SCUOLA
NORMALE
SUPERIORE

Classe di Scienze

PhD in computational methods and mathematical models for sciences and finance

XXXVII cycle

Reinforcement learning for optimal execution, trading and impact games

Settore Scientifico Disciplinare **STAT/04**

Candidate
dr. Andrea Macrì

Supervisor
Prof. Fabrizio Lillo

Co-supervisor
Prof. Stefano Marmi

Academic year 2025–2026

Contents

To Mattia and Maria Elisa,
may this inspire you to always believe in yourselves.

Abstract

This thesis explores how Reinforcement Learning (RL), and in particular how deep neural network architectures can be used to learn optimal trading and optimal execution strategies in a complex environment, such as the financial market. Moreover, we show how these learning algorithms can shed light on how multiple financial agents interact when market impact is generated by their trading.

Therefore, the first result presented in this thesis, shows that Double Deep Q-Learning (DDQL) can successfully recover optimal trading strategies for execution problems when market liquidity is latent and time-varying, both in deterministic and stochastic settings. In fact, considering an Almgren-Chriss environment with temporary and permanent impacts evolving according to different dynamics, we show that DDQL learns the analytical optimal solution whenever it exists, and outperforms standard benchmarks and approximations otherwise. These findings highlight the ability of an agent, modelled with deep RL, to infer optimal behaviour when the underlying market dynamics are unknown, moreover help shed lights on how the behaviour of the agent changes, in terms of performance, when the information provided changes.

In the second contribution, we extend the use of RL to trading environments driven by latent information. We study an optimal trading problem where the trading signal follows an Ornstein–Uhlenbeck process with Markovian regime-switching dynamics for its parameters. To extract information from these latent regimes, and thus find the optimal trading strategy, we combine the Deep Deterministic Policy Gradient (DDPG) algorithm with Recurrent Neural Networks (RNNs), specifically Gated Recurrent Units (GRUs), to model temporal dependencies and the hidden structure that drives the evolution of the trading signal. We design, develop and compare the performance in terms of trading results of three novel architectures: a one-step approach that directly encodes GRU hidden states (*hid-DDPG*), and two two-step approaches that respectively rely on posterior regime probabilities (*prob-DDPG*) or forecasts of the next signal value (*reg-DDPG*). Through extensive numerical experiments and an empirical application to equity pair trading, we find that probabilistic inference of hidden regimes significantly enhances the performance, along with the interpretability, of RL-based trading strategies.

Finally, the third contribution of the thesis examines how autonomous traders, modelled with deep RL, interact in the same market where market impact is present. By modelling two agents with DDQL in a multi-agent Almgren-Chriss framework, we investigate how independent learning dynamics may deviate from traditional game-theoretic equilibria, resulting in supra-competitive equilibria. In fact, our results based on extensive numerical simulations show that, instead of converging to the Nash equilibrium, agents learn on average supra-competitive strategies, aligning closely with the Pareto-optimal solution to the impact game considered. Furthermore, we explore how changes in market volatility affect both the stability and convergence of learned equilibria.

Overall, this thesis studies how deep RL can be a powerful tool for optimal execution, market microstructure, and to study the emergent behaviour of autonomous financial agents. Therefore, using tools from stochastic control, econometrics, and neural network design, we provide novel insights into how RL can learn to act optimally under uncertainty, adapt to latent dynamics, and interact strategically in complex market environments.

List of publications

This dissertation is mainly based on the following list of papers and preprints:

- Macrì Andrea and Fabrizio Lillo (2024). "Reinforcement learning for optimal execution when liquidity is time-varying". In: Applied Mathematical Finance 31.5, pp. 312–342; [1]
- Lillo Fabrizio and Macrì Andrea (2026). "Deviations from the Nash equilibrium in a two-player optimal execution game with reinforcement learning". In: Annals of Operations Research, pp. 1-33; [2]
- Macrì Andrea, Jaimungal Sebastian and Lillo Fabrizio (2025). "Optimal Trading with Partial Information". In: arXiv preprint arXiv:2511.00190. *Under review in Quantitative Finance.* [3]

The contents of this thesis are based on papers and research produced over the years spent at Scuola Normale Superiore in Pisa. Although the topic of reinforcement learning applied to financial problems is the *fil rouge* that connects the different chapters in this thesis, we advise the reader that the mathematical notation across chapters may vary. In fact, each chapter has to be intended as self contained, not fully (or in part) relying on contents or notation from preceding chapters.

[1] [Macrì and Lillo, 2024]
[2] [Lillo and Macrì, 2026]
[3] [Macrì et al., 2025]

Chapter 1

Introduction

«Χαλεπὰ τὰ καλά.»

– Socrates in Plato's *Cratylus*
384b

Foreword

The act of making sequential decisions to maximise some notion of *satisfaction* has always been innate to the human brain. Such decisions -and the actions that follow -stem from a continuous learning process; think of a child learning to walk or a student preparing for an exam. Ideally, behaviour should be optimal in terms of the utility gained and the costs incurred at each stage. Such optimality is encouraged by positive *reinforcement* from the surrounding environment.

Within this framework, Reinforcement Learning (RL) formalises the process of learning an optimal behaviour for an agent. Optimality here is both *sequential* and *global*: the agent must act to maximise cumulative rewards across all possible interactions with a changing and uncertain environment. Therefore, this is a natural framework for problems where the environment is only partially observable and the optimal policy, the optimal set of actions such that rewards are maximal, must be learnt *agnostically*[1]. Among real-world domains where such conditions arise, *finance* stands out as one of the most relevant, mostly due to the partial observability of the processes involved and to the way in which, financial markets by their own nature can be modelled as environment, made by individual -or groups of individuals- agents that interact in

[1]Where by model agnostic we mean that a model for the environment does exist and the agent does not directly account for it, but bases its decisions on states that summarise informations on the model that drives the dynamics of the environment.

order to maximise some reward or utility objective.

In this thesis, Reinforcement Learning is applied to problems of optimal execution, trading and impact games, which represent some of its most natural and promising fields of application in finance.

Reinforcement learning and time varying liquidity

The *first* result, presented in Chapter 3 of this thesis, is an application of Double Deep Q-Learning (DDQL), a deep RL algorithm, to the optimal execution problem when liquidity is time varying, a classical problem in market microstructure theory. Institutional investors, such as brokers or mutual funds, operate on a daily basis in financial markets by exchanging assets through buy and sell orders. When these orders constitute a substantial fraction of the trading market volume for an asset (i.e. stocks, for example) a one-shot execution might prove inefficient from a profit and loss point of view, as trading a substantial amount of volume would adversely impact the price of the security and consequently increase the execution costs. To this end, a financial agent who is facing this problem would divide the initial volume to sell or buy (often referred to as *meta-order*) over a window of time, into many smaller *child* orders. The main task is thus to find balance between two competing objectives: (i) trading gradually to reduce execution costs and adverse market impact, and (ii) trading quickly to limit exposure to price and inventory risk.

Having this in mind, in Chapter 3 we study a liquidation problem, for a trader whom objective is to sell an initial amount of assets q_0 over a time window of length T, with an initial price S_0. The liquidation costs considered in our framework are modelled by the relationship:

$$IS = q_0 S_0 - \sum_{t=1}^{N} \tilde{S}_t v_t \,. \tag{1.1}$$

This cost is commonly referred to as *implementation shortfall* (IS), and relates the initial value of the position to the cash that the trader is able to generate with their trading, over the time interval of length T divided into N time steps of length τ; i.e. when the child orders v_t are executed. Clearly, these costs are hardly equal to zero as the price $\tilde{S}_t$ is adversely impacted by the volumes v_t sold at each time-step t, this impact is called *market impact*, and can be decoupled into two main components: the *permanent impact* and the *temporary impact*, as

modelled in Almgren and Chriss [2000] and thoroughly reviewed in Chapter 2 below.

The problem, in its most classical definition, consists of solving a mean variance functional: $\mathbb{E}\left[IS(\vec{v})\right]+\lambda\mathbb{V}\left[IS(\vec{v})\right]$, where λ is a risk aversion parameter, assuming that the mid-price evolves as:

$$S_t = S_{t-1} - g\left(\frac{v_t}{\tau}\right)\tau + \sigma\tau^{\frac{1}{2}}\xi_t\,,$$
$$\tilde{S}_t = S_{t-1} - h\left(\frac{v_t}{\tau}\right).$$

where $g\left(\frac{v_t}{\tau}\right) = \kappa\frac{v_t}{\tau}$, $h(\frac{v_t}{\tau}) = \tilde{\alpha}\frac{v_t}{\tau}$ are the permanent and temporary impacts respectively.[2]

However, liquidity is dynamic and the impacts are latent variables (as noted in Campigli et al. [2022], for instance), henceforth one would use more complex models able to reproduce this dynamic nature of liquidity. Under these circumstances, the optimisation problem is either analytically difficult to be solved in closed form or simply implies the knowledge of the dynamics that drive the impacts at each time-step.[3]

In this case, the estimation problem cannot be overlooked. On the the one hand estimation introduces biases and might be error prone; one the other, historical data are often not useful for the task as it can be difficult to assess market impact on trades that have already occurred, making calibration of complex models quite difficult.[4]

In this context, reinforcement learning provides an invaluable tool to *learn* how to solve the optimal execution problem in an *agnostic* manner. In fact, the procedure of finding sequential actions, such that some costs are reduced or rewards maximised, perfectly overlaps with the task of optimising the volumes to be sold over time, in order to minimise the liquidation cost.[5]

To address the issues above we have thus developed a DDQL algorithm capable of *learning* the optimal trading strategy when impacts are time-varying and their dynamics are increasingly more complex.

We consider four different model specifications: (i) The model in Almgren and Chriss [2000], (ii) deterministic linearly increasing or decreasing impacts, (iii) regime switching impacts as in Ma et al. [2020] and (iv) square root mean reverting stochastic impacts as in Barger and Lorig [2019].

Solution to case (i) is well known in closed form, as it is a quadratic minimisation problem;

[2] In what follows, we directly use and model the quantity $\alpha = \frac{\tilde{\alpha}}{\tau}$.
[3] See Palmari et al. [2024], Barger and Lorig [2019].
[4] Apart from the case where detailed proprietary data are available.
[5] On the topic we refer to Jaimungal [2022].

while for problem (ii) the solution can be easily obtained via standard numerical optimisation. In case (iii), a theoretical solution is obtainable given the knowledge of the Markov chain that governs the regime switching, along with the values of the regimes, that the impacts might follow. Finally in case (iv), a first order approximation to the real solution is known, it is a perturbed TWAP strategy that depends on the parameters that drive the stochastic differential equations for the regimes dynamics. Clearly, as argued above, impacts are dynamic and latent (especially for stochastic impacts this further complicates the task of finding an optimal strategy) and to this end, we aim at learning the optimal strategy just by looking at the rewards coming from trading, now modelled as cash generated by the trading: $r_t = S_{t-1}v_t - \alpha v_t^2$.

Learning happens over a first *training* phase and is then evaluated over a second *testing* phase. In the former phase we train a DDQL algorithm to solve the optimal execution problem for different liquidity models, while the latter phase we test what learnt by the agent in terms of cash generated while trading. Therefore, performance is assessed via comparison of the cash generated by the RL-agent with the cash that would have been generated if the correct model was known from the beginning, i.e., via the performance index: $\Delta P\&L = (C_{\text{agent}} - C)/C$.

We repeat the experiment twice considering, for both training and testing phases, either less or more informative states of the environment observed by the agent: $\mathbf{s_t} = (q_t, t)$ or $\mathbf{s_t} = (q_t, t, S_{t-1})$.[6] Overall, we find that a RL-agent trained and tested on increasingly more complex models of the environment (i.e., model for the financial market) is able learn the optimal policy even if the parameters of the model are not known.

We find that, when considering settings where the optimal solution is known, the algorithm is able to learn strategies with an expected cost comparable with the optimal one. Results for the Almgren and Chriss [2000] model assuming linear and constant impacts are reported in Table 1.1.

Table 1.1: Constant impact. Comparison between the costs of the DDQL agent and of the A&C model. ΔP&L values are reported in basis points. The standard errors of the nominal raw differences between TWAP and RL cost of the strategies are in parenthesis. Impact parameters used are: $\kappa = 0.001$, $\alpha = 0.002$ and $\sigma = 0.0001$.

Constant Impact				
	A&C	**DDQL**	ΔP&L	ΔP&L
features	$\mathbb{E}[IS]$	$\mathbb{E}[IS]$	**avg.**	**std.dev.**
Q,T	0.2607	0.2698 (0.00071)	-0.455	2.5
Q,T,S	0.2607	0.2652 (0.00046)	-0.225	1.6

[6] A more thorough explanation on Q-learning with deep neural networks is provided in Chapter 2.

More interestingly, when we consider settings where the optimal solution is not available or is available in approximate form, we find that the DDQL algorithm finds better solutions. This indicates that RL could be successfully used in real settings where liquidity is naturally latent and time-varying.

Moreover, we empirically show how the algorithm is sensitive, relative to the achieved IS cost, to increasing levels of volatility. We show how adding the price as a feature, helps overcoming the problem of increased costs when the number of training episodes is limited. Furthermore, we analyse how increasing the number of training episodes helps achieving costs in line with the theoretical optimum, once again confirming the fact that static strategies are equivalent to dynamic ones. At the same time, our methodology provides a way around a possibly lengthy training phase by just adding the observed mid-price as feature to the algorithm. Moreover, we notice how the RL-agent is sensitive to the way in which information about the environment is used during training. For instance, considering the case of linearly increasing or decreasing impacts, modelled as: $\kappa_t = \kappa_0 \pm \beta_\kappa \times t$ and $\alpha_t = \alpha_0 \pm \beta_\alpha \times t$. We see how if the training is performed in a set of episodes containing both the impact cases the performance increases dramatically. This indicates that the more diverse the information contained in the training, the better the solution in the most complex cases.

Deep reinforcement Learning for optimal trading with partial information

The *second* result presented in this thesis, is an application of deep reinforcement learning to trading. In Chapter 4 we tackle the problem of optimal trading, with transaction costs, when the underlying signal to be traded follows a mean reverting process. In this case, the parameters driving the dynamics of the process are time-varying and their regimes evolve according to independent Markov chains.

To this end, the signal S_t to be traded is modelled as an Ornstein-Uhlebeck stochastic differential equation:

$$dS_t = \kappa_t(\theta_t - S_t)dt + \sigma_t dW_t, \tag{1.2}$$

where $W = (W_t)_{t\geq 0}$ is a Brownian motion in a suitable probability space and the parameters $\kappa_t \geq 0$, $\theta_t > 0$, and $\sigma_t > 0$ are, respectively, the long run mean at which the process mean reverts to, the velocity of mean reversion, and the volatility of the process at time t.

Clearly, the existence of a long run mean θ_t creates opportunities for favourable trading strategies, making so called statistical arbitrage opportunities profitable. However, the long run mean value, along with the other time-varying parameters κ_t and σ_t are latent and must be estimated dynamically together with the optimization of the strategy. Through some computation it is straightforward to see how the change in value for S_t is trivially related to the expected level of θ_t conditioned on the filtration $\mathcal{F}_t$ generated by S_t, if for instance κ_t and σ_t are assumed constant. In this context, assuming that a risk neutral trader is interested in maximising the expected returns generated by a discrete time trading strategy, the optimal policy would need to satisfy:

$$\max_{(q_t)_{t\geq 0}\in\mathcal{A}} \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\, r_t(q_t, S_t, S_{t+1}, \lambda) \;\Big|\; I_0, S_0\right], \tag{1.3}$$

where $q_t = I_{t+1} - I_t,\ \forall\ t \in \mathbb{N}$, are changes in the inventory held by the agent I_t that belong to the set of all possible actions in this problem $\mathcal{A}$, and $r_t(q_t, S_t, S_{t+1}, \lambda) := I_{t+1}\left(S_{t+1} - S_t\right) - \lambda\left|q_t\right|$, with λ now being the transaction cost and I_0, S_0 being the initial inventory and signal level respectively.

As rewards depend on the change in value for the signal S_t, and implicitly on the level of long run mean reversion along with the other time-varying parameters, we tackle the optimal trading problem via a novel methodology that blends the two different aspects of the problem: the variability in time of the hidden parameters and the optimisation of the trading strategy, in order to obtain a trading policy that delivers the maximal discounted returns from trading.

In order to so do, we have developed a new methodology by employing a Gated Recurrent Network architecture (GRU), a recurrent neural network algorithm, along with a Deep Deterministic Policy Gradient (DDPG), a deep RL algorithm. We employ this methodology in three different approaches to the optimal trading problem: (i) a one step approach (hid-DDPG), that directly encodes hidden states from the GRU into the RL trader; (ii) a two-step (prob-DDPG) approach that makes use of posterior regime probability estimates for the regimes of θ_t via a pre-trained GRU network, and finally (iii) a two-step approach (reg-DDPG) that relies on forecasts of the next signal value obtained by a pre-trained GRU network.

We employ these three approaches in several scenarios, considering increasingly more complex model dynamics for the signal S_t with time-varying parameters: $\theta_t \in \{\phi_1, \phi_2, \phi_3\}$, $\kappa_t \in \{\psi_1, \psi_2\}$ and $\sigma_t \in \{\xi_1, \xi_2\}$.

