes

Latencies

When orders are matched in the order book, the matching sequence depends on the agents latencies. The latency parameter is inspired by Vuorenmaa and Wang 2014. It is supposed to reflect the aspect that, in reality, actors in the financial markets do not have the same speed to access the market. Often market makers achieve the fastest speed (lowest latency) through super computers physically located close to the actual exchanges. Furthermore it seems reasonable to assume that investors and predictors have better facilities to access the market than retail investors. We therefore define the following latency distributions

$$\delta_t^{(\mathrm{M})} = U(0, 1)$$
$$\delta_t^{(\mathrm{I})} = \delta_t^{(\mathrm{T})} = U(1, 2)$$
$$\delta_t^{(\mathrm{R})} = U(2, 3).$$

Thus market makers hit the order book first, followed by trend followers and investors, and lastly the random actors. The latencies are given by uniform distributions such that

a market maker, who may have the lowest latency at time t, does not necessarily have the lowest latency at time $t+1$.

Calculating Profit and Loss
To measure the performance of the agents we can compare their profit and loss (PnL). The profit and loss for each agent at time t is calculated as

$$\mathrm{PnL}_t = \mathrm{CF}_t + \begin{cases} h_t \, \mathrm{P}_t \, (1 - \mathit{slippage}), & h_t > 0 \\ h_t \, \mathrm{P}_t \, (1 + \mathit{slippage}), & h_t < 0 \end{cases}$$

where $\mathrm{CF}_t \in \mathbb{R}$ is the sum of all cash flows received by the agent at time t and represents the *realized return*. For example if an agent purchases 1 stock at the price of 100 it will have a cash flow of -100 (and +100 if selling). $h_t \in \mathbb{Z}$ is the agents position size at time t. Further, we have the fee cost that the agent pays whenever it executes a trade. We set this fee to 0 because, in our model, it will only affect the RL agents desire to buy and sell, which we will control through the reward function. $\mathit{slippage} \in [0, 1]$ is a parameter deciding the cost of having to buy back (sell off) stock to create a neutral position, i.e. it is assumed that an agent would have to pay a premium compared to the current market price in order to neutralize its position. Thus the term in brackets represent the *unrealized return*. In this study we will consider a $\mathit{slippage} = 0.005$, resulting in the agents paying a 0.5% premium to get rid of their position. In reality the size of the slippage is relative to the position size, since it is harder to neutralize a big position than a small position.

13 Environment Calibration

13.1 Stylized Facts in High-Frequency Data

Like in part II, we will have the stylized facts in mind when performing simulation of the financial markets. In this part though, as opposed to part II, we model very high frequency data. The data is supposed to represent live data and include all trades in the market. Nevertheless we do still expect stylized facts, though the expectation to some of them changes a bit. We will again take basis in the 3 empirical facts: absence of autocorrelation, heavy tails and volatility clustering.

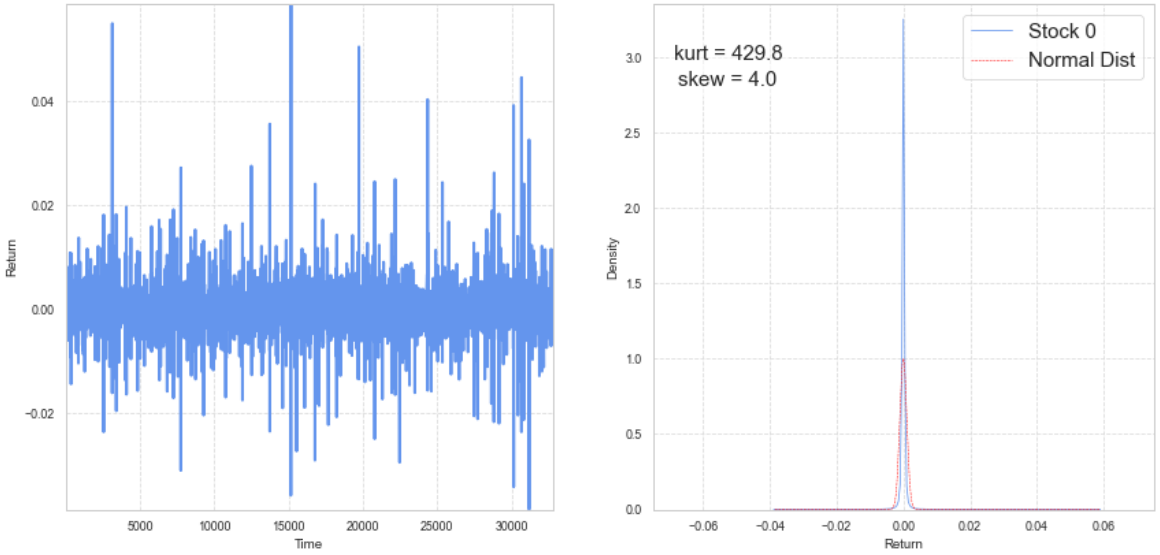
As mentioned the absence of autocorrelation only applies when the time intervals are larger than 20 minutes. This is indeed not the current case. In high-frequency data, returns often show negative autocorrelation at low lags. This is usually attributed to the bounces between bid and ask prices and the micro-structure of modern financial markets (Cont 2001). The micro-structure refers the way the markets operate, with order books, quotations, spreads etc.

As for heavy tails and the general distribution of the returns, it happens that the lower the frequency of data, the more the distribution diverges from the normal distribution. Thus we will expect our distribution to have even heavier tails than the distributions examined in part II.

Volatility clustering is still observed, and as mentioned a natural objective for market makers is to increase spreads as volatility increases, which has a recursive effect because larger spreads will make the bounces between ask and bid prices larger.

13.2 Data

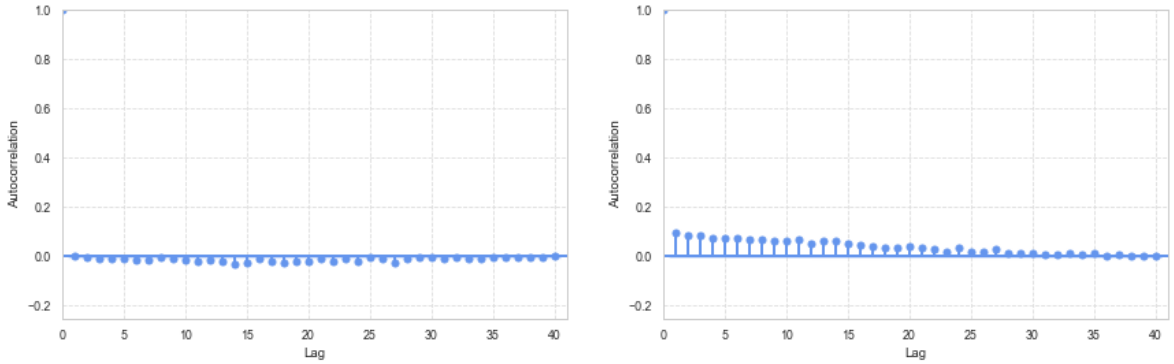
We would like our model to reflect real financial markets, therefore we need some real high-frequency data for comparison. We will use data provided in a recent Kaggle competition regarding realized volatility prediction, hosted by the market making firm Optiver (*Kaggle, Optiver realized volatility prediction* n.d.). The data consists of time period, price and volume traded at the given price of real stocks, which has been anonymized through *stock_ids*. We have selected to focus on *stock 0*. Figure 19 (a) and (b) shows the returns of 'stock 0' and the corresponding density. The returns illustrate that the volatility clustering is present and the density has a high kurtosis of 429.8 and is right-skewed with a skewness of 4. Figure 20 (a) and (b) shows autocorrelation of returns and squared returns respectively. The autocorrelation of the returns is approximately zero. This is a bit surprising, because literature (Cont 2001) implies an expected negative correlation on low lags. Even though the referenced article is old and the market facts could have changed by now, we have only found literature confirming this, e.g. (Vyetrenko et al. 2019). The autocorrelation of the squared returns are positive and decays slowly, as expected.