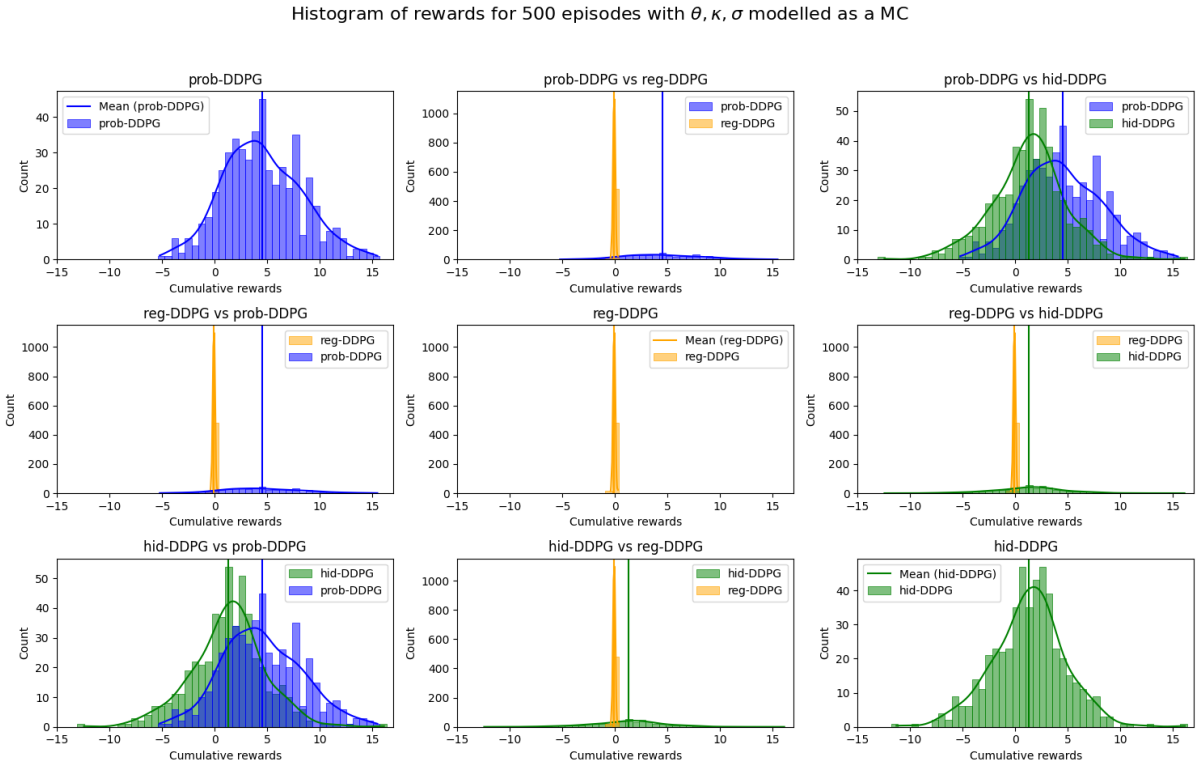


Figure 1.1: Comparison of the cumulative returns for the different DDPG approaches when θ_t, κ_t, and σ_t follow a Markov chain.

Through extensive numerical simulations, we observe that an agent equipped with posterior probability estimates for mean reversion regimes, learnt though the first GRU step, achieves the best performance in terms of returns from trading. In turn, this indicates that supplying structured and interpretable information to a RL algorithm significantly enhances its capability to approximate solutions to inter-temporal optimization issues.

Interestingly, it appears that the type of information used is also crucial. Specifically, using estimates for the next signal values does not lead to significant improvements, this implies that it is better to provide more general information about the process, as for the *hid-DDPG* approach, while ensuring that this information still remains close to the actual parameters governing the underlying dynamics, as shown by the results obtained in the *prob-DDPG* approach. The latter approach indeed proves to be the most effective, yielding the highest returns from trading in both numerical and real data scenarios, as shown in Figure 1.1 and Figure 1.2 respectively.

Overall, we can conclude that the *type* and the *quality* of information provided to the RL algorithm is vital for solving optimization problems with no clear closed form solution and a rather difficult to estimate data generating process. While all three approaches yield a trading policy that, on average, results in either positive or nearly zero returns across all the increasingly

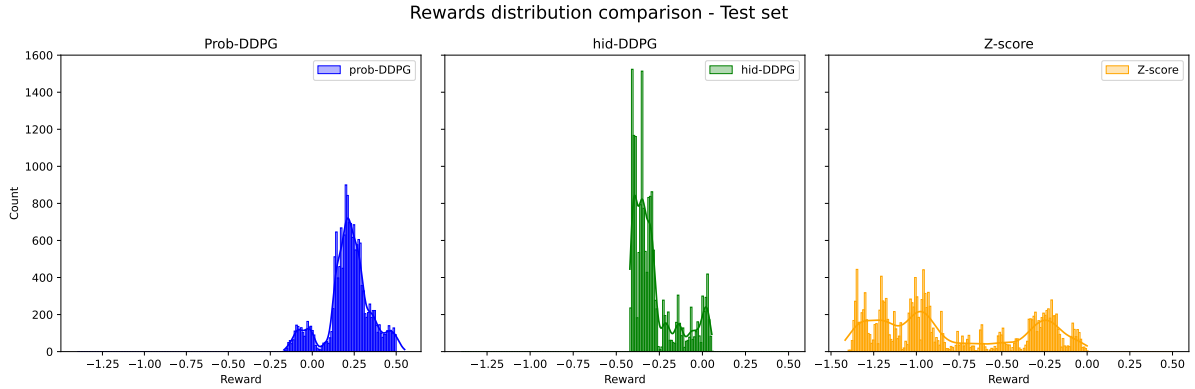


Figure 1.2: Comparison of realised cumulative returns for the *hid-DDPG*, *prob-DDPG* and rolling Z-score strategy on the testing set.

complex scenarios. We empirically show how the most effective strategy first involves the understanding of the dynamics for the data-generating process and, only then, translating this knowledge into the features used by the RL algorithm. Concluding how explainability might not be an advantage just for human understanding but for model performance too.

Optimal execution game with deep reinforcement learning

The *third* result presented in this thesis, is an application of deep reinforcement learning to market impact games. In fact, in Chapter 5 we extend the problem tackled in Chapter 3 to a multiple agents setup from both a theoretical and numerical standpoint.

Turning to a multiple agent setup, in fact, we extend the study on how agents, all tasked with an optimal execution task, interact over a specified window of time, still operating in a market with price impact generated partly by the joint trading of the two agents.

Thus, using the Almgren-Chriss model and considering $n \in \mathbb{N}$ players, tasked with selling an initial quantity q_0 over $[0, T]$ in N time steps, we can define their individual objective to be:

$$\min_{\vec{v}} \ \mathbb{E}[IS(\vec{v})] - \lambda \mathbb{V}[IS(\vec{v})] \qquad s.t. \ \sum_{t=1}^{N} v_t = q_0 \,. \tag{1.4}$$

Obviously, now the mid-price differs from the single agent case and is now, considering agent $k \in \{1, 2, \dots, n\}$ viewpoint:

$$\begin{aligned} S_t &= S_{t-1} - g\left(V_t/\tau\right)\tau + \sigma\tau^{\frac{1}{2}}\xi_t \\ \tilde{S}_t^{\,(k)} &= S_{t-1} - h\left(v_t^{(k)}/\tau\right) \end{aligned} \tag{1.5}$$

where $V_t = \sum_{k=1}^{n} v_t^{(k)}$ are the total trades by each agent and $g(\cdot), h(\cdot)$ are respectively the permanent and temporary impacts, as defined above. Clearly, the problem now is more involved. Even if the impacts are linear and constant in the price, the efficient price process S_t, is impacted by the trades performed by other agents as well. In this way, an agent $k \in n$ faces two different kinds of problems in one: on the one hand, it has to deal with the solution of an optimal execution problem in its classical formulation, while on the other the agent deals with a non-trivial inference problem, as ideally now the optimal volume -in terms of IS costs- that should be traded depends on the trading performed by the other agents as well.

Therefore, the focus now changes from that of a mere liquidation cost minimisation problem to that of finding an equilibrium. In this context, a *unique* Nash equilibrium exists (as shown by Schied and Zhang [2017]) and is a trading strategy such that, considering two players $n = 2$ who aim at liquidating a portfolio q_0, inventory holdings at each t are:

$$q_t^{(1)^*} = \frac{1}{2}(\Sigma(t) + \Delta(t)) \qquad \text{and} \qquad q_t^{(2)^*} = \frac{1}{2}(\Sigma(t) - \Delta(t)), \tag{1.6}$$

with:

$$\Sigma(t) = Qe^{-\frac{\kappa t}{6\alpha}} \frac{\sinh\left(\frac{(N-t)\sqrt{\kappa^2+12\alpha\lambda\sigma^2}}{6\alpha}\right)}{\sinh\left(\frac{N\sqrt{\kappa^2+12\alpha\lambda\sigma^2}}{6\alpha}\right)} \qquad \text{and} \qquad \Delta(t) = \tilde{Q}e^{\frac{\kappa t}{2\alpha}} \frac{\sinh\left(\frac{(N-t)\sqrt{\kappa^2+4\alpha\lambda\sigma^2}}{2\alpha}\right)}{\sinh\left(\frac{N\sqrt{\kappa^2+4\alpha\lambda\sigma^2}}{2\alpha}\right)} \tag{1.7}$$

where $Q = q_0^{(1)} + q_0^{(2)}$ and $\tilde{Q} = q_0^{(1)} - q_0^{(2)}$.

Differently from the single agent case, the price level does not enter directly into the optimal inventory formula, but the permanent impact and the volatility of the asset do, proportionally to agents' risk aversion λ. Starting from this notion of Nash equilibrium we investigate the existence of other non-Nash equilibria and whether these equilibria might be *supra-competitive* if compared to the Nash equilibrium. Leveraging on Dou et al. [2024], we define a supra-competitive equilibrium as: 'An equilibrium where agents set prices (or quantities) that are above (below) in order to maximise the profits or costs that would be achievable in a Nash equilibrium'. Moreover, a collusive equilibrium is generically defined by two key properties: (i) all agents achieve supra-competitive profits, and (ii) there are short-term gains for agents who deviate from on-path equilibrium actions at the expense of others.[7] Therefore, we first

[7][Dou et al., 2024]

characterise the Pareto efficient set of solutions for the multiple agent optimal execution problem in Theo. 1, as that specific set of strategies -one for each agent- such that for one agent to be better off in terms of IS costs, the other has to be worse off; then we prove how the simplest of all strategies, the TWAP strategy (i.e., $v_t^{(k)} = q_0/N$) is indeed the *collusion* strategy in the particular case of a two-agents problem in Theo. 2.

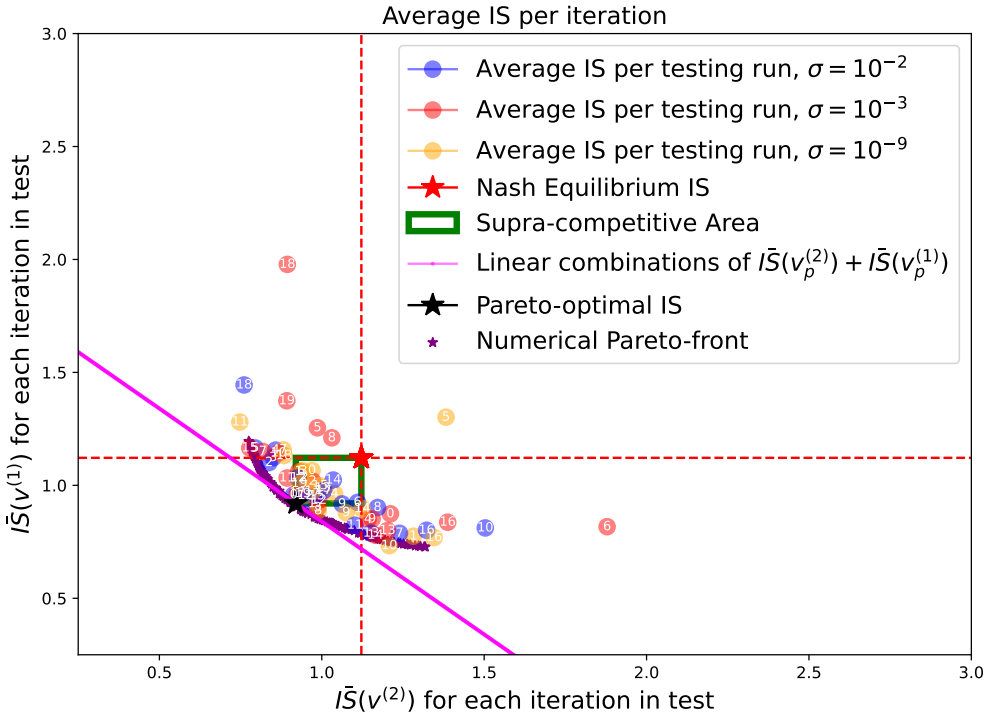


Figure 1.3: Scatter plot of the IS centroids of the two agents in the 20×3 testing runs of $2,500$ iterations each, for all the considered values of σ.

Therefore, using Double Deep Q-Learning (DDQL), we model the interaction between two agents facing an optimal liquidation problem in the same market under different volatility regimes. The agents, trained independently, learn to liquidate their positions while adapting to the indirect price impact generated by the other. Across low, moderate, and high volatility scenarios, their behaviour converges to *supra-competitive* and *Pareto-efficient* strategies, i.e. cost outcomes lying between the Nash equilibrium and full collusion; as reported in Figure 1.3 and in Figure 1.4. Notably, this emergent coordination arises without direct communication, as each agent infers the presence and actions of the other solely from market impact signals. Moreover, this behaviour still appears when volatility levels are mis-specified and the agents are trained and tested on two different levels of volatility for the asset price.

Thus, we can conclude that the emergence of *supra-competitive* strategies is almost natural in a setting where the agents have to *learn* the best policy, based on a partially observable

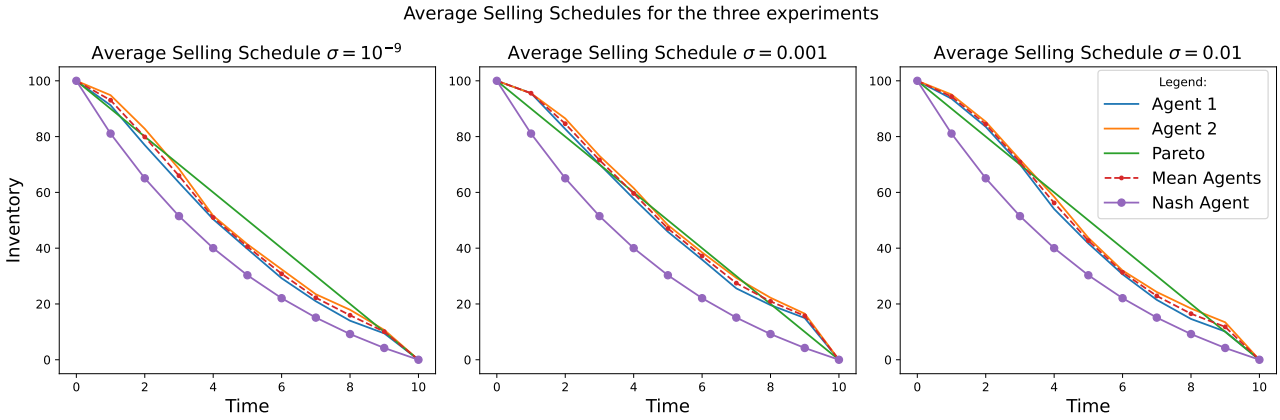


Figure 1.4: Average optimal execution strategy over the 20 testing runs of 2,500 iterations, for all the values of σ considered.

environment. We stress that this happens even if there were no direct instructions to the agents on whether to collude or compete, nor on what kind of more general behaviour the agents have to adopt. However, such behaviour may have adverse effects on market welfare: while in the Nash equilibrium agents trade more aggressively at the beginning, favouring faster information incorporation into prices, collusive strategies distribute trading more evenly over time, thereby slowing down price discovery and potentially distorting the price formation process, having a negative welfare impact on other market participants.

Chapter 2

Background material

Policeman: «So, what you doin here?»
Turkish: «I'm taking the dog for a walk.
What's the problem?»
Policeman: «What's in the car?»
Turkish: «Seats and a steering wheel.»

– G. Ritchie, *The snatch* (2000)

Financial markets and optimal execution problem: an overview

In this section we will introduce the main theoretical workhorses that are going to be used. In each chapter, depending on the particular phenomenon and problem studied, the models and methods presented in the following subsections will be modified accordingly. Any modification or theoretical difference, along with the results obtained in these framework is explained in details in each chapter of reference.

Financial markets functioning and the limit order book

In modern electronic exchanges, traders mainly use two different order types: limit orders and market orders. A limit order (LO) specifies the intention to buy or sell given number of shares at a chosen price. If no counterparties are willing to trade at that price (or at a better price), the order is stored in the limit order book (LOB) until it is either executed or cancelled. Hence, LOs offer price certainty but, obviously, do not guarantee execution. Since the posting of these orders does signal available liquidity on either the sell side or buy side of the book, and they do

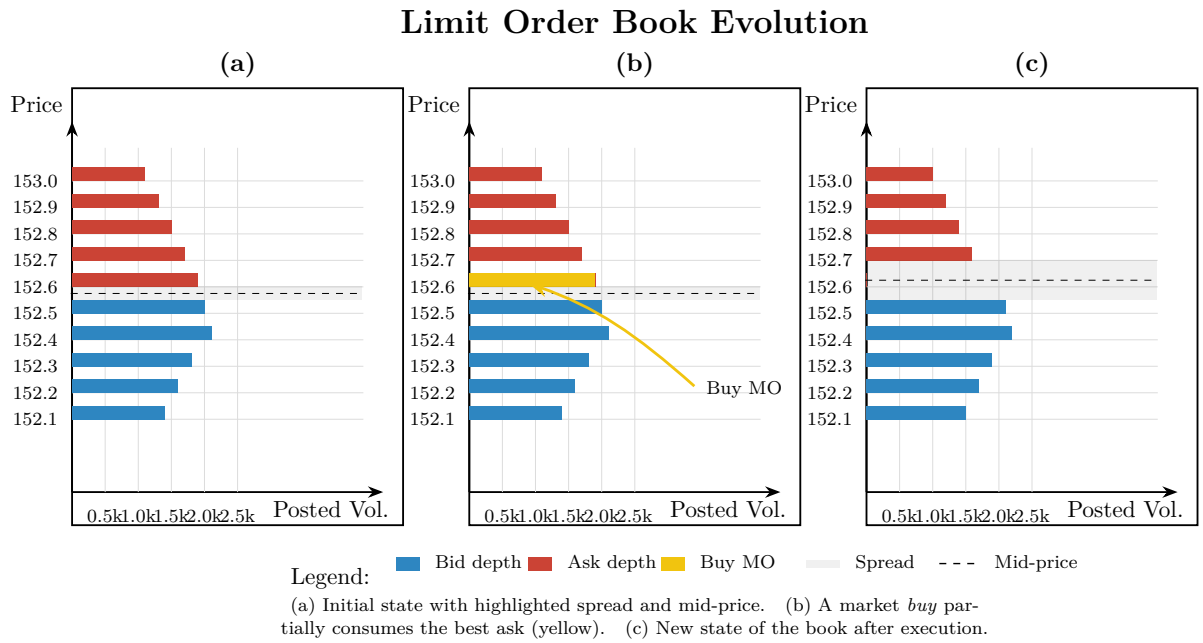


Figure 2.1: Limit Order Book evolution after a buy market order.

so at different price levels, traders who submit limit orders are, in fact, providing liquidity to the market.

By contrast, a market order (MO) is the type of order that implies the immediate purchase or sale of a given quantity of shares and is executed against the best outstanding limit orders in the book. The lowest-priced sell LO defines the ask price, while the highest-priced buy LO defines the bid price. Their difference is called the bid–ask spread, and their average is referred to as the *mid-price*. The smallest permissible change in bid or ask prices is the *tick size*. If a MO consumes one (or more) levels of the book the mid-price changes, and subsequent postings of LOs would eventually replenish the volume levels missing because of the MO that had *consumed* liquidity for a level of price in the book. Figure 2.1 shows how the LOB moves after a buy MO arrives.

Optimal execution problem

Optimal execution is an important problem primarily faced by institutional investors. In fact, investors and market operators such as asset managers, hedge funds, pension funds and brokers regularly interface in financial markets to buy and/or sell large amounts of assets.

Considering the problem to be tackled just with MOs, as this class of orders are the ones that actually trigger transactions in the market, the buy and sell interactions might usually pose

a problem to institutional investors[1] whenever the proportion of the trades is a significant part of the available liquidity for the asset considered. In fact, a direct consequence of trading large volumes of assets in the market is the adverse movement of the mid-price, that usually reacts quite quickly to the consumed liquidity, thus implying an adverse movement of the mid-price due to the consumption of available volumes at each price level. Clearly, the more *aggressive* the MO, in terms of consumed volumes available per level of price in the LOB the stronger the adverse movement of the mid-price at which the agent who is executing the order trades.

Therefore, when an order represents a large share of the trading volume, completing it in a single transaction can be costly and frequently unfeasible due to liquidity limitations. This challenge has led to the classical problem in quantitative finance literature and practice: determining the optimal way to divide large asset blocks (meta orders) into smaller transactions (child orders). Through the optimal execution literature, models are devised to assist market participants in managing their trading costs effectively. These models yield trading schedules that seek to optimally balance the need to trade more *slowly*, thereby reducing execution costs (the gap between a reference price and the average realized trade price) and minimizing negative price impacts from trading activity, against trading more *quickly* to curb inventory and minimize price risk.

This problem, often referred to as the *optimal execution problem* has become the centre of an active area of research in market microstructure. In fact, starting from the first formulations of the problem and solutions, often times naturally set as a decision problem in the seminal works of Bertsimas and Lo [1998] and Almgren and Chriss [2000], many extensions have been proposed. In this dissertation we are going to mostly use the problem formulation given in Almgren and Chriss [2000]. In the following subsections we are going to introduce the main ideas and fundamental notions that justify the setup chosen. Then, in each chapter we will specify the modifications introduced in order to tackle the different problems studied.

The Almgren-Chriss model

Assume that an agent holds an amount q_0 of a single asset with initial mid-price value S_0. Thus, the initial value to be liquidated (bought) by the agent is $q_0 \cdot S_0$. The objective of the agent is to

[1] Mostly buy-side agents such as asset managers, mutual funds, hedge funds, pension funds and insurance companies.

completely unwind (or liquidate) their initial inventory over a time window $[0, T]$ with $T > 0$.[2] We divide the time window of length T into $N > 0$ many time-steps t of length τ, such that $N \cdot \tau = T$. In this way the time window is divided as $t_0 = 0 < \cdots < t_n = n \cdot \tau < \cdots < t_N = N \cdot \tau = T$.

Now, having divided the time window[3] into N steps, at the start of each time interval $[t_n, t_{n+1}]$ the agent has to choose the number of shares to sell (equivalently buy) over the interval, we denote this quantity by v_t.[4] For the selling problem at hand we will call q_t the available inventory at each time t over the time-window. Clearly, $q_{t-1} - q_t = v_t$, meaning that the available inventory will decrease by exactly v_t units at each time step t up to the point where for t_N the remaining inventory $q_N = 0$. The set of all v_t actions to be chosen forms the trading strategy. Fore each time-step t, v_t is the amount of shares sold over the interval of length τ. Moreover, in this context the condition $\sum_{t=0}^{N} v_t = q_0$ holds.

The mid-price is assumed to follow an arithmetic Brownian motion, meaning that :

$$S_t = S_{t-1} - g\left(\frac{v_t}{\tau}\right)\tau + \sigma\tau^{\frac{1}{2}}\xi_t$$
$$\tilde{S}_t = S_{t-1} - h\left(\frac{v_t}{\tau}\right)$$

where, by S_t we denote the *fundamental* price process with a parameter σ for the asset's volatility and $\xi_t \sim \mathcal{N}(0, 1)$ that is a draw from a normal random variable with zero mean and unit standard deviation. The fundamental price process has no drift term, the intuition behind this as reported by Almgren and Chriss [2000], is that we do not have any information about the direction of future price movements. There are two terms that are the most important and link the dynamics of the LOB with those of the Almgren and Chriss [2000] model. In fact, as market participants start detecting the volume that the agent is selling (buying) they naturally adjust their bids (offers) downward (upward). This phenomenon is captured by the market impact that, contrary to volatility, is an endogenous factor in this market model. The market impact due to the agent's trading is decoupled into two main components: *permanent* and *temporary* impacts, $g\left(\frac{v_t}{\tau}\right)$ and $h\left(\frac{v_t}{\tau}\right)$ respectively.

The *temporary impact* refers to temporary imbalances in market's supply and demand caused

[2]The reasoning is exactly the same up to a sign change in the case of an agent wanting to buy a quantity of assets q_0 with initial price S_0.

[3]The assumption that is being made is that the agent that wants to perform such trades has already decided on the time of the day(s) where the execution has to be performed.

[4]In this setup if the agent wants to buy we would have that $v_t \leq 0$.

by agent's trading leading to temporary price movements away from equilibrium, while *permanent impact* means changes in the mid-price due to the agent's trading, which remains at least until the end of the liquidation (acquisition) program. Both the impacts are functions of the average rate of trading v_t/τ.

The core idea behind temporary market impact lies in the strategy of a trader intending to sell a specified number of units, v_t, over the interval between t and $t+1$. The strategy might involve dividing the order into smaller batches to identify optimal points of liquidity. If the total units, v_t are large enough, the execution price might decline steadily over the period from t to $t+1$, partly due to depleting the available liquidity at each successive price level in the LOB. This effect is short-term, particularly as liquidity is restored after each period, leading to a new equilibrium price. Almgren and Chriss [2000] model this phenomenon by incorporating a temporary price impact function $h(\cdot)$, which represents the temporary reduction in the average price per share due to trading at an average rate v_t/τ over one time interval. Consequently, the realised price per share for the sale at t is given by $\tilde{S}_t = S_{t-1} - h\left(\frac{v_t}{\tau}\right)$, but the influence of $h(\cdot)$ does not persist into the subsequent "market" price S_t. The functions $g(\cdot)$ and $h(\cdot)$ can be tailored to any preferred market microstructure model while adhering to some basic convexity conditions.

Impact functions and optimal strategies

Having defined the major features of the framework, the agent has to choose an optimal strategy for the liquidation of the initial assets position of value $q_0 \cdot S_0$. A first important step is to choose the performance criterion that has to be optimised, an intuitive way to define this is to think about the costs paid by the agent throughout the execution window.

Clearly, every agent's trade generates cash (either obtained in the case of a selling program, or spent if the agent is buying), we can indicate the cash obtained for a selling problem over the length of the execution window $[0, T]$, as $\sum_{t=1}^{N} \tilde{S}_t v_t$. Since the real objective for the agent is to spread out their trades over time, such as to not adversely impact the price too much, a first measure for the efficiency of the strategy is the difference between the initial value of the holdings and the cash generated with the trading. Thus, we define a *liquidation* (*acquisition*) strategy, for the agent as $\vec{v}_t = \{v_1, \ldots, v_N\}$, that is the set of signed volumes to be executed over the N steps of the time window $[0, T]$, i.e. the actions that the agent has to take at each time step t. In this setting, we will focus on strategies that belong to the set

of deterministic strategies, these are the admissible strategies in this context. Meaning that $\vec{v} \in \mathcal{A}^{\text{det}} = \left\{\{v_1, \ldots, v_N\} \in \mathbb{R}^N, \quad \sum_{t=1}^N v_t = q_0\right\}$, a deterministic strategy only depends on model parameters and time. Put simply, a deterministic strategy can be computed at the beginning of the execution window and left unchanged throughout the sell (buy) program. Clearly, it does not depend on the evolution of the price.[5]

Having defined the set of admissible strategies, the functional that the agent has to minimise by choosing a $\vec{v} \in \mathcal{A}^{\text{det}}$ is called *Implementation Shortfall* (IS) and is computed as:

$$IS(\vec{v}) = \underbrace{q_0 S_0}_{\text{Initial value}} - \underbrace{\sum_{t=1}^{N} v_t \tilde{S}_t}_{\text{Cash generated}} \tag{2.1}$$

That is nothing more than the difference between the initial value of the agent's position and the cash that the agent is able to generate over the trading window, as anticipated above. The objective, as said, is to minimise this difference, that clearly is hardly exactly equal to zero because of the *market* impact. Thus, the optimisation problem, as the IS is a stochastic functional that reads:

$$\min_{\vec{v}} \{\mathbb{E}\left[IS(\vec{v})\right] + \lambda \mathbb{V}[IS(\vec{v})]\} \tag{2.2}$$

In Equation (2.2) is stated the objective functional to be optimised. The agent wants to find the selling strategy that minimises the expected value of the IS and, at the same time, the strategy that minimises the variance of the IS adjusted by a risk aversion parameter $\lambda \geq 0$. Clearly, the higher the risk aversion the more the agent is concerned about the variance of the IS.

In order to obtain the quantities involved in Equation (2.2) minimisation, one can rewrite the part of Equation (2.1) that accounts for the cash generated by agent as:

$$\sum_{t=1}^{N} v_t \tilde{S}_t = q_0 S_0 + \sum_{t=1}^{N} \left(\underbrace{\sigma \tau^{\frac{1}{2}} \xi_t}_{(a)} - \underbrace{\tau g\left(\frac{v_t}{\tau}\right)}_{(b)} \right) q_t - \sum_{t=1}^{N} \underbrace{v_t h\left(\frac{v_t}{\tau}\right)}_{(c)} \tag{2.3}$$

Where the term (a) accounts for the total effect of volatility, (b) refers to the loss in value of the agent's total position cause by the permanent impact and (c) accounts for the price drop

[5] In Schied et al. [2010] it is shown how in this context deterministic strategies are equivalent to non deterministic ones, i.e. those that adjust with the evolution of the price.

due to the temporary impact.

Now, taking the expected value and the variance, we readily obtain:

$$\begin{aligned}\mathbb{E}[IS] &= \sum_{t=1}^{N} \tau q_t g\left(\frac{v_t}{\tau}\right) + \sum_{t=1}^{N} v_t h\left(\frac{v_t}{\tau}\right) \\ \mathbb{V}[IS] &= \sigma^2 \sum_{t=0}^{N} \tau q_t^2\end{aligned} \tag{2.4}$$

Notice that up to now the model is a very general model, to make it more specific the functional forms for both the permanent and the temporary impact have to be specified. In fact, even if Almgren and Chriss [2000] formulation does not require it, the computation of optimal trajectories is significantly easier taking the impacts to be linear functions of the rate of trading.

Henceforth, in the original formulation of the model the permanent impact is assumed to be constant and $g(\cdot) = \kappa \frac{v_t}{\tau}$, while the temporary impact is assumed to be constant and $h(\cdot) = \tilde{\alpha} \frac{v_t}{\tau}$. The choice of linear impacts, in reality, is not only justified by pure mathematical convenience. In fact, linear impacts prevent the existence of *dynamic arbitrage*.[6] Thus, substituting the linear and constant impacts in Equations. (2.4), we obtain:

$$\begin{aligned}\mathbb{E}[IS] &= \frac{\kappa}{2} q_0^2 + (\tilde{\alpha} + \kappa) \sum_{t=1}^{N} v_t^2 + \kappa \sum_{t \neq k} v_t v_k \\ \mathbb{V}[IS] &= \sigma^2 \sum_{t=0}^{N-1} \left(\sum_{k=t+1}^{N-1} v_k \right)^2\end{aligned} \tag{2.5}$$

Clearly, the optimisation problem in Equation (2.2) is quadratic and admits a unique minimiser, thus the optimal *inventory holding* q_t for each time $t = 1, \dots N$ is:

$$q_t^* = q_0 \frac{\sinh(\omega(T-t))}{\sinh(\omega T)} \tag{2.6}$$

where ω solves $2(\cosh(\omega\tau) - 1) = \frac{\lambda \sigma^2}{2\tilde{\alpha}} \tau^2$. When the trader is risk neutral, $\lambda = 0$, the *optimal strategy* is the TWAP, i.e. the inventory is sold at a constant rate $v^* = (\frac{q_0}{N}, \dots, \frac{q_0}{N})$ or equivalently holding $q_t^* = (N-t)\frac{q_0}{N}$ stocks for each time $t = 1, \dots N$.[7]

Remark 1. *Notice that the TWAP strategy is the optimal strategy, i.e. has minimum cost when* $\kappa < 2\tilde{\alpha}$. *To see this, consider the cost in Equation* (2.3) *expanded as:* $\sum_{t=1}^{N} v_t \tilde{S}_t =$

[6] For a thorough discussion on the topic see Guéant [2016].

[7] For a thorough derivation of this result see Almgren and Chriss [2000] and Guéant [2016].

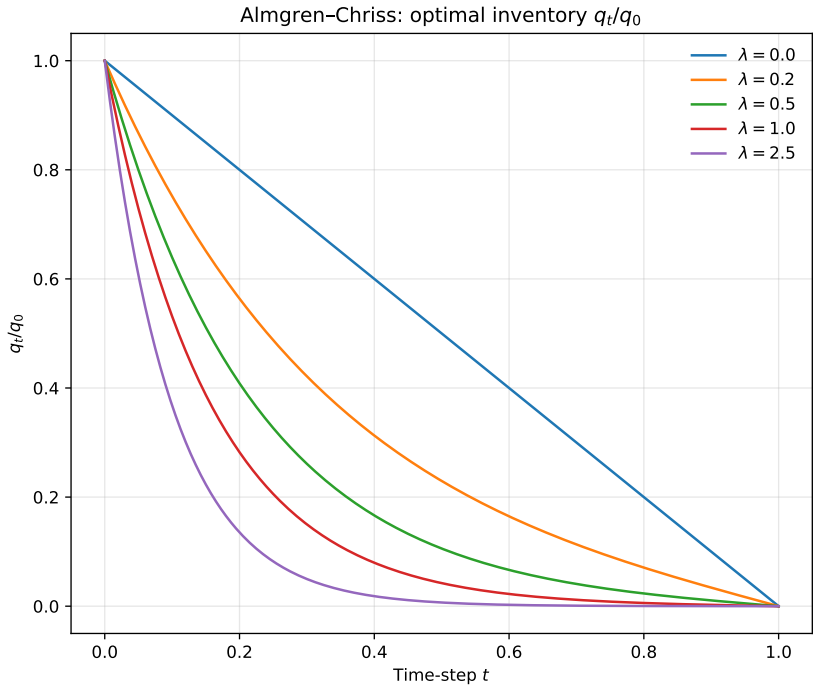


Figure 2.2: Almgren-Chriss: optimal inventory holdings for different levels of risk aversion λ. The other parameters used are: $N = 400$, $\sigma = 0.2$ and $\tilde{\alpha} = 0.001$.