(a) *Stock 0 returns.*

(b) *Stock 0 density of returns.*

Figure 19: *Stock 0 returns and corresponding density compared to normal distribution . We see that the kurtosis is 429.8 and the skewness is 4.*



(a) *Stock 0 acf of returns.* (b) *Stock 0 acf of squared returns.*

Figure 20: *ACF of returns (left) and squared return (right) from the observed returns in stock 0. We see little to no autocorrelation of returns, though we expected a negative first lag according to literature. And we see slow decay of autocorrelation in squared returns as expected.*

13.3 Calibration

To calibrate our model we need an objective function to optimize over in order to estimate the parameters. We let $\boldsymbol{\theta}^{\text{ABM}}$ denote the parameter vector of all parameters included in the agent based model, i.e. all parameters for individual agents as well as $N^{\text{RAgents}}, N^{\text{IAgents}}, N^{\text{TAgents}}, N^{\text{MAgents}}$, which respectively represent the number of random, investor, trend and market making agents in the environment. Furthermore we let $\mathcal{L}(\boldsymbol{\theta}^{\text{ABM}})$ denote the desired objective function to optimize. A natural objective in our setting would be to use a loss function reflecting the difference in descriptive statistics of the target distribution and the ABM distribution. We therefore suggest a loss function of the structure

$$\ell(\boldsymbol{\theta}^{\text{ABM}}) = \Big(\hat{\mathbf{y}}(\boldsymbol{\theta}^{\text{ABM}}) - \mathbf{y}\Big)^{\top} \mathbf{W} \Big(\hat{\mathbf{y}}(\boldsymbol{\theta}^{\text{ABM}}) - \mathbf{y}\Big), \quad \sum_i W_{ii} = 1$$

where $\mathbf{y}$ is a vector with all our statistical targets, and $\hat{\mathbf{y}}(\boldsymbol{\theta}^{\text{ABM}})$ is the corresponding estimates from the ABM. $\mathbf{W}$ is a diagonal weight matrix with $i'th$ diagonal element representing the weight of the $i'th$ statistic. Appropriate statistics in our scenario would be the mentioned stylized facts about kurtosis, autocorrelation of returns and autocorrelation of squared returns. For simplicity we imagine that we only care about one lag for both autocorrelations: lag $= \tau$. Such that

$$\hat{\mathbf{y}}(\boldsymbol{\theta}^{\text{ABM}}) = \begin{bmatrix} \hat{\text{Kurt}}(\boldsymbol{\theta}^{\text{ABM}}) \\ \hat{\rho}(\tau, \boldsymbol{\theta}^{\text{ABM}}) \\ \hat{\rho}^{\text{abs}}(\tau, \boldsymbol{\theta}^{\text{ABM}}) \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} \text{Kurt} \\ \rho(\tau) \\ \rho^{\text{abs}}(\tau) \end{bmatrix}, \qquad \mathbf{W} = \begin{bmatrix} w_1 & 0 & 0 \\ 0 & w_2 & 0 \\ 0 & 0 & w_3 \end{bmatrix},$$

where Kurt, $\rho(\tau)$ and $\rho^{\text{abs}}(\tau)$ represents the target kurtosis, autocorrelation of returns with lag τ and autocorrelation of squared returns at lag τ. $\hat{\text{Kurt}}(\boldsymbol{\theta}^{\text{ABM}})$, $\hat{\rho}(\tau, \boldsymbol{\theta}^{\text{ABM}})$ and $\hat{\rho}^{\text{abs}}(\tau, \boldsymbol{\theta}^{\text{ABM}})$ represents their corresponding estimates from the ABM. The loss is easily extended to include more lags or other statistics.

We minimize the loss function by grid searching through a specified range of a selected subset of parameters, while holding the rest fixed. We select a subset because in practice our ABM simulation is computationally heavy and we select the parameters which we think has the largest influence on the loss function. An appealing feature of using grid search is that we get a natural method to standardize each (target, estimate) pair in the loss function, with respect to the sampled targets. Taking example in kurtosis that is

$$\hat{\text{Kurt}}(\boldsymbol{\theta}^{\text{ABM}}{}_k)^{\text{std}} = \frac{\hat{\text{Kurt}}(\boldsymbol{\theta}^{\text{ABM}}{}_k) - \mu_{\hat{\text{Kurt}}}}{\sigma_{\hat{\text{Kurt}}}}, \qquad \text{Kurt}^{\text{std}} = \frac{\text{Kurt} - \mu_{\hat{\text{Kurt}}}}{\sigma_{\hat{\text{Kurt}}}}$$

where $\boldsymbol{\theta}^{\text{ABM}}{}_k$ is the k'th parameter vector and $\mu_{\hat{\text{Kurt}}}$ and $\sigma_{\hat{\text{Kurt}}}$ are the sample mean and standard deviation of the estimated kurtosis. Thus after the full grid search, we estimate the parameter combinations by plugging the standardized (target, estimate) pairs into $\hat{\mathbf{y}}(\boldsymbol{\theta}^{\text{ABM}})$. In order to avoid too much randomness, for instance because an investor agent has entered the market multiple times in a row, which has very low probability, we run through a batch of 5 episodes for each parameter combination $\boldsymbol{\theta}^{\text{ABM}}{}_k$. We then let the estimated values be the mean over these 5 episodes.

We choose to calibrate against kurtosis and the first five lags of each autocorrelation function. We weight it such that the three terms gets weighted equally and that each lag also gets weighted equally within the autocorrelation term, providing the loss

$$\begin{aligned}\ell(\boldsymbol{\theta}^{\text{ABM}}{}_k) = \frac{1}{3}\Big(\hat{\text{Kurt}}(\boldsymbol{\theta}^{\text{ABM}}{}_k)^{\text{std}} - \text{Kurt}^{\text{std}}\Big)^2 + \frac{1}{15}\sum_{\tau=1}^{5}\Big(\hat{\rho}(\tau, \boldsymbol{\theta}^{\text{ABM}})^{\text{std}} - \rho(\tau)^{\text{std}}\Big)^2 \\ + \frac{1}{15}\sum_{\tau=1}^{5}\Big(\hat{\rho}^{\text{abs}}(\tau, \boldsymbol{\theta}^{\text{ABM}})^{\text{std}} - \rho^{\text{abs}}(\tau)^{\text{std}}\Big)^2.\end{aligned}$$

We strongly believe that the composition of the different agent types will have a large influence on the environment. Therefore we have selected $N^{\text{RAgents}}, N^{\text{TAgents}}$ and N^{MAgents} to be part of the grid. We have not included N^{IAgents} because we wish to keep the number of investors in the market low, so we fix $N^{\text{IAgents}} = 2$, where one of the agents are allowed to short sell. We will though add the investor related parameters $\lambda^{(\text{I})}, v_s^{(\text{I})}$ and $v_b^{(\text{I})}$. They control the intensity and aggressiveness of the investors and thus, somewhat compensate for the not included number of investors parameter. Furthermore we will add $n^{(\text{R})}$ and $c^{(\text{M})}$, which we have found to be two important parameters for the random agent and the market maker agent. These two parameters control the amount of volume that the random agents put in the market and the base spread of the market maker. They both have a large impact on the number of total trades happening in the market, and thus strongly defines the amount of movement in the market.

Calibration Results

After the grid search we observed that the losses ranged from 0.5, to 8.5 - which means that the parameters we have selected does indeed have an influence on the resulting statistics. The final loss minimizing parameter vector $\boldsymbol{\theta}^{\text{ABM}*}$ is shown in Table 5. The table includes both the parameters estimated using grid search and the ones fixed in advance.

We see that especially a high number of market makers were needed, which might be due to the fact that they are the main liquidity providers. Moreover we have made it such that both the trend agents and the market maker agents have some randomness such that they don't trade exactly the same way. This is done by drawing the $ma_1^{(T)}, ma_2^{(T)}$ and $\gamma_2^{(M)}$ parameters randomly from a uniform distribution for each of the N^{TAgents} and N^{MAgents} trend and market maker agents.

Table 5: *Parameter vector $\boldsymbol{\theta}^{ABM*}$ used for simulating the ABM estimated through grid search. Note that the parameters that are within brackets are upper and lower bounds, where they are drawn uniformly within those bounds; either drawn as integers or doubles.*

Random		Investor		Trend		Market Maker	
N^{RAgents}	4	N^{IAgents}	2	N^{TAgents}	4	N^{MAgents}	15
$lb^{(\text{R})}$	0.0001	$\lambda^{(\text{I})}$	0.01	$ma_1^{(\text{T})}$	$[10, 15]$	$n_\sigma^{(\text{M})}$	3
$ub^{(\text{R})}$	0.003	$m_s^{(\text{I})}$	0.015	$ma_2^{(\text{T})}$	$[20, 30]$	$\gamma_1^{(\text{M})}$	0.00005
$\sigma^{(\text{R})}$	0.0025	$m_b^{(\text{I})}$	0.015	$th^{(\text{T})}$	5	$\gamma_2^{(\text{M})}$	$[0.5, 0.75]$
$n^{(\text{R})}$	3	$n_{bo}^{(\text{I})}$	8	$m^{(\text{T})}$	0.005	$c^{(\text{M})}$	0.1
$p_s^{(\text{R})}$	0.5	$n_{so}^{(\text{I})}$	4			$v^{(\text{M})}$	3
$p_b^{(\text{R})}$	0.5	$v_b^{(\text{I})}$	20				
		$v_s^{(\text{I})}$	40				

The distribution of the returns for the $\boldsymbol{\theta}^{\text{ABM}^*}$ compared to the real distribution from figure 19 (b), is shown in figure 21. It shows that the distribution of the returns simulated from the ABM is very narrow and centered, all though the distribution we saw for stock 0 is a bit less narrow and has heavier tails, we still consider these results comparable. The kurtosis in our ABM is 235.5, which is quite high, but still a bit far from the kurtosis of 429.8 that we saw in stock 0. Figure 22 shows the acf of returns (a) and squared returns(b) from the calibrated market. The result is not great. The returns have negative autocorrelation for lag 1,3,4 and it suddenly increases again around the 15-17 lag. The first lag has a negative acf of approx -0.2, opposite stock 0, which had 0. This reflect literature, but not the data that we calibrated our model against. The acf of squared returns seems reasonable for the first lags. It starts high at about 0.4, and decays slowly, a bit too slowly. What is quite strange though is that it starts to increase again around the 10th lag. We would have liked to see a slowly decaying function like the one of stock 0. One thing to keep in mind is that we calibrated the model only weighting the importance of the first 5 lags. Thus the bizarre patterns that we observe on the higher lags, could maybe be bettered by including them in the loss function.

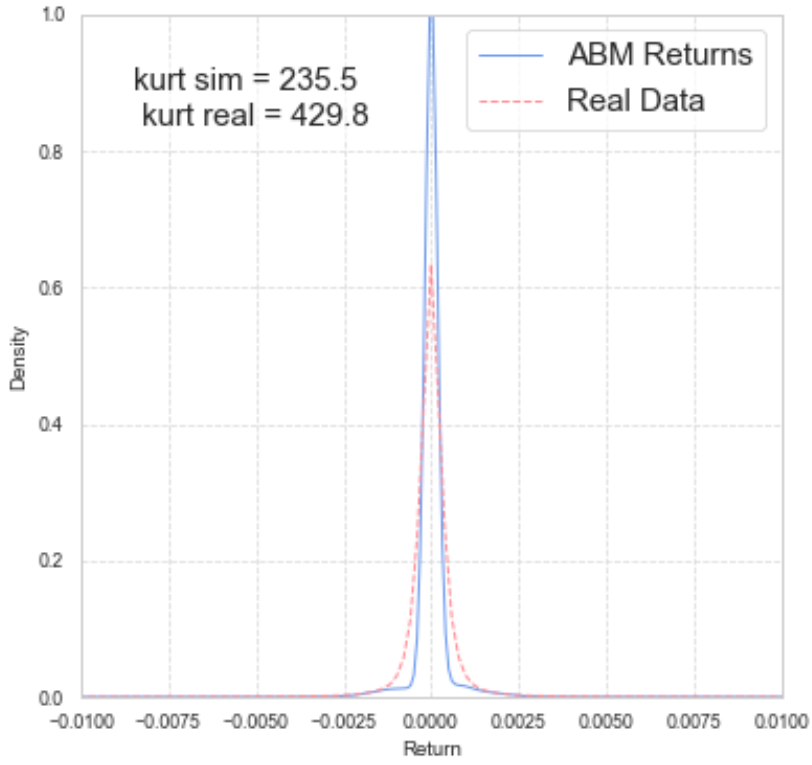
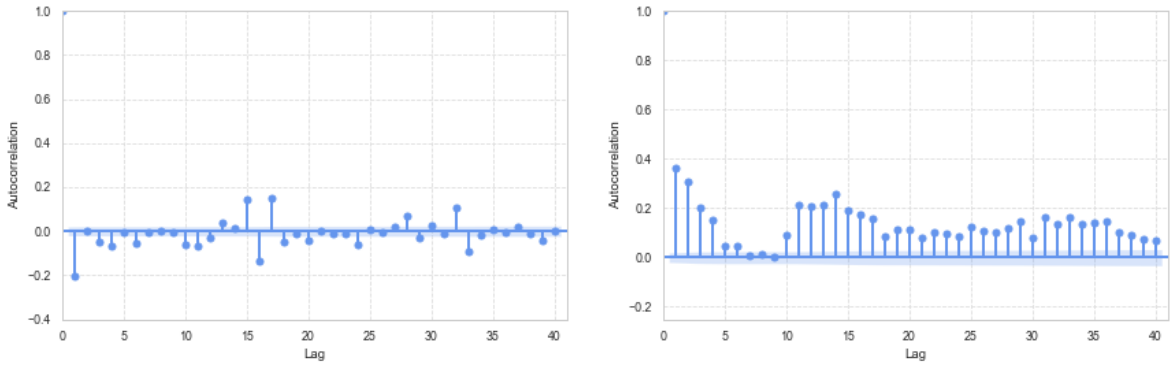


Figure 21: *Density comparison of abm simulated returns and real returns from stock 0.*



(a) *Acf of returns.* (b) *Acf of squared returns.*

Figure 22: *Autocorrelation of returns (right) and squared returns (left), from the ABM simulation using* $\boldsymbol{\theta}^{ABM}$.

14 A2C Market Maker Agent

As for the market maker RL agent we have decided to use the A2C method (algorithm 7), which utilizes the Actor-Critic framework with advantage as described in Part I.

The idea is to improve the market making agent by applying RL to learn optimal actions. The RL agent, like any of the other agents, place the action

$$\mathrm{a}_t^{(\mathrm{RL})} = \begin{bmatrix} \mathrm{bp}_t^{(\mathrm{RL})} & \mathrm{sp}_t^{(\mathrm{RL})} & \mathrm{bv}_t^{(\mathrm{RL})} & \mathrm{sv}_t^{(\mathrm{RL})} \end{bmatrix},$$

at time t. To start off, we let the actor choose the entire action space, i.e. buy and sell prices and volumes. At the same time it was rewarded by change in PnL, which we quickly changed to a risk-sensitive approach that we will discuss later. We found that setting volumes gave the agent too much power. Without a limitation to the volumes, the agent could easily move the market in the desired direction and thus quickly learned that it could take advantage of this since the slippage was a constant and did not scale with position size. Then the agent could just keep buying at extremely high prices and keep doing so, because the more it bought the more it moved the price in the same direction. In a way it was a positive sign that the agent found this strategy. But it obviously reflected a mistake in the model assumptions and especially emphasized the importance of setting strict model assumptions when using RL algorithms and ABMs. An illustration of this initial observations is shown in appendix A.1.