$\sum_{t=1}^{N} v_t \left(S_{t-1} - \tilde{\alpha} v_t\right)$. *Considering the case of* $N = 2$, *the cost equation becomes* $q_0 S_0 - \kappa v_1 v_2 - \tilde{\alpha}(v_1 + v_2)$, *assuming that sell orders have positive sign. Taking the second derivative with respect to* v_1, *having set* $v_2 = q_0 - v_1$, *we obtain* $2\left(2\tilde{\alpha} - \kappa\right)$, *meaning that if* $\kappa < 2\tilde{\alpha}$ *then the TWAP strategy* $v_1^* = v_2^* = q_0/2$ *has minimum cost, while when the contrary is true the TWAP strategy has maximal cost. Notice that when* $\kappa = 2\tilde{\alpha}$ *the cost is independent on the strategy.*

Examples of the optimal inventory holdings for different levels of risk aversion λ are reported in Figure 2.2.

Reinforcement learning

In this section we are going to provide the theoretical framework that backs the main method used throughout this dissertation: Reinforcement Learning (RL). This section is heavily inspired by Sutton and Barto [1998], which is a fundamental book on RL methods, and by Chapter 9 in Dixon et al. [2020] whose focus is more on financial applications of machine learning techniques.

RL is a difficult item to define univocally. In Sutton and Barto [1998] words, RL can be defined in three different but intertwined ways. In fact, RL pertains to a problem, a set of effective solution methods for these problems, and the domain that examines both the issues

and solutions.

Practically, RL is one of the paradigms of Machine Learning (ML) and pertains the task of learning *actions* in order to maximise some sort of numerical *reward*, maximised by mapping effectively *situations* to actions. Hence, the agent tasked with the learning objective is modelled using an algorithm, a set of instructions, with the main aim of finding a suitable policy capable of yielding a maximal numerical returns. Again, in the words of Sutton and Barto [1998], these are inherently closed-loop problems since the actions taken by the learning system influence future inputs. Unlike many machine learning scenarios, the learner isn't provided with explicit actions to take; instead, the learner must experiment to determine which actions result in the highest reward. In the most complex scenarios, actions impact not only immediate rewards but also future situations, consequently influencing all subsequent rewards. The three primary characteristics that set RL problems apart are their closed-loop nature, the absence of explicit action instructions, and their extended temporal influence of actions on rewards.

More specifically, one observes that a RL problem is defined generally by the existence of a learning agent and of an environment where the agent lives and acts, modifying at each interaction the environment itself. On top of this, and going into more details, we can identify four main sub-elements of a RL problem: *a policy*, *a reward signal*, *a value function* and eventually a *model of the environment*.

Imagining an inter-temporal decision problem, where an agent has to interact with the environment at discrete time-steps, receiving a reward from the actions taken, we can give more context to the concepts above following Sutton and Barto [1998].

In fact, a *policy* specifies how a learning agent behaves at a given time. Formally, it maps the perceived states of the environment to the actions to be taken in those states at each time-step where the agent interacts with the environment. A policy can range from a simple lookup table to a complex computational process, such as a search procedure. It is the fundamental component of a reinforcement learning agent, as it is sufficient to define the agent's behaviour, and in general it may be stochastic.

The *reward* signal defines the objective of a reinforcement learning problem. At each time step, the environment provides the agent with a scalar value, the reward. The agent's goal is to maximize the cumulative reward over time. The reward signal determines what outcomes are favourable or unfavourable for the agent. Rewards represent the immediate consequences of the agent's interaction with the environment. The reward depends on both the current state and

the chosen action, but the agent cannot modify how the environment assigns it, only influence it indirectly through the actions.

Rewards are the primary driver of policy adaptation. If an action yields a low reward, the policy is adjusted so that the same action is less likely to be chosen in similar circumstances in the future. While the reward provides an immediate evaluation, the concept of a *value function* captures the long-term desirability of states. The value of a state of the environment is defined as the expected cumulative reward that can be obtained starting from that state and following a certain policy (a *rule* to link states of the environment to actions taken, that will result in a reward for the agent). Thus, while rewards reflect short-term outcomes, values measure long-term benefits, taking into account future states and their associated rewards. For instance, a state that consistently provides low immediate reward may still have high value if it is usually followed by states with substantial rewards.

Rewards are clearly fundamental, as they are the main ingredient to *values*, as the sole purpose of the agent is to maximise them. Nevertheless, *values* are central to decision-making and planning, since actions are selected based on value assessments rather than immediate rewards. The difficulty lies in their estimation: rewards are provided directly by the environment, whereas values must be inferred from experience, often requiring continual updates. Indeed, nearly all RL algorithms focus primarily on efficient value estimation, which has become paramount in the field.

A final key component in some RL systems is a *model of the environment*. Such a model reproduces or approximates the environment's dynamics, allowing the agent to predict the next state and reward given the current state and action. Models enable planning, meaning that the agent can simulate possible futures before acting. Algorithms that rely on models for decision-making are called model-based methods, whereas those that learn exclusively through direct interaction are known as model-free/model-agnostic methods. Modern RL spans the entire spectrum between these two extremes, integrating trial-and-error learning, model building, and planning into unified approaches.

In what follows we will provide introductory aspects of RL, subsequently we will provide a thorough explanation of the deep RL methods used.

Introductory aspects of reinforcement learning

We now focus on RL as the problem of learning from interaction in order to achieve a goal. As specified above, through RL we aim at modelling the behaviour of an *agent*[8] that interacts with an *environment*. The interaction happens at each time step, the agent selects actions in a certain state of the environment, and the latter responds to those actions by changing its state (often-times, as a *consequence* of the actions) and presenting the agent with a reward that depends on the previous state and on the action chosen.

More specifically, agent and environment are modelled to interact at discrete time steps $\mathbb{T} = \{0, 1, 2, \dots\}$, [9] at each of these time-steps $t \in \mathbb{T}$ the agent is sequentially presented with some representation of the environment's state, denoted as $X_t \in \mathcal{X}$, where $\mathcal{X}$ is the set of all possible states for the environment under consideration. On that information, the agent selects an action $A_t \in \mathcal{A}(\mathcal{X}_t)$. Clearly, $\mathcal{A}(\mathcal{X}_t)$ is the set of available actions in each state X_t at time $t \in \mathbb{T}$. Having selected an action at a certain state and at a certain time, the agent receives from the environment a reward $R_{t+1} \subset \mathcal{R} \in \mathbb{R}$, where $\mathcal{R}$ is the set of all possible rewards, and finds itself at a new state of the environment X_{t+1}. A directed graph representation of such mechanism is reported in Figure 2.3. At each $t \in \mathbb{T}$ the aim of the agent is to select a policy

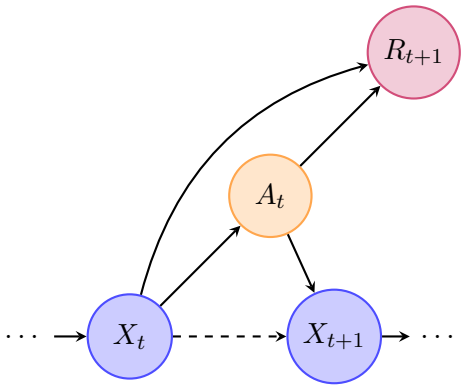


Figure 2.3: Directed graph representation of the State-Action-Reward interaction.

that maps a state X_t to an action A_t, such that rewards received over the long run are maximal.

At each time step t, the agent adopts a map from a state X_t to probabilities of selecting each possible action. This map is called policy and is denoted with $\pi(a|x) = Pr(A_t = a|X_t = x)$, or simply π_t.

[8] Intuitively an agent might be whatever figure is supposed to decide on a matter. To the writer it is easy to imagine the agent as an economic agent, for instance a trader or probably some representative household -in microeconomic terms- that has to allocate some consumption and/or wealth inter-temporally. Agents might as well be robots or, similarly, some controlling algorithm that manages a plant or a self-driving car. In general whatever entity interacts with an environment with the goal of optimising some output coming from the environment itself.

[9] Generalisation to continuos time case is straightforward.

RL is the tool that allows the agent to change the policy depending on the experience they make in terms of rewards. Thus, the agent wants to *adopt an optimal policy* π^* that maximizes the total amount of reward it receives over the long run.

Policies can be deterministic, in the case of a $A_t = f(X_t)$ or stochastic $A_t \sim \pi(\cdot|X_t)$. For a deterministic policy, if the agent visits the same state many times, the chosen action will always be the same. In the case of a stochastic policy, the policy itself becomes a probability distribution over the set of actions, i.e. $\pi(A_t|X_t)$. Moreover the distribution might depend on the current state as a parameter of the distribution itself. Clearly, if the agent visits the same state more than once they may take a different action according to the probability distribution of the possible actions in that state. In the latter case, the action is a draw from a probability distribution and the agent has to either select the correct parameters or the correct distribution.

In order to choose such a policy, the agent has to maximise the *expected return* $G(\mathbb{T}, \{R_u\}_{u\in\mathbb{T}})$ which is a function of the time steps and of the sequence of returns at those time steps. The simplest way to think about this function is the case of $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$, which is the sum of the rewards from time t up to a certain time T. The return can be expressed differently depending on the problem at hand. For instance the sum of rewards is going to be used in (almost) all the works presented in this dissertation, as for the financial problems that are being tackled, exits a precise notion of time that is passing.[10] This first example of G_t implies the existence of a 'final' time step T, this definition is in fact suitable for episodic tasks, i.e. learning that happens repeatedly over episodes of finite length where the agent has to solve the same problem. In the case of a $T = \infty$ to tackle the problem and avoid the series for R_t to diverge it is useful to introduce a discount factor $0 \leq \gamma \leq 1$. In this way, for an infinite horizon, $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. Clearly, if $\gamma < 1$ the infinite sum has a finite value as long as the reward sequence is bounded, a $\gamma = 0$ implies that the agent just cares about the immediate reward from an action, while a $\gamma = 1$ models an agent that equally cares about all the rewards in the same way, by construction in RL problems either the agent has $\gamma = 1$ or the horizon is $T = \infty$.[11]

Rewards' value clearly depend on the state of the environment. The state is whatever

[10] It might not be same in some other cases such as controlling a robot or a self-driving car. In those precise cases the notion of returns has to change according tot he specifics of the problem.

[11] The intuition behind this is that our agent really cares about the rewards, as their sole goal is to maximise them. Of course the agent wants to apply a discount factor to weigh future returns from a policy. It might be helpful for the reader to think about this in terms of a very hungry agent, of course the first bite is going to be very satisfactory, but toward the end of the feast the satisfaction - reward- from eating is going to decrease.

information is available to the agent at each time step $t \in \mathbb{T}$. What the agent would like, ideally, is a state X_t that summarizes past information compactly, such that all relevant information is retained from past states $\{X_u\}_{u=t}^{t-m}$, where m is the time past the beginning of the learning routine. A state signal that succeeds in retaining all relevant information is said to be *Markov*, or to have the *Markov property*.

Formally, if a state signal that has the Markov property, the environment's response at $t+1$ given an action in a state at time t depends only on the state and action at that time t. We can define the environment's dynamics as Markov and by specifying only:

$$p(x', r|x, a) = Pr\{R_{t+1} = r, X_{t+1} = x'|X_t, A_t\} \tag{2.7}$$

For every x', r, X_t and A_t. In other words, the probability of being transitioning from one state to the other depends only on the state where the agent actually is and on the action that they have taken at time t, the rest of the history does not matter.

The best policy π for choosing actions as a function of a Markov state is just as good as the best policy for choosing actions as a function of complete histories.

Thus, the RL task built in this way is also called *Markov Decision Process* (MDP). The transition probability, the quantities in Equation (2.7) and the set of possible states, actions, time-steps and rewards $\mathcal{X}, \mathcal{A}(\mathcal{X}), \mathbb{T}$ and $\mathcal{R}$ completely define a MDP and thus a RL problem. Moreover they are the building blocks used in the *actual* solution to the learning problem. In fact, given the dynamics specified in Equation (2.7), one can compute:

1. Expected rewards for state-action pairs:

$$r(x, a) = \mathbb{E}\left[R_{t+1}|X_t = x, A_t = a\right]$$

2. State-transition probabilities

$$p(x'|x, a) = Pr\left\{X_{t+1} = x'|X_t = x, A_t = a\right\} = \sum_{r \in \mathcal{R}} p(x', r|x, a)$$

3. Expected rewards for state-action-next state triplets

$$r(x, a, x') = \mathbb{E}\left[R_{t+1}|X_t = x, A_t = a, X_{t+1} = x'\right] = \frac{\sum_{r \in \mathcal{R}} r \cdot p(x', r|x, a)}{p(x'|x, a)}$$

Defining the quantities above, makes the value function a derivate of the expected rewards straightforward.

In pedestrian terms, having defined the quantities above the agent can determine how good it is to be in a state; obviously in terms of rewards that can be obtained conditioning to following a policy π. This concept stands behind the idea of *value function*.

Recalling that a policy is, roughly speaking, just a map from $\mathcal{X} \to \pi(A_t|t, X_t)$, the agent can condition themselves to following one particular *way of choosing the actions* π, from a state X at time t thereafter. This relationship can be expressed as a function $V_\pi : \mathcal{X} \to \mathbb{R}$; more specifically, for an MDP, one can define $V_\pi(x)$ as:

$$V_\pi(x) = \mathbb{E}_\pi \left[G_t | X_t = x\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | X_t = x\right] \tag{2.8}$$

Equation (2.8) is the state-value function, and expresses the long run desirability of being in a state for the agent, conditioned to following a certain policy. In a similar way, one can define the action-value function for a policy π, that is function $Q_\pi : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$, as:

$$Q_\pi(x, a) = \mathbb{E}_\pi \left[G_t | X_t = x, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | X_t = x, A_t = a\right] \tag{2.9}$$

Equation (2.9) expresses the value of taking a certain action a at a time step t in a certain state x under a policy π in terms of expected returns.[12] Having defined the properties of MDP above, we can recursively further extend the value function as:

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi\big[G_t \,\big|\, X = x\big] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, X = x\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, X = x\right] \\
&= \sum_a \pi(a|x) \sum_{x'} \sum_r p(x', r|x, a) \left[r + \gamma\, \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, X_{t+1} = x'\right]\right] \\
&= \sum_a \pi(a|x) \sum_{x', r} p(x', r|x, a) \Big[r + \gamma V_\pi(x')\Big].
\end{aligned} \tag{2.10}$$

Equation (2.10) really is a consistency condition between the value of state X_t and the values

[12] It may help the reader to think about this in terms of *quality* of an action A_t in a state X_t following the policy π.

for all the possible successive states. The first lines in Equation (2.10) are intuitive, while the most insightful seems to be the last equation where the sum over all the values of x' and the other over all values of r, into one sum over all possible values of both. Resulting in an easy to read expected value, as it is a sum over all values of the three variables, a, x' and r. For each triple, compute its probability, $\pi(a|x)p(x', r|x, a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value which is the state-value function from following the policy π. Eq (2.10) is called the *Bellman* equation, and is one of the most important building blocks in many areas of mathematics, engineering, finance etc..

Equation (2.10) helps to make an ordering between the policies, this feature will obviously help the agent in finding the optimal policy π^*. In fact, considering $\pi, \pi' \in \Pi$ if $\pi \succeq \pi' \iff V_\pi(X_t) \geq V_{\pi'}(X_t) \quad \forall X \in \mathcal{X}$. The important point to notice is that it can be proven how an optimal policy π^*, is indeed optimal for all the states of the environment, meaning that the previous relation holds $\forall X_t \in \mathcal{X}$.