As a solution to the volume problem, we changed it such that the RL agent has the same fixed volume as the MM agents, both for buy and sell volume (a value of 3 after market calibration):

$$\mathrm{bv}_t^{(\mathrm{RL})} = \mathrm{sv}_t^{(\mathrm{RL})} = v^{(\mathrm{M})}.$$

Likewise we set the latency of the RL agent equal to that of the MM agents:

$$\delta_t^{(\mathrm{RL})} = \delta_t^{(\mathrm{M})} = U(0,1)$$

As for the prices we model these with the A2C algorithm. The agent does not set specific prices, but prices relative to the last observed market prices through margins. The prices are set as

$$\begin{aligned} \mathrm{bp}_t^{(\mathrm{RL})} &= P_{t-1}(1 + mb_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t)) \\ \mathrm{sp}_t^{(\mathrm{RL})} &= P_{t-1}(1 + ms_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t)) \end{aligned}$$

where $mb_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t) \in \mathbb{R}$ and $ms_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t) \in \mathbb{R}$ are respectively buy and sell margins. Note that they are both added to 1; this is because they both can take positive and negative values. Under normal circumstances we expect $mb_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t)$ to be negative and $ms_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t)$ to be positive, i.e. buy low and sell high. $\mathbf{x}_t$ is the feature vector extracted from the information provided by the state s_t, and is thus dependent on s, but we leave that out to spare notation. The features consists of the standard key information from the state

plus an estimated local volatility calculated similarly to that of the Market Maker agent:

$$\sigma_t^{(\mathrm{RL})} = \sqrt{\frac{\sum_{i=t-n_\sigma^{(\mathrm{RL})}}^{t-1} \left(\mathrm{P}_i - \bar{\mathrm{P}}\right)^2}{n_\sigma^{(\mathrm{RL})} - 1}}, \qquad \bar{\mathrm{P}} = \frac{1}{n_\sigma^{(\mathrm{RL})}} \sum_{i=t-n_\sigma^{(\mathrm{RL})}}^{t-1} \mathrm{P}_i.$$

as well as returns from the last $k^{(\mathrm{RL})}$ time steps. Recall that $n_\sigma^{(\mathrm{RL})}$ denotes the number of returns used to calculate the local volatility. Thus at time t the feature vector is

$$\mathbf{x}_t = \{\mathrm{RE}_{t-k},\ \mathrm{RE}_{t-k+1}, ...,\ \mathrm{RE}_{t-1},\ \mathrm{TMV}_{t-1},\ \mathrm{MBP}_{t-1},\ \mathrm{MSP}_{t-1},\ \mathrm{TBV}_{t-1},\ \mathrm{TSV}_{t-1},\ \sigma_t^{(\mathrm{RL})}\}.$$

Because the actions are continuous, we model them in the continuous setting estimating means in the multivariate normal distribution

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{1}{\sqrt{(2\pi)^2|\Sigma|}} \exp\Big(-\frac{1}{2}(a - \mu(\mathbf{x}_t, \boldsymbol{\theta}))^\top \Sigma^{-1}(a - \mu(\mathbf{x}_t, \boldsymbol{\theta})\Big),$$

where $\mu(\mathbf{x}_t, \boldsymbol{\theta}) \in \mathbb{R}^2$ is estimated by our actor through a function approximator and $\Sigma \in \mathbb{R}^{2\times 2}$ is a diagonal matrix, with predefined fixed variances for both buy and sell price margin. a denotes the sampled action (buy and sell margin):

$$\Big[mb_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t) \quad ms_{\boldsymbol{\theta}}^{(\mathrm{RL})}(\mathbf{x}_t)\Big] = a \sim N\Big(\mu(\mathbf{x}_t\, \boldsymbol{\theta}), \Sigma\Big)$$

Recall that the Actor-Critic method is a policy gradient method and that the actor uses the advantage function as an estimate for the action-value function in the gradient step. Furthermore the advantage function is approximated by the TD error. Thus the actor loss function is given by

$$\begin{aligned}
\mathcal{L}_{\text{actor}} &= -\frac{1}{2n_{\mathcal{B}}} \sum_{i\in\mathcal{B}} \sum_{j=1}^{2} \Big(\log \pi_{\boldsymbol{\theta}}(a_{i,j}|\mathbf{x}_i) \hat{Adv}_{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_i, a_{i,j}) \Big) \\
&= -\frac{1}{2n_{\mathcal{B}}} \sum_{i\in\mathcal{B}} \sum_{j=1}^{2} \Big(\log \pi_{\boldsymbol{\theta}}(a_{i,j}|\mathbf{x}_i) [R_{i+1} + \gamma \hat{v}_{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_{i+1}, \mathbf{w}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_i, \mathbf{w}] \Big)
\end{aligned}$$

where $n_{\mathcal{B}}$ is the number of observations in our batch $\mathcal{B}$. $a_{i,j}$ represents the estimated action given features $\mathbf{x}_i$, where j represents the two actions that the agent needs to take; buy and sell price margins. Note the minus in front of the function. That is because the term without represents probability times corresponding excess return, which is something we would like to maximize. We therefore add the minus when the problem is formulated as minimization problem of a loss function, in contrast to maximization of a performance measure. When drawing the samples from our batch we draw from all previous time steps and thus uses the experience replay as explained in Part I.

Recall that the critic estimates the advantage function, which in our case is approximated by the TD-error. Thus the critic adapts the estimated state-value through the following loss:

$$\mathcal{L}_{\text{critic}} = \frac{1}{n_{\mathcal{B}}} \sum_{i\in\mathcal{B}} \left(R_{i+1} + \gamma \hat{v}_{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_{i+1}, \mathbf{w}) - \hat{v}_{\pi_{\boldsymbol{\theta}}}(\mathbf{x}_i, \mathbf{w})\right)^2.$$

As for the reward R_t, we still care about risk. Generally market makers want to profit from the spread and hedge all other risk as much as possible. However since we only have one tradeable asset, it is not possible to hedge a position in this market. A way to manage risk is by keeping a position as close to market neutral as possible. Thus we have included a penalty on the agents position size in the reward function, inspired by the method of (Ganesh et al. 2020). The reward that the agent receives at time t is:

$$R_t = \mathrm{PnL}_t - \mathrm{PnL}_{t-1} - (h_t \cdot \lambda^{(RL)})^2,$$

where $\lambda^{(RL)}$ is a risk-aversion parameter that we can change accordingly to the amount of penalty we want the agent to suffer form carrying risk by holding a position in the market.

As for function approximation we use feed-forward neural networks for both the actor and the critic. To decide the number of hidden layers and hidden units in each layer we approached the problem in a way similar to Part II.

The final neural networks consist of two hidden layers, each with 256 units and ReLU activation functions between the hidden layers. The critics output-layer uses a linear activation function with one dimensional output since we simply use this to approximate the state-value function, which takes values in $\mathbb{R}$. The actor outputs the mean vector $\mu(\mathbf{x}_t, \boldsymbol{\theta})$ and uses a output-layer with a modified tanh activation function. Tanh returns values in $[-1, 1]$, but because we expect the agent to return prices close to the market price in order to compete and get orders filled we adjust it to

$$\sigma_{\text{actor}} = \frac{1}{40} \frac{e^x - e^x}{e^x + e^{-x}},$$

such that the outputted values are in the interval $[-0.05, 0.05]$. This way we help the actor to set prices so that it is hopefully able to learn quicker. If we used a normal tanh activation function the actor would have to figure out itself that the remaining interval $[-1 : -0.05,\ 0.05 : 1]$ would either result in a trade not happening with almost certainty or that the agent is selling (buying) way below (above) the market price, which seems like a bad idea.

Notice that when the model is up and running it will on an on going basis observe the market, act and place orders, learn from these actions (hopefully) and adjust its trading strategy accordingly. This can be seen as a fully automated trading strategy without any needed input from the modeller through the trading process.

A visualization of the A2C Market Maker agent is shown in figure 23. It illustrates the relationship between the environment, the actor and the critic.