If we call the optimal policy π^* the associated value function is going to be called $V^*(x)$, the Bellman optimality equation, and is defined as:

$$V^*(X_t) = V_{\pi^*}(X_t) = \max_\pi V_\pi(X_t) \quad \forall X_t \in \mathcal{X}, \quad \forall t \in \mathbb{T}$$

Similarly, for the state-action function:

$$Q^*(X_t, A_t) = Q_{\pi^*}(X_t, A_t) = \max_\pi V_\pi(X_t, A_t) \quad \forall X_t \in \mathcal{X} \; \forall A_t \in \mathcal{A}(\mathcal{X}), \quad \forall t \in \mathbb{T}$$

Now, notice that thanks to the recursive relationships defined above between $V_\pi(X_t)$ and $Q_\pi(X_t, A_t)$, the Bellman optimality equation can be rewritten as:

$$\begin{aligned}
V^*(x) &= \max_{a \in A(x)} Q^*(x, a) \\
&= \max_a \mathbb{E}_{\pi^*}\big[G_t \,\big|\, X_t = x, A_t = a\big] \\
&= \max_a \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, X_t = x, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, X_t = x, A_t = a\right] \\
&= \max_a \mathbb{E}\big[R_{t+1} + \gamma V^*(X_{t+1}) \,\big|\, X_t = x, A_t = a\big] \\
&= \max_{a \in A(x)} \sum_{x', r} p(x', r \mid x, a) \Big[r + \gamma V^*(x')\Big]
\end{aligned} \tag{2.11}$$

Which is an intuitive way of expressing how the value of a state under an optimal policy must be equal to the expected return conditioning to following the optimal policy π^*. A similar result holds for the state-action function $Q_\pi(X_t, A_t)$:

$$\begin{aligned} Q^*(x,a) &= \mathbb{E}\big[R_{t+1} + \gamma V^*(X_{t+1}) \,\big|\, X_t = x,\ A_t = a\big] \\ &= \sum_{x',r} p(x', r \mid x, a)\,\Big[r + \gamma \max_{a'} Q^*(x', a')\Big], \end{aligned} \tag{2.12}$$

Obviously, the existence of an optimal policy (especially in the case of stochastic policies) is far from trivial. A proof of existence is outlined in Ch.9 in Dixon et al. [2020], here we report it in details:

Theorem 1 (Existence of an optimal policy via Bellman's equation as in Dixon et al. [2020])**.** *Let $(\mathcal{X}, \mathcal{A}, p, R, \gamma)$ be a finite Markov decision process, where $\mathcal{X}$ is the set of states, $\mathcal{A}$ the set of actions, $p(\cdot \mid x, a)$ a transition kernel, and $R(x, a)$ bounded rewards. Assume $\gamma \in [0, 1)$. Define the optimal Bellman operator*

$$(Tv)(x) \;=\; \max_{a \in \mathcal{A}} \Big\{ R(x,a) + \gamma \sum_{x' \in \mathcal{X}} p(x' \mid x, a)\, v(x') \Big\}.$$

Then there exists a stationary deterministic policy π^* such that $v^{\pi^*}(x) = v^*(x)$ for all $x \in \mathcal{X}$, where v^* is the unique fixed point of the Bellman optimality equation $Tv = v$.

Proof. Consider the sup–norm $\|v\|_\infty = \max_{x \in \mathcal{X}} |v(x)|$. For any v, w and any x,

$$\begin{aligned} |(Tv)(x) - (Tw)(x)| &= \Big| \max_a \{R(x,a) + \gamma \mathbb{E}[v(X_{t+1}) \mid x, a]\} - \max_a \{R(x,a) + \gamma \mathbb{E}[w(X_{t+1}) \mid x, a]\} \Big| \\ &\le \gamma \|v - w\|_\infty. \end{aligned}$$

Hence T is a γ-contraction on $(\mathbb{R}^{|\mathcal{X}|}, \|\cdot\|_\infty)$. By Banach's fixed-point theorem, there exists a unique v^* such that $Tv^* = v^*$.

Now define a greedy policy with respect to v^*:

$$\pi^*(x) \in \arg\max_{a \in \mathcal{A}} \Big\{ R(x,a) + \gamma \sum_{x'} p(x' \mid x, a)\, v^*(x') \Big\}.$$

Let T^{π^*} denote the Bellman operator for π^*:

$$(T^{\pi^*} v)(x) = R(x, \pi^*(x)) + \gamma \sum_{x'} p(x' \mid x, \pi^*(x)) v(x').$$

By construction, $(T^{\pi^*}v^*)(x) = (Tv^*)(x) = v^*(x)$. Hence v^* is a fixed point of T^{π^*}. Since T^{π^*} is a contraction, its fixed point is unique and equals v^{π^*}. Thus $v^{\pi^*} = v^*$, i.e., π^* is an optimal stationary deterministic policy. □

Remark 2 (Existence via Policy Improvement)**.** *An alternative proof uses the policy improvement theorem. Start from any policy π_0 and define recursively*

$$\pi_{k+1}(x) \in \arg\max_{a\in\mathcal{A}} \Big\{ R(x,a) + \gamma \sum_{x'} p(x' \mid x, a)\, v_k^{\pi}(x') \Big\}.$$

Then $v^{\pi_{k+1}}(x) \geq v_k^{\pi}(x)$ for all x, so the sequence $\{v_k^{\pi}\}$ is monotone increasing and bounded above by v^*. Since the set of stationary policies is finite, this process eventually stabilizes at some $\bar{\pi}$ which must be greedy with respect to $v^{\bar{\pi}}$, hence optimal.

Reinforcement learning methods

Above we have defined RL as that class of methods to solve inter-temporal decision problems, where an agent learns the best policy to satisfy an inter-temporal decision problem. Framing the main elements as *a policy*, *a reward signal*, *a value function* and eventually a *model of the environment*. We then have formalised how an optimal policy will result in maximal discounted rewards from the environment, if actions are chosen according to the policy. This is framed within MDPs theory. However, RL is not the unique way to solve MDP models. In fact, Dynamic Programming (DP) methods attempt at doing the same: finding the optimal policy employing Equation (2.8) and Equation (2.9). The main difference between DP and RL is that in DP methods one attempts to solve these equations exactly, while with RL methods we aim at solving the inter-temporal decision problem relying on data relying on partial information about the environment. Obviously, a solution though DP methods is feasible if and only if the model is completely known to the agent, for instance the law of transition probabilities in Equation (2.7) is known; while RL does not rely on the assumption that the agent perfectly knows the model of the environment, rather it is based on samples from the environment that are samples from the true data generating process unknown to the agent, as reported in Ch.9 of Dixon et al. [2020]. Clearly, given this difference RL methods can be viewed as approximate DP methods that rely on sample trajectories instead of complete models.

There are different RL methods, for the sake of conciseness, in this section we will present Q-learning as a foundational RL method and then we will consider Double Q-Learning, before

presenting the other deep RL method used in this thesis, i.e., the Double Deep Q-Learning (DDQL) and the Deep Deterministic Policy Gradient (DDPG) architecture.

Q-Learning

Q-Learning (Watkins [1989]) is a temporal difference *off-policy* method, meaning that the algorithm is able to learn an optimal policy from transitions in the environment, generated using suboptimal, or even random, policies.

In temporal difference methods such as Q-learning, we aim at updating the action-value function in Equation (2.9) at each time step. Meaning that we would like to obtain $Q^*(x, a)$ as in Equation (2.12) based on data samples as:

$$Q_t(x, a) \leftarrow Q_t(x, a) + \alpha \left[r_t + \gamma \max_{a'} Q_{t+1}(x', a') - Q_t(x, a) \right] . \tag{2.13}$$

where α is a step size that manages the update of $Q_t(x, a)$. Equation (2.13) really provides a *rule* for updating the the sum of discounted rewards given an initial state and action, as expressed in Equation (2.9). Q-Learning is provably convergent (see: Watkins and Dayan [1992]) for finite MDPs assuming that all the states have been visited at least once.Hence, being an off-policy algorithm, rule (2.13) postulates that the policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for convergence is that all pairs continue to be updated.[13] Moreover, the next action a' is the *greedy* action that maximises the next action-value function, in this way we can greedily select the actions that return the highest value for $Q_{t+1}(x', a')$ and eventually adjust our estimate for future discounted rewards according to the gain or loss in $\max_{a'} Q_{t+1}(x', a') - Q_t(x, a)$ discounted, not considering the policy as a map but as a result of a greedy selection of the actions that maximise the $Q_{t+1}(x', a')$ term. In turn, Q-learning does not depend on the policy used to generate the states x, seen by the agent, while searching for the best actions. As pointed out in Dixon et al. [2020], in fact, the rule in Equation (2.13) directly uses observed states to make updates in values of the action-value function.

Clearly, the update rule in Equation 2.13 implies that $Q_t^*(x, a)$ cannot be found by just single observed transitions $(x, a) \rightarrow (r, s', a')$, but iterations over the states and the possible actions are needed for this algorithm to converge. The way to do this is to maintain a table

[13] [Sutton and Barto, 1998]

for the values of $Q(x, a)$ for all couples of (x, a), updating this table every time new transitions occurs, the table is often called *experience replay*, and is a concept used in deep reinforcement learning as well. The Q-learning algorithm is reported in Algorithm 1.

Algorithm 1 Q-Learning Algorithm

Require: Learning rate $\alpha \in (0, 1]$, discount factor $\gamma \in [0, 1)$, exploration rate ε
Require: Initialize $Q(x, a)$ arbitrarily for all $(x, a) \in \mathcal{X} \times \mathcal{A}$
for each episode **do**
  Initialize starting state s
  while x is not terminal **do**
    Choose action a from x using policy derived from Q (e.g. ε-greedy policy)
    Execute a, observe reward r and next state x'
    Update Q-value:

$$Q(x, a) \leftarrow Q(x, a) + \alpha\Big[r + \gamma \max_{a'} Q(x', a') - Q(x, a)\Big]$$

    $x \leftarrow x'$
  end while
end for

Double Q-Learning

In Q-learning, the target policy is the greedy policy derived from the current action-value estimates, meaning that it selects the action with the highest estimated value. This involves taking a maximum over the estimated action values, which implicitly serves as an estimate of the true maximum value. However, such a procedure can introduce a positive bias. To illustrate this, consider a single state s with several actions a, whose true values (e.g. $q(x, a)$) are all zero, while their estimated values $Q(x, a)$ are uncertain and distributed both above and below zero. In this case, the true maximum value is zero, but the maximum of the estimated values will typically be positive. Another way to look at the problem is to think about the fact that we would be using the same samples both to determine the maximizing action and to estimate its value. The systematic upward shift in the estimate that is generated is known as the *maximization bias*.

In order to overcome this bias, in literature forms of double learning have been developed. In particular, Double Q-Learning [14] introduces two independent value functions, $Q_1(x, a)$ and $Q_2(x, a)$. At each update step, one of the two estimators is randomly chosen to be updated independently.

[14] [Hasselt, 2010]

Therefore, of the two separate Q_1 and Q_2 one, say Q_1, is employed to *determine* the greedy action, just as in standard Q-learning, i.e. $A^* = \arg\max_a Q_1(a)$, while Q_2 is used to *evaluate* that action: $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$.

Notice that now the evaluation is obtained with another independent estimator, and this will provide an unbiased estimate: $\mathbb{E}[Q_2(a^*)] = q(a^*)$. Where $q(a^*)$ is the real optimal value obtained with the optimal action a^*.

By reversing the roles of Q_1 and Q_2, one obtains a second unbiased estimate $Q_1(\arg\max_a Q_2(a))$. Hence the *double* in double Q-learning. Consequently, double learning requires twice as much memory, but does not increase the computational cost per iteration.

This idea extends naturally to full Markov Decision Processes. In the double Q-learning algorithm, time steps are alternately assigned to update either Q_1 or Q_2, for instance, by flipping a coin at each iteration. If the coin comes up heads, Q_1 is updated as:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)\big].$$

If it comes up tails, the update is performed symmetrically for Q_2. The behaviour policy can make use of both estimators, for example by adopting an ε-greedy strategy based on the average (or sum) of Q_1 and Q_2. Empirically, Double Q-learning has been shown to eliminate the negative effects caused by maximization bias as reported in Hasselt [2010].

An algorithm for double Q-learning is provided in Algo 2.

Algorithm 2 Double Q-Learning (for estimating $Q_1 \approx Q_2 \approx q$)

Require: Step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(x, a)$ and $Q_2(x, a)$ arbitrarily for all $x \in \mathcal{X}$, $a \in \mathcal{A}(x)$
for each episode **do**
    Initialize starting state x
    while x is not terminal **do**
        Choose action a from x using an ε-greedy policy derived from $Q_1 + Q_2$
        Take action a, observe reward r and next state x'
        if with probability 0.5 **then**
            $Q_1(x, a) \leftarrow Q_1(x, a) + \alpha\Big[r + \gamma Q_2(x', \arg\max_{a'} Q_1(x', a')) - Q_1(x, a)\Big]$
        else
            $Q_2(x, a) \leftarrow Q_2(x, a) + \alpha\Big[r + \gamma Q_1(x', \arg\max_{a'} Q_2(x', a')) - Q_2(x, a)\Big]$
        end if
        $x \leftarrow x'$
    end while
end for

Deep Q-Learning and Double Deep Q-Learning

Taking root from the work of Watkins [1989], Mnih et al. [2013] extended RL techniques using deep neural networks effectively introducing what is called *deep reinforcement learning* that is at the basis of the the main techniques used in throughout this thesis.

The first extension proposed in Mnih et al. [2013] is the deep Q-learning algorithm. The main idea behind consists on replacing the lookup tables used in Algorithm 1 with weights of a neural network. Thus effectively using a feed forward neural network to approximate the Q-function in Equation (2.9), now $Q(x, a; \theta)$ denotes the parametric approximation of the optimal value function $Q^*(x, a)$, where θ represents the parameters of the network.

Learning happens, roughly speaking, as in the classical Q-learning case, but in this case the update rule for the Q-function estimator aims at updating the weights θ of the neural network $Q(\cdot; \theta)$. Thus, in Deep Q-Learning, the function $Q(x, a; \theta)$ is trained to satisfy (2.9) in expectation. At the beginning of the algorithm a deep feed forward neural network, function $Q(x, a; \theta)$, is initialised with random θ weights. Throughout the training, the agent aims at making as much experience as possible, for instance by adopting ε-greedy policies, the transitions are stored in a *replay buffer* of past experiences, now batches of transitions are sampled to be used for the training of the deep Q-network. In fact, considering a finite horizon learning problem, and given a transition (x_t, a_t, r_t, x_{t+1}) sampled from the replay buffer $\mathcal{D}$, the *target* value is defined as:

$$y_t^{\mathrm{DQL}} = r_t + \gamma(1 - d_t) \max_{a'} Q(x_{t+1}, a'; \theta^-), \tag{2.14}$$

where $d_t \in \{0, 1\}$ indicates whether x_{t+1} is terminal, and θ^- are the parameters of a periodically updated *target network*. The target network is a copy of the online network whose parameters are held fixed for a number of iterations to improve the stability of learning. The parameters θ of the online network are optimized by minimizing the expected squared temporal-difference (TD) error:

$$\mathcal{L}(\theta_t) = \mathbb{E}_{(x,a,r,x') \sim \mathcal{D}}\Big[\big(y_t^{\mathrm{DQL}} - Q(x, a; \theta_t)\big)^2\Big]. \tag{2.15}$$

To update the weights it is employed the gradient of the loss with respect to θ_t, that yields the update direction:

$$\nabla_{\theta_t} \mathcal{L}(\theta_t) = -\,\mathbb{E}_{(x,a,r,x') \sim \mathcal{D}}\Big[\big(y_t^{\mathrm{DQL}} - Q(x, a; \theta_t)\big)\, \nabla_{\theta_t} Q(x, a; \theta_t)\Big]. \tag{2.16}$$

Hence, the parameter update at each iteration is performed as:

$$\theta_{t+1} = \theta_t + \alpha(y_t^{\text{DQL}} - Q(x_t, a_t; \theta_t))\nabla_{\theta_t} Q(x_t, a_t; \theta_t) \tag{2.17}$$

where $\alpha > 0$ denotes the learning rate. Finally, to stabilize learning, the target network parameters θ^- are periodically updated by copying the current online parameters:

$$\theta^- \leftarrow \theta \quad \text{every } \tau \text{ steps.} \tag{2.18}$$

Clearly, as the same network $Q(x, a; \theta)$ is being used the overestimation bias on action values might still be an issue, especially in complex environment such as those that model financial markets behaviour.

This happens because the target value in Equation (2.14) uses the same value estimates both to *select* and to *evaluate* an action. In other words, the maximization operator in $\max_{a'} Q(x', a'; \theta^-)$ tends to select actions whose estimated values are optimistically high due to random approximation errors. As a result, the learned action-value function $Q(x, a; \theta)$ can become biased upward, potentially leading to suboptimal or unstable policies.

To address this problem, Van Hasselt et al. [2016] have proposed double deep Q-learning (DDQL) as a solution to the overestimation problem in deep Q-learning. As before, two different estimators for the action-value function are used. This time, the second estimator is a deep feed forward neural network as well. This, generalizes the Double Q-learning idea Hasselt [2010] to the deep-learning setting. The key idea is to decouple the action *selection* from its *evaluation* by employing two networks: the *online network* with parameters θ_t and the *target network* with parameters θ_t^-. The only element that changes now is the target, that now becomes a target network:

$$y_t^{\text{DDQL}} = r_t + \gamma(1 - d_t)\, Q\big(x_{t+1}, \arg\max_{a'} Q(x_{t+1}, a'; \theta_t); \theta_t^-\big). \tag{2.19}$$

Here, the *online network* $Q(\cdot; \theta_t)$ is used to determine the greedy action:

$$a^* = \arg\max_{a'} Q(x_{t+1}, a'; \theta_t)\,,$$

while the *target network* $Q(\cdot; \theta_t^-)$ is used to evaluate its value. This decoupling mitigates the maximization bias since the evaluation is based on a separate set of parameters not directly involved in the action selection.

The online network is trained by minimizing the mean squared Bellman error as in DQL:

$$\mathcal{L}(\theta_t) = \mathbb{E}_{(x,a,r,x')\sim\mathcal{D}}\Big[\big(y_t^{\mathrm{DDQL}} - Q(x,a;\theta_t)\big)^2\Big], \tag{2.20}$$

where y_t^{DDQL} is defined in (2.19). The corresponding stochastic gradient update is given by:

$$\theta_{t+1} = \theta_t + \alpha(y_t^{\mathrm{DDQL}} - Q(x_t,a_t;\theta_t))\nabla_{\theta_t} Q(x_t,a_t;\theta_t) \tag{2.21}$$

and the target network parameters are periodically synchronized with the online ones, i.e.

$$\theta_t^- \leftarrow \theta_t \quad \text{every } \tau \text{ steps.} \tag{2.22}$$

Equations (2.19)-(2.21) define the Double Deep Q-Learning update rule. By using the online network for action selection and the target network for value evaluation, Double DQL effectively reduces the overestimation bias inherent in DQL. Empirically, this results in more accurate value estimates, improved stability, and better policy performance across a wide range of benchmark tasks, including the Atari 2600 suite, as noticed by Van Hasselt et al. [2016]. The algorithm for double deep Q-learning is reported in Algorithm 3

Algorithm 3 Double Deep Q-Learning (DDQL)

Require: discount $\gamma \in [0,1)$, step size $\alpha > 0$, exploration ε, target period τ, batch size B
Require: replay capacity N
Initialize online network $Q(x,a;\theta)$ and target network $Q(x,a;\theta^-) \leftarrow \theta$
Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$
for each episode **do**
    observe initial state x
    while episode not terminal **do**
        With probability ε select a random action a; otherwise $a \leftarrow \arg\max_{a'} Q(x,a';\theta)$
        Execute a, observe reward r, next state x', and terminal flag $d \in \{0,1\}$
        Store transition (x,a,r,x',d) into $\mathcal{D}$ (cancel oldest if $|\mathcal{D}| > N$)
        Sample a mini-batch $\{(x_i,a_i,r_i,x_i',d_i)\}_{i=1}^B$ from $\mathcal{D}$
        for each sample $i = 1,\ldots,B$ **do**
            $a_i^* \leftarrow \arg\max_{a'} Q(x_i',a';\theta)$ // action *selection* by online net
            $y_i \leftarrow r_i + \gamma(1-d_i)\,Q(x_i',a_i^*;\theta^-)$ // action *evaluation* by target net
        end for
        Update θ by one step of SGD on $\frac{1}{B}\sum_{i=1}^B \ell(y_i - Q(x_i,a_i;\theta))$ (e.g., MSE)
        Every τ environment steps: $\theta^- \leftarrow \theta$ // hard target update
        $x \leftarrow x'$
        Decrease ε
    end while
end for

Deep Deterministic Policy Gradient

While Q-learning and its extensions such as DQL and Double DQL are designed for environments with *discrete* action spaces, many real-world applications require continuous control. The *Deep Deterministic Policy Gradient* (DDPG) algorithm, introduced by Lillicrap et al. [2015], extends the principles of Q-learning to continuous action domains by combining value-based and policy-based learning in an actor-critic framework.

In DDPG, the policy (actor) is modelled as a deterministic function

$$a = \mu(x; \theta^{\mu}), \tag{2.23}$$

which maps each state $x \in \mathcal{X}$ to an action $a \in \mathcal{A} \subseteq \mathbb{R}^n$. The critic $Q(x, a; \theta^Q)$ approximates the action-value function, while the actor $\mu(x; \theta^{\mu})$ is optimized to maximize the expected return estimated by the critic. The critic parameters θ^Q are learned by minimizing the mean squared Bellman error. Given a transition (x_t, a_t, r_t, x_{t+1}) sampled from the replay buffer $\mathcal{D}$, the target is defined as:

$$y_t = r_t + \gamma(1 - d_t)\, Q^{-}\Big(x_{t+1},\, \mu^{-}(x_{t+1}; \theta^{\mu^{-}});\, \theta^{Q^{-}}\Big), \tag{2.24}$$

where $\theta^{Q^{-}}$ and $\theta^{\mu^{-}}$ denote the parameters of the target networks for the critic and the actor, respectively, and $d_t \in \{0, 1\}$ indicates whether the episode has terminated. The critic is then updated by minimizing the following loss function:

$$\mathcal{L}(\theta^Q) = \mathbb{E}_{(x,a,r,x')\sim\mathcal{D}}\Big[\big(y_t - Q(x, a; \theta^Q)\big)^2\Big]. \tag{2.25}$$

The actor aims to maximize the expected return

$$J(\theta^{\mu}) = \mathbb{E}_{x\sim\mathcal{D}}[Q(x, \mu(x; \theta^{\mu}); \theta^Q)].$$

The deterministic policy gradient theorem Silver et al. [2014] provides the gradient of this objective as:

$$\nabla_{\theta^{\mu}} J(\theta^{\mu}) = \mathbb{E}_{x\sim\mathcal{D}}\Big[\nabla_a Q(x, a; \theta^Q)\big|_{a=\mu(x)}\, \nabla_{\theta^{\mu}} \mu(x; \theta^{\mu})\Big]. \tag{2.26}$$

In practice, this expectation is approximated via mini-batches sampled from the replay buffer.

As in DQL, DDPG employs separate target networks to stabilize learning. However, instead of being copied periodically, they are updated gradually through a *soft update* (Polyak

averaging):

$$\theta^{Q^-} \leftarrow \tau\,\theta^{Q} + (1-\tau)\,\theta^{Q^-}, \qquad \theta^{\mu^-} \leftarrow \tau\,\theta^{\mu} + (1-\tau)\,\theta^{\mu^-}, \tag{2.27}$$

where $\tau \in (0,1)$ is a small constant (typically $\tau \approx 10^{-3}$). Since the policy $\mu(x;\theta^\mu)$ is deterministic, exploration during training is encouraged by adding temporally correlated noise to the selected action:

$$a_t = \mu(x_t;\theta^\mu) + \mathcal{N}_t, \tag{2.28}$$

where $\mathcal{N}_t$ is often modelled as an Ornstein–Uhlenbeck process or as Gaussian noise. The algorithm for deep deterministic policy gradient is reported in Algorithm 4.

Algorithm 4 Deep Deterministic Policy Gradient (DDPG)

Require: discount $\gamma \in [0,1)$, actor step size $\alpha_\mu > 0$, critic step size $\alpha_Q > 0$, soft update $\tau \in (0,1]$, batch size B
Require: replay capacity N, exploration noise process $\mathcal{N}_t$
Initialize actor $\mu(x;\theta^\mu)$ and critic $Q(x,a;\theta^Q)$ with random weights
Initialize target networks: $\theta^{\mu^-} \leftarrow \theta^\mu$, $\theta^{Q^-} \leftarrow \theta^Q$
Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$
for each episode **do**
    observe initial state x
    while episode not terminal **do**
        (Exploration) $a \leftarrow \mu(x;\theta^\mu) + \mathcal{N}_t$
        Execute a, observe reward r, next state x', terminal flag $d \in \{0,1\}$
        Store (x,a,r,x',d) in $\mathcal{D}$ (evict oldest if $|\mathcal{D}| > N$)
        Sample mini-batch $\{(x_i,a_i,r_i,x_i',d_i)\}_{i=1}^{B}$ from $\mathcal{D}$
        Critic target: $y_i \leftarrow r_i + \gamma(1-d_i)\,Q^-(x_i',\,\mu^-(x_i';\theta^{\mu^-});\,\theta^{Q^-})$
        Critic update: minimize $\dfrac{1}{B}\displaystyle\sum_{i=1}^{B}\left(y_i - Q(x_i,a_i;\theta^Q)\right)^2$ w.r.t. θ^Q
        Actor update: ascend on $\dfrac{1}{B}\displaystyle\sum_{i=1}^{B}\nabla_a Q(x_i,a;\theta^Q)\big|_{a=\mu(x_i)}\,\nabla_{\theta^\mu}\mu(x_i;\theta^\mu)$
        Soft target updates:
            $\theta^{Q^-} \leftarrow \tau\,\theta^{Q} + (1-\tau)\,\theta^{Q^-}$
            $\theta^{\mu^-} \leftarrow \tau\,\theta^{\mu} + (1-\tau)\,\theta^{\mu^-}$
        $x \leftarrow x'$
    end while
end for

Part I

Optimal execution and trading problems in a single agent setting

Chapter 3

Deep reinforcement learning for optimal execution

> "A computer can never be held accountable, therefore a computer must never make a management decision."
>
> *Internal note IBM, 1979*

The contents of this chapter have been published in Macrì and Lillo [2024].

Abstract Optimal execution is an important problem faced by any trader. Most solutions are based on the assumption of constant market impact, while liquidity is known to be dynamic. Moreover, models with time-varying liquidity typically assume that it is observable, despite the fact that, in reality, it is latent and hard to measure in real time. In this contribution we show that the use of Double Deep Q-learning, a form of Reinforcement Learning based on neural networks, is able to learn optimal trading policies when liquidity is time-varying. Specifically, we consider an Almgren-Chriss framework with temporary and permanent impact parameters following several deterministic and stochastic dynamics. Using extensive numerical experiments, we show that the trained algorithm learns the optimal policy when the analytical solution is available, and overcomes benchmarks and approximated solutions when the solution is not available.

3.1 Introduction

The problem of optimally executing financial trades has been extensively studied for at least two decades. Building on the seminal papers of Bertsimas and Lo [1998] and Almgren and Chriss [2000], the concept of dividing large sell (buy) orders into smaller trades, considering risks and rewards, has been widely applied and extended through mathematical optimisation. Starting with the seminal, yet simplistic, Almgren-Chriss setting, which assumes an arithmetic random walk motion for mid-price movements with linear permanent and temporary price impacts, the literature has progressed to either more complex models, notable examples are Guéant and Lehalle [2015], Gueant et al. [2012], Cartea and Jaimungal [2016], Casgrain and Jaimungal [2019], Cartea et al. [2015], Bouchaud et al. [2009, 2003], Gatheral et al. [2012], Obizhaeva and Wang [2013], or more advanced dynamic programming techniques that make use of the cost of execution for a large order, to adjust the trading as price moves in Lorenz and Almgren [2011]. Since these approaches make use of specific market impact models, which might or might not accurately describe real data, the non-parametric approach based on Reinforcement Learning (RL) has increasingly been used to solve optimal liquidation problems. As described by Sutton and Barto [1998], RL models an agent that sequentially learns optimal policies in a generalised environment, optimising a reward function with respect to future discounted cumulative rewards based on actions taken at each time step. Q-learning is one of the most widely used RL algorithms. Q-learning in tabular form, was firstly applied by Nevmyvaka et al. [2006] in the context of optimal execution. In their setting a tabular Q-agent has to decide how many stocks to sell within a certain time window using limit orders. The amount and the execution price at which these stocks are sold depend on the time left for execution, on the inventory left to be executed, and on market prices. Hendricks and Wilcox [2014], using real data and assuming no permanent price impact, adopts a hybrid approach, both analytical and RL-based, where starting from the optimal execution à la Almgren-Chriss, a stylised agent, still modelled with a classic tabular Q-Learning algorithm, is able to modify its trade execution schedule to minimise a reward function that corresponds to the Implementation Shortfall (see below for the definition).

More recently, due to the recent advancements in machine learning methods, the lookup table has been replaced by deep neural nets, thus Deep Q-Learning (DQL) has been widely applied[1] to various financial problems, including market making as in Abernethy and Kale [2013], Kumar

[1]For a more comprehensive overview on the state of the art on RL methods in finance please refer to Sun et al. [2023] and Hambly et al. [2023]

[2020], portfolio optimisation as in Yu et al. [2019], trading as in Kearns and Nevmyvaka [2013], Wei et al. [2019], Briola et al. [2021], Jaisson [2022] and optimal trade execution as in Ning et al. [2021], Dabérius et al. [2019], Karpe et al. [2020], Jeong and Kim [2019], Schnaubelt [2022]. For what concerns optimal execution problems, a major recent contribution is the paper by Ning et al. [2021], where a Double Deep Q-Network (DDQN) algorithm for optimal execution is introduced, overcoming the bias in Q-value estimation found in traditional DQL. This technique, incorporating two neural networks (Q-network and target Q-network), improves the policy estimation accuracy. In Ning et al. [2021], the authors consider an Almgren-Chriss framework without permanent impact and with a constant (across time and stocks) temporary impact. When applied to real data, they found that DDQN outperforms the Time-Weighted Average Price (TWAP) strategy in seven out of nine stocks considered. Other notable contributions include Jeong and Kim [2019], where DQL devises a trading strategy tested on various stock indexes, and Dabérius et al. [2019], comparing DDQN and Proximal Policy Optimisation (PPO) algorithms on artificial data. PPO and DDQN outperform TWAP when it is not optimal and perform similarly to a slice-and-dice strategy when it is.

Our paper considers the application of RL to optimal execution in a more realistic environment. In particular, we consider a setting where liquidity, as described by the temporary and permanent impact coefficients, are not constant but time varying. It is well known that liquidity is a dynamic and latent variable. Some papers (for example, Barger and Lorig [2019] and Fouque et al. [2022]) have considered optimal execution with time varying liquidity, but the underlying assumption is that the model's parameters are known, while their real time estimation is typically very complicated and noisy as shown, for instance, in Campigli et al. [2022] and Mertens et al. [2022]. Thus estimating the impact parameters and simultaneously adjusting the trading strategy is very challenging and here, in a simulated environment, we test if RL can be successfully used. To this end, we consider an Almgren-Chriss market where the permanent and temporary coefficients are linear but time-varying. We consider different dynamics, from very simple and deterministic ones to the case when they follow a stochastic process. We find that when a closed form solution is known, the RL algorithm finds very similar strategies with similar costs. More interestingly, when the solution is not known or is known only in some regime parameters of low variability, the RL algorithm outperforms the benchmarks. In summary, the aim is to train a 'model robust' agent capable of adapting its trading strategy based on the current liquidity profiles in the market. This entails leveraging minimal

information, represented as features to the neural networks, to agnostically select the correct model and selling schedule to follow.

The chapter is organised as follows: Section 2 introduces baseline models and the DDQL algorithm, Section 3 discusses the main results, and finally Section 4 provides conclusions and outlines further research directions.

3.2 Methods

We consider the problem of a trader who wants to liquidate a portfolio of q_0 stocks within a time window $[0, T]$. The goal is to unwind the portfolio *optimally* using arrival price as benchmark, i.e. minimising the difference between the initial and final portfolio values. We assume that the trading influences the stock price via permanent and temporary impacts. The magnitude and dynamics of these impacts change during the trading experiments. Initially, we assume that price dynamics follows the classic Almgren and Chriss [2000] model, and then we test these results against those obtained by an agent modelled with a DDQL algorithm. We then further modify and complicate market dynamics to test RL's capabilities in more complex environments. This section introduces the main tools used in the experiments, namely the baseline financial models and the DDQL algorithm used to model our agent.

3.2.1 Market impact models

Baseline Model. The Almgren and Chriss [2000] (A&C) model is used as our baseline model. The model considers a trader who wants to liquidate an initial inventory q_0 within a time window $[0, T]$, which is divided into N sub-intervals of length $\tau = \frac{T}{N}$. Setting $t = 1, 2, ..., N$ the model reads:

$$\begin{aligned} S_t &= S_{t-1} - g\left(\frac{v_t}{\tau}\right)\tau + \sigma\tau^{\frac{1}{2}}\xi_t \\ \tilde{S}_t &= S_{t-1} - h\left(\frac{v_t}{\tau}\right) \end{aligned} \tag{3.1}$$

The mid-price for the stock at time t, S_t, evolves because of a diffusion part ξ_t (a draw from a standard normal random variable) multiplied by the price volatility σ and a drift which depends on the number of shares v_t sold during the interval $[t-1, t]$. Specifically, the mid-price S_t is impacted by the *permanent impact* term $g_t\left(\frac{v_t}{\tau}\right)$, assumed to be linear and constant: $g\left(\frac{v_t}{\tau}\right) = \kappa\frac{v_t}{\tau}$. The price received by the trader, $\tilde{S}_t$, is equal to the mid-price impacted by a *temporary impact*

term also assumed to be linear and constant in time: $h(\frac{v_t}{\tau}) = \tilde{\alpha}\frac{v_t}{\tau}$. In what follows, we directly use and model the quantity $\alpha = \frac{\tilde{\alpha}}{\tau}$. The aim is to find an optimal quantity to sell v_t for each sub-interval t, minimising the *Implementation Shortfall* (IS) cost functional given by:

$$IS = S_0 q_0 - \sum_{t=1}^{N} \tilde{S}_t v_t \tag{3.2}$$

Since IS is stochastic, A&C assumes a mean variance optimisation (or equivalently a CARA utility maximiser) characterised by a risk aversion parameter λ. A&C shows that the optimal inventory holding q_t for each time $t = 1, \dots N$ is:

$$q_t^* = q_0 \frac{\sinh(\omega(T-t))}{\sinh(\omega T)} \tag{3.3}$$

where ω solves $2(\cosh(\omega\tau) - 1) = \frac{\lambda\sigma^2}{2\tilde{\alpha}}\tau^2$.

When the trader is risk neutral, $\lambda = 0$, the optimal strategy is the TWAP, i.e. the inventory is sold at a constant rate $v^* = (\frac{q_0}{N}, \dots, \frac{q_0}{N})$ or equivalently holding $q_t^* = (N-t)\frac{q_0}{N}$ stocks for each time $t = 1, \dots N$.

Time-dependent impacts baseline model. We then consider a different model by assuming that the permanent and the temporary impacts are *time-varying*. This is a more realistic assumption, since in real-market situations liquidity is variable and fluctuates. In fact, as discussed in Campigli et al. [2022], price impacts have strong intraday time-dependency since they react quite quickly to changing market conditions. Impact dynamics usually have a deterministic and a stochastic component. In this setting we consider deterministic dynamics, while in the next one we will consider stochastic dynamics for the impacts.