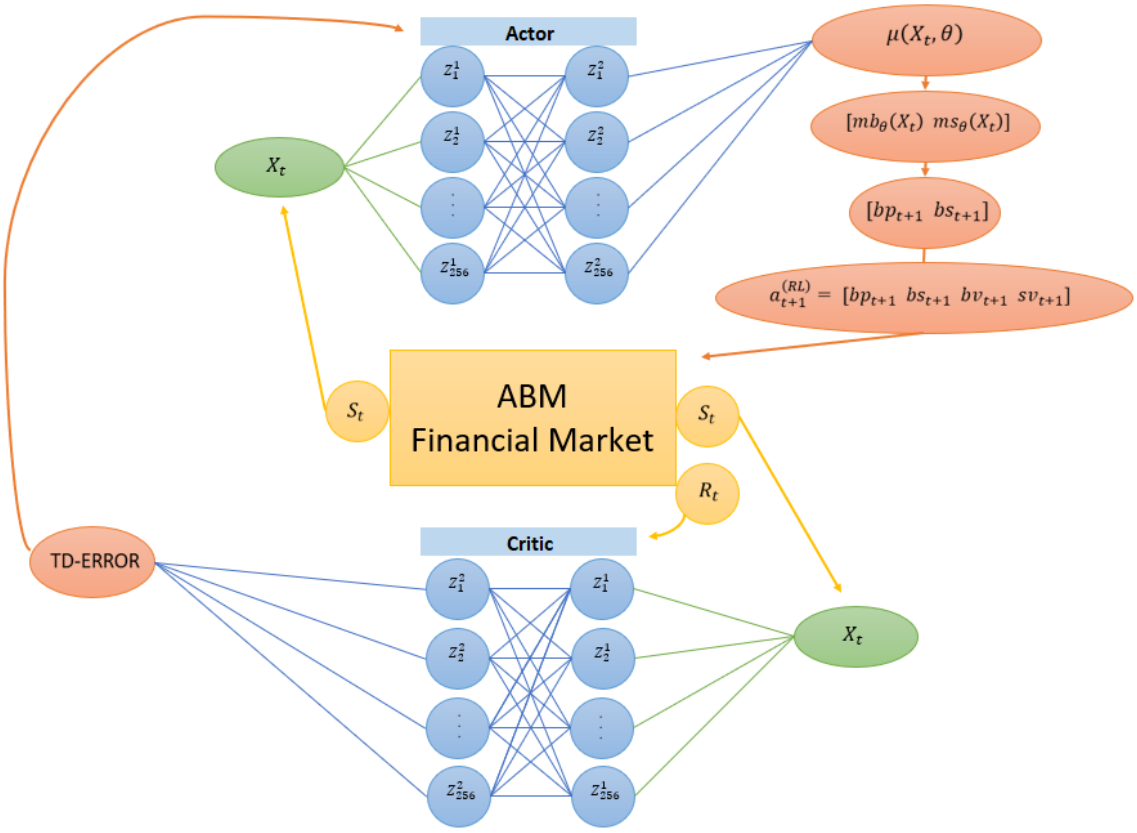


Figure 23: *Illustration of the A2C market maker agent. The yellow box represents the financial market and the order book, which sends out stat information to both the actor and the critic (blue boxes), which extracts their features (green boxes). The critic estimates a TD-Error (red box), which it sends to the actor, in order for the actor to perform a gradient step. The actor then calculates a mean for the normal distribution, which spits out the margin used to calculate buy and sell prices. The prices together with the volumes are then send back into the order book, and so on it continues.*

15 Results Part III

Running this experiment has been computationally heavy. One episode of 500 time steps takes approximately 60 seconds through Google Clouds c2-standard-8 engine. The C2 machine family is compute-optimized and is their highest performing engine for single-core tasks. We did not implement the code so that it could be run in parallel, and thus using this engine was our best option. Generally when applying RL, a substantial amount of training is required in order to evaluate whether the agent is learning or not. This is because the agent needs to explore many states, before it is able to approximate the value functions. Furthermore our RL agent and our ABM has a lot of hyper parameters. These factors combined with the slow simulation has caused the tuning of the agent and the environment very time consuming. Therefore we have not been able to calibrate our ABM of the environment nor train the actor-critic agent, as much as we would have liked. Considering that the REINFORCE agent in Part II used approximately 10.000 episodes with 525 time steps before it converged to a strategy, we might need hundreds of thousands of episodes for this experiment considering the extreme stochastic nature of the environment alongside the complexity of this agent. Since we weren't able to train that many episodes, we tried increasing the agent's learning rate, but this resulted in the agent falling into local minimums where either the estimated mean for the buy or sell price seems to be stuck. To try and mitigate this, we have tested different kinds of regularization methods as presented in Part I but with no significant improvement. Thus, our final agent is not ideal, but the results show improvement with respect to reward and are still helpful for further development of the environment and the agent.

In regards to hyper parameter tuning the agent have shown to be very sensitive to the penalty that we put on the position size. But setting the $\lambda^{(RL)}$ hyperparameter have shown to be very hard, since the ratio between the contribution to the loss from the position penalty and the contribution from the spread, is in reality influenced by a lot of other factors:

• The reward from the spread alone is not very high compared to the reward (positive or negative) from holding a position, i.e. having a large position when the market moves has a huge impact on return.
• The penalty from holding a position can be adjusted by $\lambda^{(RL)}$, but the total reward is also highly influenced by the market movements.

Thus adjusting the position penalty is a hard balance since we want it to actually make trades while we also want it to keep a low position all the time. But the reward it gains from the spread is smaller, and thus it is more influenced by the market movements and the position penalty, than the actual gain reward from good spreads. At some point the agent even put its buy prices above its sell prices since this resulted in a position size of approximately 0, which was better than generating a large negative reward from the position penalty. To fix this we reduced the position penalty, and finally ended up with training the agent for 24 hours with the hyper parameters seen in Table 6.

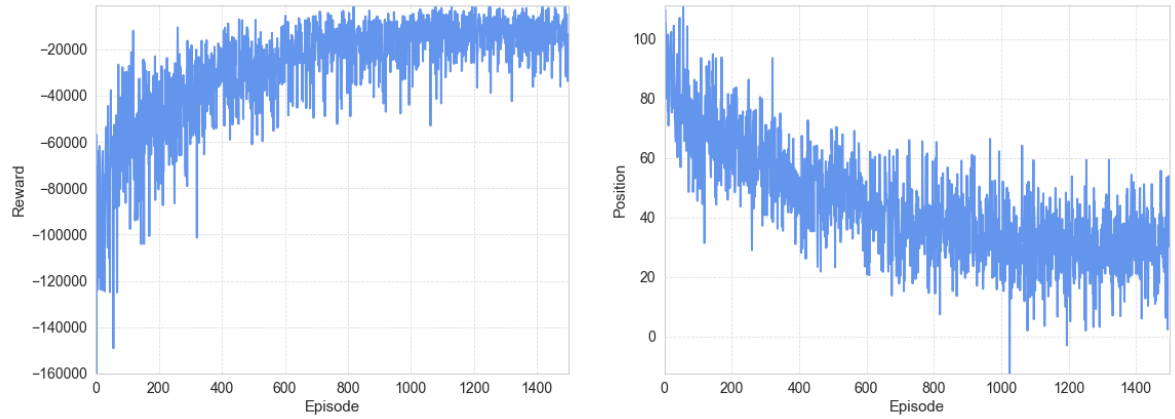
Table 6: *Most essential hyper parameter values for the A2C agent*

Parameter	Value	Explanation
$\kappa^{(\mathrm{RL})}$	$1e-03$	Variance term multiplier
$\alpha^{(\mathrm{RL})}_{\mathrm{actor}}$	$1e-07$	Learning rate of the Actor NN
$\alpha^{(\mathrm{RL})}_{\mathrm{critic}}$	$1e-07$	Learning rate of the Critic NN
$bv^{(\mathrm{RL})}$	3	Buy volume
$sv^{(\mathrm{RL})}$	3	Sell volume
$\epsilon^{(\mathrm{RL})}$	0.05	Exploration
$k^{(\mathrm{RL})}$	5	time steps before t to include returns from
$\lambda^{(\mathrm{RL})}$	0.1	Position Penalty
$\gamma^{(\mathrm{RL})}$	0.95	Discount factor
$\sigma^{(\mathrm{RL})}$	0.0025	Standard deviation in action space (Normal)
$\sigma^{(\mathrm{RL})}_{\mathrm{priceroof}}$	0.025	Max percentage difference from observed price
$n^{(\mathrm{RL})}_{\mathcal{B}}$	5	Number of samples used in each gradient step
$\lambda^{(RL)}_{\mathrm{L2}}$	$1e-05$	L2 regularization weight
$n^{(\mathrm{RL})}_{\sigma}$	10	n observations to calculate volatility
p_{dropout}	0.3	Dropout probability for each hidden layer

In figure 24(a) we see that the agent increases it's reward quite steadily through time, especially in the beginning, which is a very good sign because it indicates learning. Furthermore we see in figure 24(b) that it's average position size through time decreases. This relationship is expected since the position is penalized in the reward function. It could indicate that the agent simply stopped trading, e.g. by placing prices too far from the mid-price in order to avoid trades and keep a position of zero. Though this is indeed not the case. From figure 25 we see a 10 time step rolling average volume contribution as an average over the first 50 episodes (a) and last 50 episodes (b). We clearly see that the change in traded volume does not change a lot, if at all.

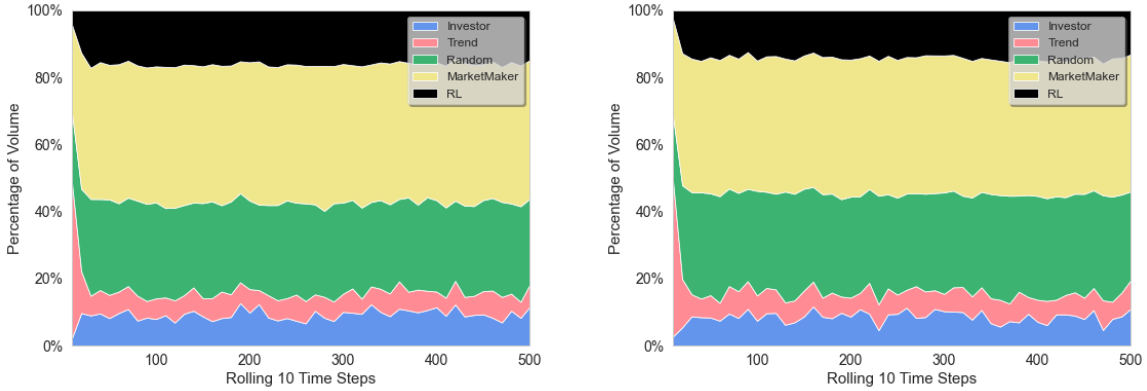
At first sight we see that the reward is negative (fig 24(a)), but this does not imply that the actual profit that the agent achieves is negative. Remember that the reward is a function that includes a penalty on the position size. However the figure might insinuate that it has converged in the far end, but there is still a lot of variation between the rewards, which lets us believe that the agent still tries out different actions.

Figure 26 show the actor (a) and critic (b) loss functions for each gradient step. When looking at loss functions in a supervised learning setting we assume that our observations is sampled i.i.d, but this is not the case in the RL setting. Since the agents actions affect both the rewards and the states we observe, while the rewards also affect how the agent acts through the loss functions. Moreover the loss function is highly affected by what the agent observes. If the agent observes something new, e.g. the first time an investor enters



(a) *A2C agents reward at end of episode.* (b) *A2C agents position size at end of episode.*

Figure 24: *The left figure shows the total reward that the A2C agent received at the end of each episode during training. The right figure shows the A2C agents position size at the end of the episodes during training.*

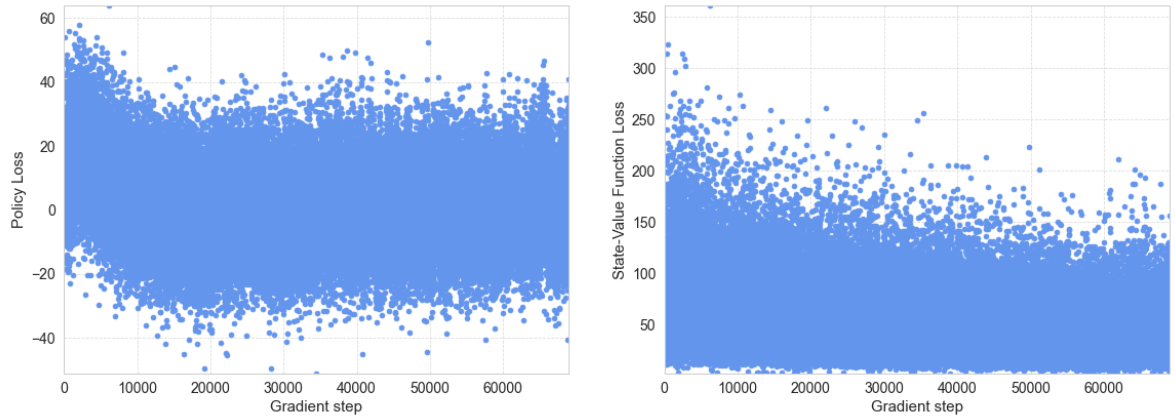


(a) *Average volume contribution first 50 episodes.* (b) *Average volume contribution last 50 episodes.*

Figure 25: *Average 10 time step rolling volume contribution over the first 50 episodes (left figure) and last 50 episodes (right figure). We see that there is not a lot of change in the distribution of traded volume between the start and end of the training period.*

the market. Then it will move the price rapidly, and if the agent has not observed this before its loss function could potentially spike. We, therefore, expect more randomness in both loss functions than we would in a standard supervised learning setting. And the shape of the loss function is not necessarily an indication of increased performance or convergence. Here it is more telling to look at the reward function.
Both the function approximators and the reward still have a lot of stochasticity, and it is hard for the agent to reduce this.

Lastly we take a look at how the RL agent performed with respect to risk and return in comparison to the other agents with a focus on the market maker. These results can be seen in Figure 27 (a-f). In (a) we see average end of episode return for each agent class.

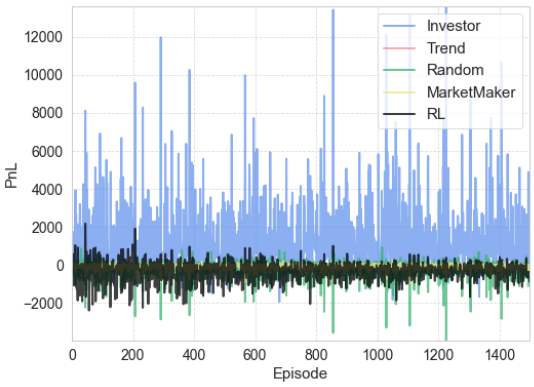


(a) *Actors loss during training.*

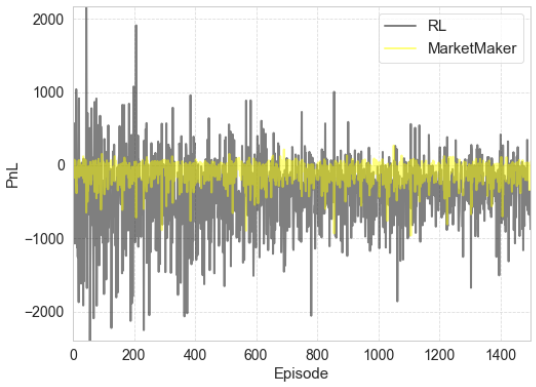
(b) *Critics loss during training.*

Figure 26: *The left figure shows the Actors loss function value at each gradient step. The right figure shows the Critics loss function value at each gradient step.*