In general, if both impacts are time-varying, the functional to be minimised is similar to the one on the right-hand side of Equation (3.2) but with time-varying impacts, i.e.:

$$\sum_{t=1}^{N} \tilde{S}_t v_t = \sum_{t=1}^{N} (S_{t-2} - \kappa_{t-1} v_{t-1}) v_t - \sum_{t=1}^{N} \alpha_t (v_t)^2 \tag{3.4}$$

We consider here the simplest deterministic dynamics for κ_t, the coefficient of the permanent impact $g(\frac{v_t}{\tau}) = \kappa_t \frac{v_t}{\tau}$, namely we consider a linear temporal dependence, $\kappa_t = \kappa_0 \pm \beta_\kappa \times t$. The time varying temporary impact is assumed to be similarly specified, i.e., $h(v_t) = \alpha_t v_t$, with dynamics $\alpha_t = \alpha_0 \pm \beta_\alpha \times t$. Since impacts must be positive, we choose the parameters in such

a way that both $\kappa_t > 0$ and $\alpha_t > 0$, $\forall t$.

The optimal schedule in this scenario might be found via standard quadratic optimisation techniques. However, when the dynamics of the impacts are not known, blindly applying optimisation methods may lead to inefficient trading schedules. We test the DDQL results against those obtained via canonical methods as if the dynamics were known from the beginning. The aim is to test whether the algorithm has learnt the correct parameter specification for both the impacts without prior knowledge of their deterministic dynamics, in a quest to obtain a 'robust' optimal execution schedule in an agnostic way.

Stochastic impacts models. We consider two cases of stochastic temporary and permanent impacts by considering models studied in the literature. The first dynamical model, proposed in Ma et al. [2020], assumes that each of the impact parameters follows a regime switching process between two values. The switching is governed by a Markov chain with a given transition matrix. The closed form solution can be found in Ma et al. [2020] and of course requires the full knowledge of all parameters. On the contrary, in our setting, the RL agent learns the optimal strategy in the training phase.

The second stochastic impact model, proposed by Barger and Lorig [2019] assumes that both the temporary and the permanent impacts follow a square-root mean reverting stochastic process. We choose the parameters in such a way that the impacts are almost surely positive, and we assume a positive correlation between the permanent and the temporary impacts noises. The model in continuous time[2] is:

$$\begin{aligned} dS_t &= -\kappa_t \nu_t dt + \sigma dW_t \\ d\kappa_t &= \lambda_\kappa(\theta_\kappa - \kappa_t)dt + \sigma_\kappa \sqrt{\kappa_t} dB_t^{(1)} \\ \tilde{S}_t &= S_t - \alpha_t \nu_t \\ d\alpha_t &= \lambda_\alpha(\theta_\alpha - \alpha_t)dt + \sigma_\alpha \sqrt{\alpha_t} dB_t^{(2)} \\ <B_t^{(1)}, B_t^{(2)}> &= \omega \end{aligned} \tag{3.5}$$

where, as before, S_t is the stock mid-price, $\tilde{S}_t$ is the execution price, λ_κ and θ_κ are, respectively, the rate of mean reversion and the long run mean at which the permanent impact reverts to, λ_α and θ_α have same meaning but for the temporary impact dynamics.

In Barger and Lorig [2019], the authors derive an *approximate* optimal execution solution

[2]We consider a discrete time version of the model in the subsequent experiments.

obtained via a Taylor approximation around the long run mean values for the processes considered. In other words, the solution is valid when the impact fluctuations are small. In this regime, assuming that $\alpha_0 = \theta_\alpha,\ \kappa_0 = \theta_\kappa$, the trading velocity ν_t at time t, is:

$$\nu_t = \left(\frac{1}{N-t} + \frac{\lambda_\alpha(\theta_\alpha - \alpha_t)}{2\alpha_t} + (N-t)\frac{\lambda_\kappa(\theta_\kappa - \kappa_t)}{6\kappa_t} \right) \times q_t \tag{3.6}$$

It is seen that Equation (3.6) can be seen as a perturbation of the TWAP solution, which is optimal in the constant impact case. In what follows we will consider a discrete-time version of the model. We will assume that the trader trades within each of the N time-steps t, a quantity $v_t = \nu_t \tau$ of stocks. The price and the impacts' paths over the N discrete time-steps are simulated using the model in Equation (3.5).

3.2.2 DDQL algorithm

Deep Reinforcement Learning is a machine learning paradigm where an agent, modelled with neural networks, learns how to interact with an environment. This is achieved by taking actions and receiving feedback in the form of rewards or penalties. The primary objective of the agent is to maximise the cumulative reward over time by discovering an effective policy or strategy.

We denote the sets of all possible states, actions, rewards, and policies as $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$ and Π respectively. In this context, the algorithm, given the environment's state $s_t \in \mathcal{S}$, must decide which action $a_t \in \mathcal{A}$ to perform. This action modifies the environment to a subsequent state $s_{t+1} \in \mathcal{S}$ and results in receiving a reward $r_t \in \mathcal{R}$. The ultimate goal is to learn an optimal policy $\pi^* \in \Pi$ that maximises the sum of discounted future rewards, denoted as $\sum_{t=0}^{N} \gamma^k r_t$, where $\gamma \in (0, 1]$ is the discount factor.

In this study, we employ Deep Reinforcement Learning to find the optimal execution schedule in various time-varying impact settings. Specifically, we adopt the *Double Deep Q-Learning* (DDQL) algorithm, providing a model-agnostic solution to the optimal execution problem. Q-values, representing the quality of taking actions in each state of the environment, are obtained through neural networks known as Q-nets. We use Double Deep Q-learning, thus employing two neural networks, namely the main Q-net (Q_{main}) for action selection and the target Q-net (Q_{tgt}) for state evaluation. Overall, the algorithm undergoes an exploration phase, where it explores possible rewards by taking random actions in evolving states. Subsequently, it gradually transitions to an exploitation phase, leveraging learned information on rewards using

Q_{main} to select actions. To maintain stability, the weights for Q_{tgt} are periodically synchronised with those of Q_{main}.

This is achieved through a training routine that updates the main net using past states, actions, and rewards randomly sampled from a memory. The training occurs over M episodes, followed by a test phase on an additional B episodes to evaluate the DDQL agent's performance post-exploration.

Algorithm setup for numerical experiments

In order to tackle the problem of optimal execution with DDQL, we consider an agent who wants to sell a portfolio of q_0 stocks within a time-window $[0, T]$. During this time window, the agent sells its inventory totally, and this is called an *episode*. The episodes are divided into N time-steps and the time increment for the simulations is $\tau = \frac{T}{N}$. In this context, the agent decides the quantity of stocks v_t to sell in each of the t time-steps in which the episode is divided. The mid-price at which the agent trades is modelled according to Equation (3.1), but depending on the frameworks considered, the dynamics for the impact parameters change.

Over the M train episodes the algorithm learns, via exploration-exploitation, the best trading strategy and changes the weights of the Q-nets accordingly. The architecture of the algorithm in the training phase is thus divided into two phases: an *exploration* and an *exploitation* phase, managed by the parameter $\epsilon \in (0, 1]$ that decreases during training and initially set as $\epsilon = 1$. Depending on the phase, the way actions v_t are chosen changes. When the agent is exploring, it randomly selects sell actions v_t in order to explore different states and rewards in the environment, alternatively, when the agent is exploiting, it uses Q_{main} to select v_t.

Action selection and reward function. At the beginning of each time interval and for each episode, the agent has limited knowledge of the environment where it trades. This means that it has access to states s_t made either by tuples of (q_t, t) or (q_t, t, S_{t-1}) (i.e. quantity, time and possibly mid-price), depending on the considered feature setup.

Given the current value of ϵ, a draw ζ from a uniform distribution determines whether to perform exploration or exploitation. Specifically, with probability ϵ the agent chooses to explore, and the sell action v_t is sampled from a binomial distribution with number of trials equal to q_t (i.e. the inventory left at the beginning of a sub-interval) and probability of success is $\frac{1}{N-t}$. In this way, in the exploration phase the TWAP is selected on average. Alternatively (with

probability $1-\epsilon$), the agent chooses the Q-optimal action, i.e. the action that maximises the Q-value from Q_{main}, doing exploitation of what learnt in the exploration phase. Notice that the agent cannot sell more than the remaining inventory and, in addition, no buy actions can be performed in the selling program.

In this way, the agent randomly explores a large number of states and possible actions to perform. Once every m actions taken during training episodes, ϵ is multiplied by a constant $c < 1$ such that $\epsilon \leftarrow \epsilon \times c$, in this way for large number of episodes M, ϵ converges to zero and the algorithm gradually stops exploring and starts to greedily exploit what it has learned in terms of weights Q_{main}. In formulae, the action decision rule for each time step t in the training phase is:

$$\begin{aligned} &\epsilon \in (0,1)\ ,\ \zeta \sim \mathcal{U}(0,1) \\ &v_t = \begin{cases} \sim \text{Bin}(q_t, \frac{1}{N-t}) & \text{, if } \zeta \leq \epsilon \\ \arg\max_{v' \in [0,q_t]} Q_M(s_t, v' | \theta_{\text{main}}) & \text{, else} \end{cases} \end{aligned} \tag{3.7}$$

At each t, once that the action is chosen according to Equation (3.7), the reward in the case of a risk neutral agent is:

$$r_t = S_{t-1} v_t - \alpha v_t^2 \tag{3.8}$$

Overall, the rewards for every $t \in [1, N]$ are:

$$\begin{aligned} r_1 &= S_0 v_1 - \alpha v_1^2 \\ r_2 &= (S_1 + \kappa v_1 + \mathcal{N}(0,\sigma)) v_2 - \alpha v_2^2 \\ &\vdots \\ r_N &= (S_{N-1} + \kappa v_{N-1} + \mathcal{N}(0,\sigma)) v_N - \alpha v_N^2 \end{aligned} \tag{3.9}$$

Thus, for each time step t the agent sees the reward from selling v_t shares, and stores the state of the environment, s_t, where the sell action was chosen along with the reward and the next state s_{t+1} where the environment evolves. As said, the aim of the agent is to cumulatively *minimise* such reward, such that the liquidation value of the inventory happens as close as possible to the initial value of the portfolio. At the end of the episode, the reward per episode is $-q_0 S_0 + \sum_{t=1}^{N} S_{t-1} v_t - \alpha v_t^2$.

When considering a risk averse agent, we modify the reward function following Kolm and

Ritter [2019], and the reward function of Equation (3.8) becomes:

$$r_t = S_{t-1}v_t - \alpha v_t^2 + \frac{\lambda}{2}\left(v_t^2 + \sigma^2\right) \tag{3.10}$$

where λ is the risk aversion parameter.

Training scheme. The states s_t, actions v_t, rewards r_t and subsequent future states s_{t+1} obtained by selling a quantity v_t in state s_t, form a *transition* tuple that is stored into a memory of maximum length L. As soon as the memory contains at least b transitions, the algorithm starts to train the Q-nets. In order to train the nets, the algorithm samples random batches of length b from the memory, and for each sampled transition j it calculates:

$$y_t^j(\theta_{\text{tgt}}) = \begin{cases} r_t^j & \text{if } t = N; \\ r_t^j + \gamma Q_{\text{tgt}}(s_{t+1}^j, v^*|\theta_{\text{tgt}}) & \text{else} \end{cases} \tag{3.11}$$

In Equation (3.11), r_t^j is the reward for the time step considered in the transition j, s_{t+1}^j is the subsequent state reached in $t+1$, known since it is stored in the same transition j, while $v^* = \arg\max_v Q_{\text{main}}(s_t^j, v|\theta_{\text{main}})$. γ is a discount factor in this context. The loss to be minimised is the mean squared error between the target $y(\theta_{\text{tgt}})$ and the values obtained via the Q-main network. Thus, it is:

$$\mathcal{L}(\theta_{\text{main}}, \theta_{\text{tgt}}) = \frac{1}{b}\sum_{\ell=1}^{b}\left(\left[y_t^\ell(\theta_{\text{tgt}}) - Q_{\text{main}}(s_t^\ell, v_t^\ell|\theta_{\text{main}})\right]\right)^2$$

$$\theta_{\text{main}}^* = \arg\min_{\theta_{\text{main}}} \mathcal{L}(\theta_{\text{main}}, \theta_{\text{tgt}})$$

The algorithm then uses back propagation and gradient descent to update the weights of the main network. This sub-routine is repeated for each random batch of transitions sampled from the memory. Finally, as stated above, once every m actions ϵ is decreased by a factor $c < 1$ and the algorithm sets $Q_{\text{tgt}} = Q_{\text{main}}$.

After training, other $B < M$ trading episodes are fed to the algorithm in order to exploit what learnt during training, this is the testing phase and the actions are now selected using just Q_{main}. The results shown below are those obtained in the test phase only.

The features of the Q-nets are (q_t, t, v_t) or (q_t, t, S_{t-1}, v_t), we will refer to the first case as the features Q,T case and and to the second as the features Q,T,S case, the features are normalised

in the domain $[-1, 1]$ using the procedure suggested in Ning et al. [2021], normalised mid-prices $\bar{S}_t$ are obtained via min-max normalisation. In our setup we use fully connected feed-forward neural networks with 5 layers, each with 30 hidden nodes activation functions are leakyReLu, and finally we use the ADAM optimiser for optimisation. The parameters used to calibrate the algorithm are reported in Table 3.1, the training algorithm is reported in Algorithm 5.

Table 3.1: Fixed parameters used in the DDQL algorithm. The parameters not shown in the table change depending on the experiments and are reported accordingly.

DDQL parameters				**Model parameters**	
NN layers	5	M train eps.	$10,000$	N intervals	10
Hidden nodes	30	B test eps.	$5,000$	σ-stock	0.00001
ADAM lr	0.0001	m reset rate	100 acts.	S_0-stock	10$
Batch size b	32	c decay rate	0.995	q_0 inventory	20
L memory length	$15,000$	γ-discount	1		

Algorithm 5 Training of Double Deep Q-Learning agent for optimal execution

Require:
Initialise with random weights Q_{main} and make a copy Q_{tgt};
Initialise the memory with max length L. Set $\epsilon = 1$, b batch size, M train episodes;
Set market dynamics parameters;

for i in M **do**
Set $S_0^i = S_0$;

for t in N **do**
$s_t \leftarrow (q_t, t)$; ▷ or (q_t, t, S_{t-1}^i) if mid-price considered
$v_t \leftarrow \begin{cases} \text{sample Bin}(q_t, \frac{1}{N-t}) \text{ with probability } \epsilon \\ \arg\max_{v' \in [0, q_t]} Q_{\text{main}}(s_t, v' | \theta_{\text{main}}) \text{ with probability } (1-\epsilon) \end{cases}$;
$r_t \leftarrow S_{t-1}^i v_t - \alpha v_t^2$;

$S_{t-1}^i \to S_t^i$; ▷ Generate S_t^i from S_{t-1}^i

$s_{t+1} \leftarrow (q_{t+1}, t+1)$; ▷ or $(q_{t+1}, t+1, S_t^i)$

Memory $\leftarrow (s_t, r_t, v_t, s_{t+1})$; ▷ Memory storing

if Length of memory $\geq b$ **then**
for j in b **do**
Sample a batch of $(s_t^j, r_t^j, v_t^j, s_{t+1}^j)$ from memory;
$v^* = \arg\max_v Q_{\text{main}}(s_t^j, v | \theta_{\text{main}})$;
$y_t^j(\theta_{\text{tgt}}) = \begin{cases} r_t^j & \text{if } t = N; \\ r_t^j + \gamma Q_{\text{tgt}}(s_{t+1}^j, v^* | \theta_{\text{tgt}}) & \text{else} \end{cases}$
end for

$\theta^*_{\text{main}} = \arg\min_{\theta_{\text{main}}} \mathcal{L}(\theta_{\text{main}}, \theta_{\text{tgt}})$ via gradient descent with loss to minimise:

$$\mathcal{L}(\theta_{\text{main}}, \theta_{\text{tgt}}) = \frac{1}{b} \sum_{\ell=1}^{b} \left(\left[y_t^\ell(\theta_{\text{tgt}}) - Q_{\text{main}}(s_t^\ell, v_t^\ell | \theta_{\text{main}}) \right] \right)^2$$

if Length of memory $= L$ **then** halve the length of memory
end if
end if
After m iterations decay $\epsilon = \epsilon \times c$;
After m iterations $\theta_{\text{tgt}} \leftarrow \theta_{\text{main}}$;
end for
end for

3.3 Experiments and results

We now present the results of our numerical investigations. The aim of the experiments is to study whether the DDQL agent is able to find model-robust optimal execution strategies without any form of knowledge of the underlying impact model, but just based on simple sets of information. Since the algorithm does not know neither the parameters nor the model used for the dynamics of the impacts, we adopt a model agnostic approach to find optimal execution strategies. In this context, we study whether the DDQL agent is able to learn the optimal strategy (i) when the optimal solution is known but the environment is not known by the agent and (ii) when the optimal solution is not known in closed form but in an approximate form and the environment is still not known to the agent. In this sense, we investigate the use DDQL as a tool to develop 'robust' strategies. In a way, the goal is to use DDQL as a robust non-parametric technique, able to agnostically detect price impacts' magnitude and dynamics in the market by using as few information as possible.