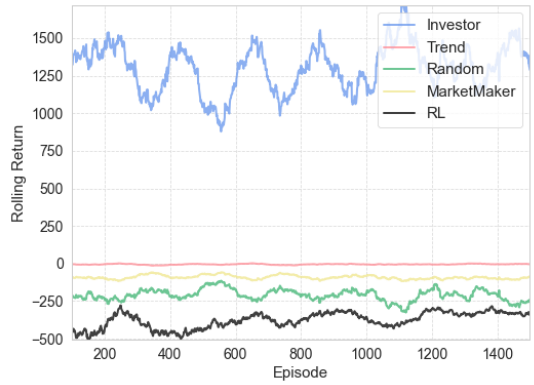
It is clear that the winner is the investor and the losers are the random agent and the RL agent. In (c) and (e) we respectively see the rolling mean and standard deviation over 100 episodes of the final PnL. We see that even though the investor clearly has the highest PnL it comes with a cost of high variance. The RL agent performs worst with respect to PnL through the entire period, but as we see in (d) it appears to improve over time. Subfigure b, d and f shows the same pictures as above, but only for the RL agent and the market maker. In (d) we see that the market maker has a quite steady PnL above the RL agent, but we also see that the variation of the PnL for RL agent is large and seems to increase through time. On the other hand we see that the standard deviation of the RL agents PnL decreases through time, but is higher than that of the market maker for the full period. The observations that the RL agent increases PnL and decreases the standard deviation of PnL through time are both good signs and indicate an overall increase in performance both concerning risk and return.



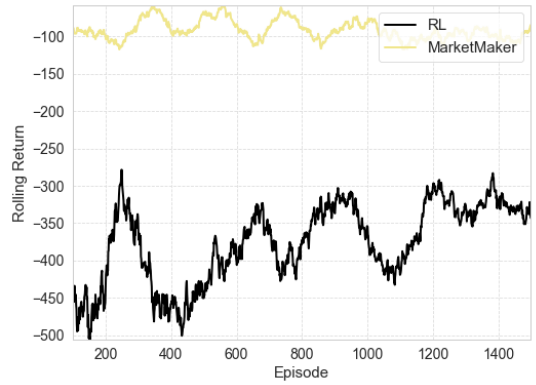
(a) *PnL all agents.*



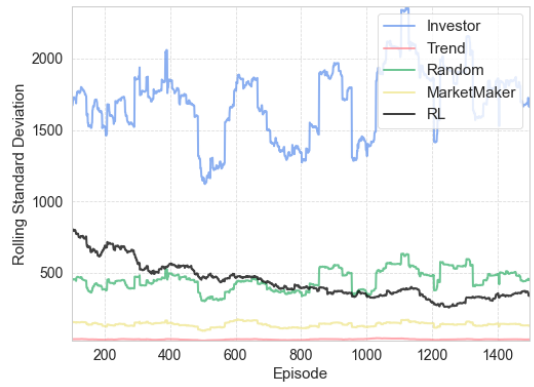
(b) *PnL RL vs MM.*



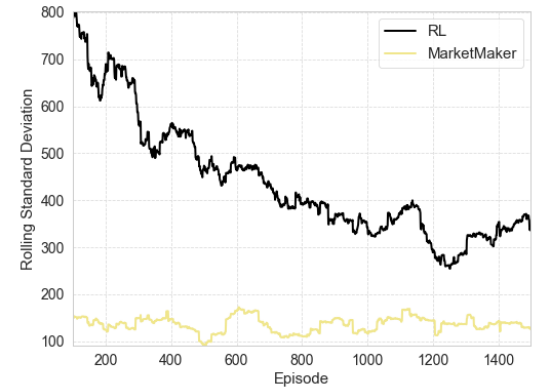
(c) *Rolling mean PnL all agents.*



(d) *Rolling mean PnL RL vs MM.*



(e) *Rolling std of PnL all agents*



(f) *Rolling std of PnL RL vs MM*

Figure 27: *Performance comparison during the 1500 simulated episodes. We see respectively see PnL, 100 period rolling mean PnL and 100 period rolling standard deviation of PnL for all agents on the left side (a), (c) and (e). On the right hand side we see the same, but only for the RL agent and the market maker agent.*

16 Discussion Part III

As we have shown, an important aspect to keep in mind when using RL is that the model assumptions are crucial. Changing how much volume the agent is allowed to buy and sell can change everything. The RL agent does not know the rules of the games, and thus if there is a flaw in the game with a possibility for exploitation - the agent will likely use it. Furthermore, these aspects underline the importance of being careful and watching out for flaws when using RL. Additionally, it is often very difficult to interpret why the actions of the agent are made due to the complexity of the neural networks often used in the process. And in case of new regulations or market features, it will be impossible to know how the RL agent will react to these. Thus if an asset manager attempts to implement RL to optimize strategies, she has to keep in mind that it will be difficult to explain the reasons behind the executed trades to her superiors.

Even though we see some similarities between the stylized facts of the real market (stock 0), and the pseudo market calibrated using stock 0, there is still a lot of room for improvement. Applying either a wider grid search or more advanced optimization methods would be interesting. It would definitely benefit to invest time implementing code in a high performance computing language like c++ with a parallelized implementation. We have only focused on the three stylized facts about heavy tails, autocorrelation in returns, and volatility clustering to avoid making things too complicated. It could also be interesting to investigate other stylized facts, such as the order book dynamics, the volume traded, bid-ask spreads, etc.

Concerning the RL part of the experiment, we have only implemented RL to optimize market making. It could also be interesting to test implementations with other reward functions reflecting the other market participants. Furthermore, the market seems to be in huge favor of the large long term investor and appears to be biased towards their goal. This is probably a result of the market calibration, where this large power of the investor happens to reflect the statistics of the stock 0 market best. Here it could be interesting to dive deeper into the different agents attributes and find a way to reflect the market better while giving the investor less power.

Since the pseudo market only included one asset, it was not possible for the agent to hedge its position, therefore we needed to punish its position size heavily in order to nudge the agent towards gaining profits from spreads rather than random market movements. In the real world there exists multiple markets with multiple assets, and it is possible to hedge positions. This could also be an interesting topic for further research.

During the experiment we calibrated our ABM simulation in order to reflect a real market, and then added our RL agent to find trading strategies. We have not included any tests of how the addition of the RL agents has affected the stylized facts of the market. One could even configure the reward function of the RL agent in order to match the stylized facts of the market. Thus the RL agent would be optimized to trade in order for the ABM to reflect the real market. This is indeed an interesting study and is currently being investigated by researchers, see e.g. (Ardon et al. 2021).

17 Conclusion Part III

In this part we have shown how ABMs can be used to simulate the dynamics of high-frequency financial time series. We have shown that through ABMs it is possible to replicate some of the stylized facts about the market. We observed that the composition of agents and their attributes plays a large role in the replication of these stylized facts. The heavy tails and volatility clustering seems easier modelled than the autocorrelation of returns, while we acknowledge that there is much room for improvement on all three.

Furthermore we have shown how RL can be applied in an ABM environment in order to find systematic market making strategies without any human decision making required in the trading process. With the goal of achieving a high return, while keeping a low inventory to mitigate risk, we see that the agent learns through time and increases its performance. We see that even though the agent learns it has still not achieved a better performance than the predefined non-learning market maker agent, but we do see clear indicators that it becomes better through time. Therefore, we will not reject that it is possible for the agent to learn a better strategy than the predefined agent, if it had more time. Additionally, we observed that the learning process was slow and likewise that the run time simulating the environment was slow. Because of this we have not been able to train the RL agent for as long time as desired.

Part IV
Conclusion

18 Accomplishments

We have presented modern RL methods to solve financial problems within portfolio optimization (Part II) and market making (Part III). In part II we let a risk-averse REINFORCE agent train on simulated prices of the S&P 500 stock index using an AR(1) + GARCH(1,1) model. Through time the agent learns and increases its return. The longer it trains the more it converges towards a buy and hold strategy, probably realizing the positive drift, which accumulates over time. This is not very surprising because the simulated model has random increments with almost no serial correlation. It agrees with classical finance theory, that the markets are unpredictable given only prices and returns. To investigate if the model could learn an optimal strategy that required a portfolio which at sometimes was long and sometimes short, we multiplied the simulation with a sinus curve. In this case the agent quickly learns the statistical arbitrage strategy.

In part III we model the underlying dynamics of the financial markets, where actors trade with each other through the order book. To do this we have built an ABM with agents representing retail and institutional investors as well as trend followers and market makers. We have calibrated the model to reflect statistical properties of real high frequency market data. At last we added an actor-critic RL agent attempting to learn optimal systematic market making strategies. Simulating the environment was essentially a computationally heavy task, and we have not been able to simulate anywhere near the amount of episodes that we would have liked. Nevertheless, we observe that the agent learns through time and improves its risk-averse reward consisting of return and a penalty to its position size. This is also reflected in increased return and decreased variance of return through time. Even after 1500 simulations a predefined market maker still has a better performance than the RL agent. However, as we have clear indications that the performance of the RL agent is increasing, we will not reject that it could outperform the predefined market maker if it was trained for a longer period.

19 Future Research

In Part II, we have only included a single traded asset. It could be interesting to include multiple traded assets across different asset classes.

In Part III, we have only tried to calibrate our simulation model to have no autocorrelation of returns, heavy tails and volatility clustering. It could be interesting to attempt to model more of the stylized facts, for example volumes in the order book. Furthermore we have only applied RL to train a market making agent, it could also be interesting to implement RL for other types of investors too.

References

Ardon, Leo et al. (2021). "Towards a fully RL-based Market Simulator". In: URL: https://arxiv.org/pdf/2110.06829.

Basodi, Sunitha et al. (2020). "Gradient amplification: An efficient way to train deep neural networks". In: *Big Data Mining and Analytics* 3.3, pp. 196–207. DOI: 10.26599/BDMA.2020.9020004.

Bellman, Richard (1957). "A Markovian Decision Process." In: *Princeton University Press* 6.5, pp. 679–684. URL: https://www.jstor.org/stable/24900506.

Bollerslev, Tim (1986). "Generalized autoregressive conditional heteroskedasticity". In: *Journal of Econometrics* 31.3, pp. 307–327. ISSN: 0304-4076. DOI: https://doi.org/10.1016/0304-4076(86)90063-1. URL: https://www.sciencedirect.com/science/article/pii/0304407686900631.

Chamberlain, Gary (1983). "A characterization of the distributions that imply mean—Variance utility functions". In: *Journal of Economic Theory* 29.1, pp. 185–201. ISSN: 0022-0531. DOI: https://doi.org/10.1016/0022-0531(83)90129-1. URL: https://www.sciencedirect.com/science/article/pii/0022053183901291.

Cont, Rama (Apr. 2001). "Empirical properties of asset returns: stylized facts and statistical issues". In: *Quantitative Finance* 1, pp. 223–236. DOI: https://doi.org/10.1080/713665670.

Durbin, Michael (2010). *All About High-Frequency Trading*. McGraw-Hill Education - Europe. ISBN: 0071743448.

Ganesh, Sumitra et al. (2020). "Multi-Agent Simulation for Pricing and Hedging in a Dealer Market". In: URL: https://www.jpmorgan.com/content/dam/jpm/cib/complex/content/technology/ai-research-publications/pdf-10.pdf.

Hansen, Bruce E. (1994). "Autoregressive conditional density estimation". In: *Internal Economic REiew* 35(3), pp. 706–730. URL: https://www.ssc.wisc.edu/~bhansen/papers/ier_94.pdf.

Huang, Shengyi and Santiago Ontañón (2020). "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *CoRR* abs/2006.14171. arXiv: 2006.14171. URL: https://arxiv.org/abs/2006.14171.

Kaggle, Optiver realized volatility prediction (n.d.). URL: https://www.kaggle.com/competitions/optiver-realized-volatility-prediction/data.

Karpe, Michaël et al. (2020). "Multi-Agent Reinforcement Learning in a Realistic Limit Order Book Market Simulation". In: URL: https://arxiv.org/abs/2006.05574.

Kissell, Robert (2020). *Algorithmic Trading Methods*. ISBN: 9780128156315.

Kolm, Petter and Gordon Ritter (Jan. 2019). "Modern Perspectives on Reinforcement Learning in Finance". In: *SSRN Electronic Journal*. DOI: 10.2139/ssrn.3449401.

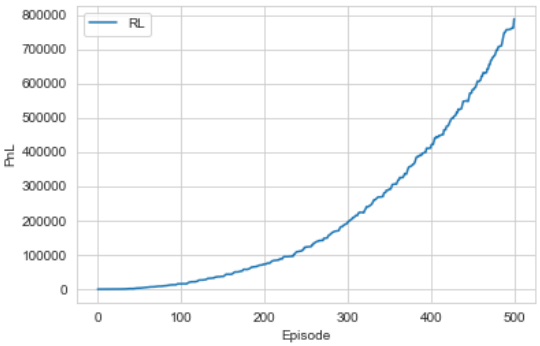
Maeda, Iwao et al. (2020). "Deep Reinforcement Learning in Agent Based Financial Market Simulation". In: URL: https://www.mdpi.com/1911-8074/13/4/71.

Ruppert and David S. Matteson (Jan. 2015). *Statistics and Data Analysis for Financial Engineering*. Springer. DOI: 10.1007/978-1-4939-2614-5.

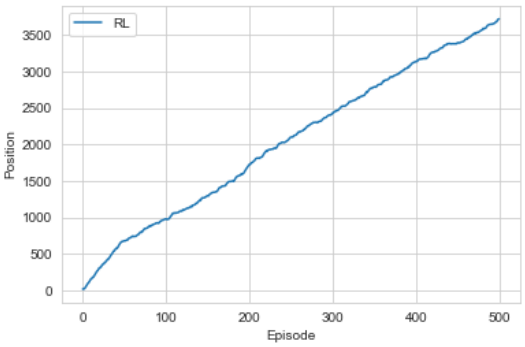
Silver, David et al. (2017). "Mastering the game of Go without human knowledge". In: 7676. DOI: https://doi.org/10.1038\%2Fnature24270.
Spooner, Thomas (2021). *Market Making, Reinforcement Learning and Uncertainty.* URL: https://thalesians.com/videos/.
Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.
Sutton, RS et al. (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *AT&T Labs - Research.* URL: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.
Sutton, S. Richard and G. Andrew Barto (2018). *Reinforcement Learning An Introduction.* Vol. II. ISBN: 978-0-262-03924-6. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.
Vuorenmaa, Tommi A. and Liang Wang (Feb. 2014). "An Agent-Based Model of the Flash Crash of May 6, 2010, with Policy Implications". In: DOI: http://dx.doi.org/10.2139/ssrn.2336772.
Vyetrenko, Svitlana et al. (2019). "Get Real: Realism Metrics for Robust Limit Order Book Market Simulations". In: URL: https://www.jpmorgan.com/content/dam/jpm/cib/complex/content/technology/ai-research-publications/pdf-3.pdf.
Williams, Ronald (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Mach Learn* 8, pp. 229–256. URL: https://doi.org/10.1007/BF00992696.

A Appendix

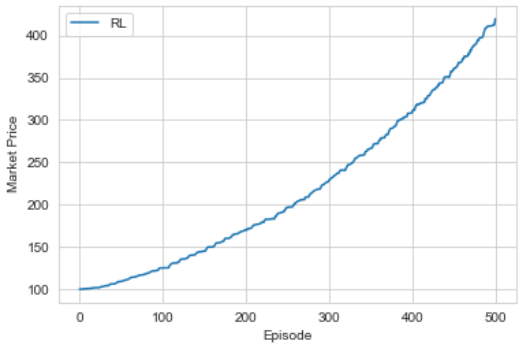
A.1 Importance of Model Assumptions when using RL



(a) *Example of A2C agents return during an episode.*



(b) *Example of A2C agents position size during an episode.*



(c) *Example of market price during an episode.*

,

Figure 28: *A2C agent strategy, when setting both volume and prices with reward defined by pnl. This resulted in it setting the highest buy prices in the market and maximizing its volume, such that it could drive the market upwards.*