Notice that the number of trading steps per episode is set to $N = 10$ and the agent has to to sell $q_0 = 20$ shares. Notice that the chosen number of shares is not a limitation, since, due to the linearity of the impact model, the same optimal strategy can be used for an arbitrary value of q_0 simply multiplying the trading velocity obtained by RL with 20 shares by $q_0/20$. After the test phase is completed, the average IS is calculated for each test episode. Thus, in the test phase we measure for each episode the IS for both strategies, namely the DDQN strategy and the optimal one if impacts dynamics and parameters were known from the beginning. The performance difference between the two approaches, for each test episode, is quantified as:

$$\Delta P\&L = \frac{\mathcal{C}_{\text{agent}} - \mathcal{C}_{\text{b}}}{\mathcal{C}_{\text{b}}}$$

where $\mathcal{C} = \sum_{t=1}^{N} v_t \tilde{S}_t$, the cash generated by the trading. Notice that $\mathcal{C}_{\text{b}}$ is the cash calculated for the baseline model, while $\mathcal{C}_{\text{agent}}$ is the cash obtained using DDQL strategies. In the tables below, we report the average values and the standard deviation over the whole test set.

It is interesting to remind that when impact is constant or deterministically time varying, the statically optimal strategy is also dynamically optimal, as shown by Schied et al. [2010]. This implies that the knowledge of the price at time t should not be relevant in determining the next optimal action. While this is true in a theoretical setting where all the parameters are known, this might (and is, as shown below) be different in RL setting. In this case a feature that

is theoretically not relevant for the optimal strategy becomes important in reducing the cost, especially when the level of noise/volatility is very high. In fact, adding a non-theoretically relevant feature in this setup, contributes to lowering the costs by reducing the demand for extensive training possibly required by the agent whenever the noise in the environment increases or the dynamics get less trivial, as is the case for time varying impacts. In other words, the noisier (or the more complex) the environment, the more difficult for the agent is to find the optimal strategy with a low number of training episodes, thus requiring longer training or more features. We tackle the point in more details in Appendix A.

3.3.1 Constant impact

Risk neutral agent

We first consider the mid-price dynamics generated according to the A&C model of Equation (3.1) with constant impact coefficients equal to $\kappa = 0.001$ and $\alpha = 0.002$. In this setting, the TWAP is the optimal solution, i.e. the strategy with minimal costs is the one where q_0 is sold equally over the time steps. We want to check that the DDQL is able to recover a similar strategy also by comparing the expected cost with that of a TWAP. This would imply that DDQL understands that the impact parameters are fixed and trades accordingly with minimal discrepancy compared to the closed-form solution's costs. Moreover, we expect that including price as feature should not improve the DDQL strategy performance.

Features Q,T The results are reported in Table 3.2. We notice that, when the agent has knowledge of just q_t and t, it retrieves an IS value very similar to the one of the TWAP strategy. The $\Delta P\&L$ shows a difference of just -0.4 basis points and a standard deviation of only two basis points. The average number of stocks sold per time-step and level of inventory is shown in the left panel of Figure 3.1. It can be observed that the agent has learned a strategy consistent with 'slice and dice' trading.

Features Q,T,S When one includes also the price among the features, we observe that the DDQL agent is able to retrieve the same cost. The $\Delta P\&L$ is close to -0.25 basis points, implying that the optimal strategy has been learnt by the agent. The average optimal trading per level of inventory q_t, time-step t, and price level S are reported in the right panel of Figure3.1. The algorithm overall does not perform any better than a noisy TWAP strategy, as expected.

Moreover, it appears that allowing for more information in such a simple environment does not add any significant value. The strategy adjusts depending on the price level considered to be optimal and does not outperform the previous case. This aligns with the theoretical findings in Schied et al. [2010], proving that dynamical strategies (which can depend on price) are also statically optimal.

Table 3.2: Constant impact. Comparison between the costs of the DDQL agent and of the A&C model. ΔP&L values are reported in basis points. The standard errors of the nominal raw differences between TWAP and RL cost of the strategies are in parenthesis.

Constant Impact				
features	**A&C** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	ΔP&L **avg.**	ΔP&L **std.dev.**
Q,T	0.2607	0.2698 (0.00071)	-0.455	2.5
Q,T,S	0.2607	0.2652 (0.00046)	-0.225	1.6

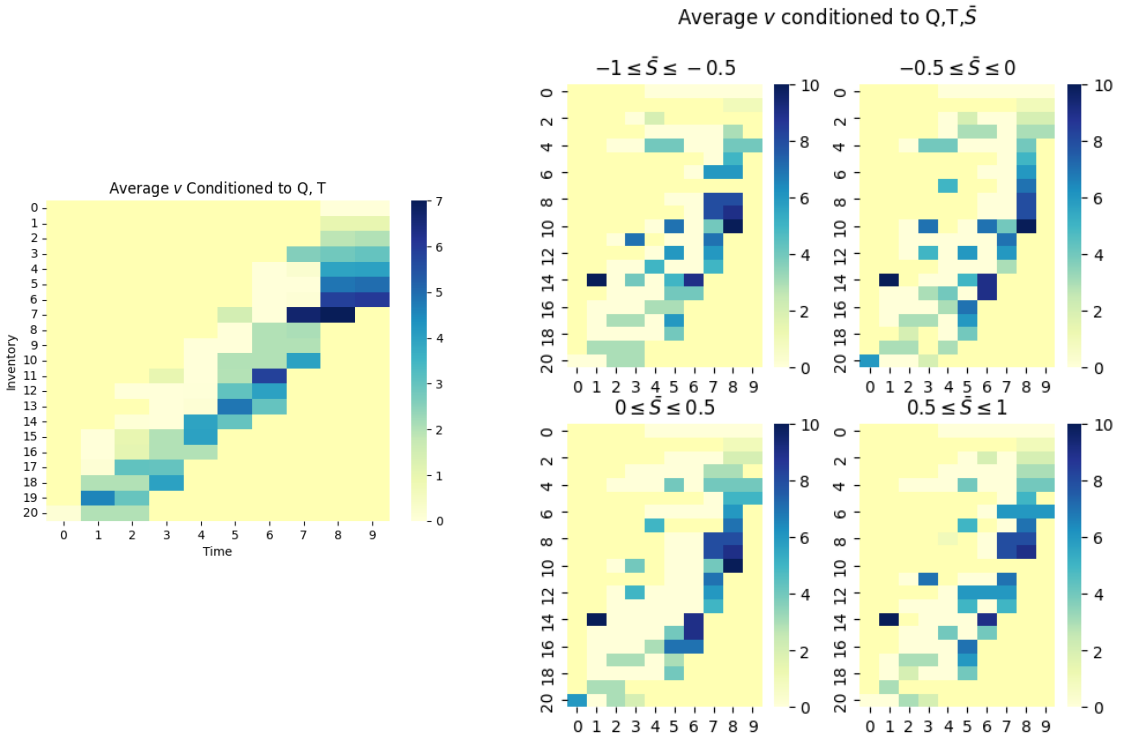


Figure 3.1: Constant impact. Left panel: Average number of stocks sold per time-step t and inventory level q_t. Right panel: Average number of stocks sold per time-step t, inventory level q_t, and normalised price $\bar{S}$.

The role of volatility. The capability of DDQL to recover optimal strategies might depend on the level of noise, in this case the volatility, of the price process. We assess the role of the

volatility parameter on the performances of DDQL. Increasing the volatility parameter with respect to the base case used above to $\sigma = 0.001$, we observe that the results of the DDQL strategy remain satisfactory. In fact, the average costs achieved by using either of the features are relativity near the A&C IS value, see Table 3.3a. As expected, whenever the information set available to the agent in terms of features includes also the mid-price as well, the DDQL is able to achieve a $\Delta P\&L$, that is still negative, but smaller than one basis point. Furthermore, comparing these results to the case where the agent uses q_t and t as features, we see that the $\Delta P\&L$ is nearly halved, outlining the importance of adding the mid-price whenever the volatility parameter increases.

Table 3.3: Comparison between the costs of the DDQL agent and the A&C model under different volatility levels. ΔP&L values are reported in basis points. Standard errors of the nominal raw differences between the cost of the strategies are in parenthesis.

(a) Constant impact and $\sigma = 0.001$.

features	**A&C** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	ΔP&L **avg.**	ΔP&L **std.dev.**
Q,T	0.2607	0.2905 (0.00089)	-1.49	2.45
Q,T,S	0.2607	0.2763 (0.00081)	-0.778	2.22

(b) Constant impact and $\sigma = 0.1$.

features	**A&C** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	ΔP&L **avg.**	ΔP&L **std.dev.**
Q,T	0.2607	0.3893 (0.094)	-6.43	259.97
Q,T,S	0.2607	0.3488 (0.061)	-4.41	168.9

(c) Constant impact and $\sigma = 0.1$ with 20,000 training episodes.

features	**A&C** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	ΔP&L **avg.**	ΔP&L **std.dev.**
Q,T	0.2607	0.2604 (0.055)	0.38	279.31

Increasing the volatility parameter to the very large value $\sigma = 0.1$, we observe that for both sets of features the agent performs poorly in terms of costs, as reported in Table 3.3b. We hypothesize that for very high levels of volatility more training is required, as the current amount of episodes, along with the features used are proved to be insufficient to deal with the higher level of volatility used. We further discuss this in more details in Appendix A. Investigating on this issue, and keeping in mind that in this case the optimal strategy does not depend on the volatility level, we trained the agent on more episodes, adjusting the reset rate m accordingly to 200 actions for $20,000$ training episodes. This results in an average IS value in line with the theoretical one, as reported in Table 3.3c with just Q,T used as features. This result is indeed

in line with the fact that optimal strategies, and therefore costs, do not depend on the level of volatility. In addition, it supports the idea that for extremely noisy environments more training is required, as the agent seems to be sensible to returns oscillations during the training phase. The low value of $\Delta P\&L$ confirms that the agent is able to recover costs that, even with a simple set of features, almost overlap with the theoretical ones if the number of training episodes is sufficient.

Risk averse agent

We now consider the case of a risk-averse agent trading in an environment with constant impact coefficients. Using the reward in Equation (3.10) with $\sigma = 0.01$ and $\lambda = 5$, we compare the DDQL optimal strategy with the one in Equation (3.3) to test whether DDQL is able to learn the optimal strategy when the agent is risk averse. The results are reported in Table 3.4. The strategies found by the agent along with the inventory levels are shown in Figure 3.2 and Figure 3.3.

Features Q, T When the agent uses only q_t and t as features, it performs poorly in terms of average IS. In fact, the $\Delta P\&L$ is -9.25 basis points. This shows that the agent has been unable to recover the correct risk averse strategy.

Features Q, T, S When the set of features also includes the price, the agent is instead able to recover a strategy more similar to the exact solution in A&C. In fact, as shown in Table 3.4, the IS is almost identical to the theoretical one and the $\Delta P\&L$ is very small, being equal to -0.062 bps. Also in this case, the price S, which is theoretically unnecessary for the optimal strategy, becomes useful when the number of training episodes is small and the noise is large.

Table 3.4: Constant impact and $\lambda = 5, \quad \sigma = 0.01$. Comparison between the costs of the DDQL agent and of the A&C model. ΔP&L values are reported in basis points. The standard errors of the nominal raw differences between the cost of the strategies are in parenthesis.

Constant Impact and risk aversion				
	A&C	**DDQL**	ΔP&L	ΔP&L
features	$\mathbb{E}[IS]$	$\mathbb{E}[IS]$	**avg.**	**std.dev.**
Q,T	0.2824	0.4676 (0.0015)	-9.25	11.25
Q,T,S	0.2824	0.2840 (0.00016)	-0.062	10.65

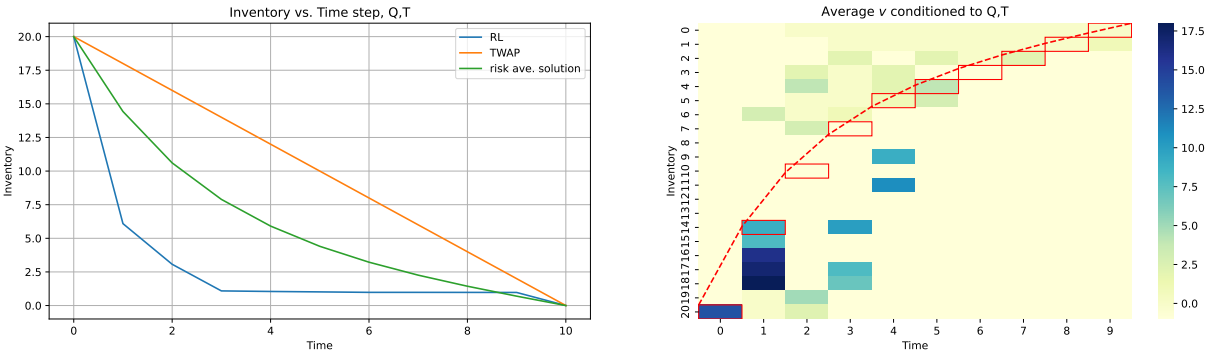


Figure 3.2: Constant impacts and risk averse agent. Trading strategies when DDQL uses Q, T as features. Left: average inventory holdings per time step. Right: average number of stocks sold per time-step t and inventory level q_t; in red the expected trajectory if the agent was to use the theoretical solution.

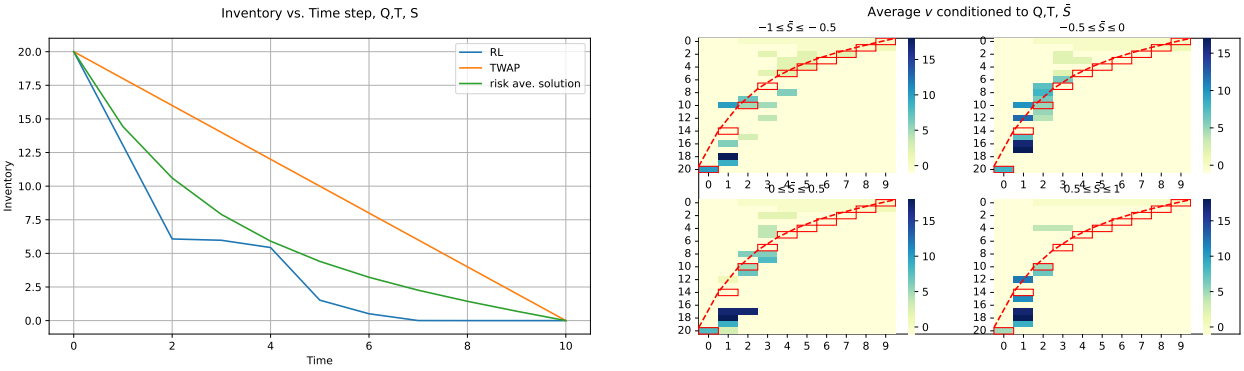


Figure 3.3: Constant impacts and risk averse agent. Trading strategies when DDQL uses Q, T, S as features. Left: average inventory holdings per time step. Right: average number of stocks sold per time-step t, inventory level q_t and normalised price $\bar{S}$; in red the expected trajectory when the agent uses the theoretical solution.

3.3.2 Deterministically time-varying impact

Having established that the DDQL agent is able to recover the optimal solution when impact coefficients are constant, we test the algorithm with two different, yet deterministic, impacts patterns. It is widely recognised that liquidity has intraday patterns[3] thus a trend in impact parameter is observed in certain parts of the trading day, e.g. in the first part of the trading day or around important market announcements. If one knows in advance the functional form of the dynamics of the impact parameters, the optimal execution is again a minimisation of a cost functional and can be solved via quadratic optimisation (although the well posedness of the problem is far from trivial, see Palmari et al. [2024]). However, the DDQL agent has no knowledge about the existence of a pattern and whether there is an increasing or decreasing

[3] see, for example, Campigli et al. [2022] for a thorough discussion on this topic.

trend. In other words, the impact parameters are implicitly estimated by the DDQL agent.

As explained in Section 3.2.1, we assume impacts follow a linear deterministic trend. We consider either an increasing or a decreasing impact, and we compare the found solutions with the optimal strategy obtained by using quadratic optimisation[4]. The resulting IS, obtained by simulating the optimal strategy, is then compared with the one obtained with the DDQL strategy. We also use, as as an additional benchmark, the IS of a TWAP strategy, which is obviously not optimal in this context.

Table 3.5: Parameters used in the time-varying settings.

(a) Increasing time-varying setting.

parameters	value
κ_0	0.0001
α_0	0.0001
β_κ	0.0002
β_α	0.0004
$\kappa_5 = \kappa$	0.001
$\alpha_5 = \alpha$	0.002

(b) Decreasing time-varying setting.

parameters	value
κ_0	0.002
α_0	0.004
β_κ	0.0002
β_α	0.0004
$\kappa_5 = \kappa$	0.001
$\alpha_5 = \alpha$	0.002

Testing and training with increasing impacts

The impacts dynamics is $g(\frac{v_t}{\tau}) = \kappa_t \frac{v_t}{\tau}$, where $\kappa_t = \kappa_0 + \beta_\kappa \times t$ and $h(v_t) = \alpha_t v_t$, where $\alpha_t = \alpha_0 + \beta_\alpha \times t$ and the used parameters are reported in Table 3.5a. The parameters are chosen such that the average value of the impact parameter during the trading period are equal to the fixed impacts in the experiments of the previous section.Figure 3.4 shows the found solutions with the two sets of features, together with the theoretically optimal solution and the TWAP. As expected, given that impacts are increasing, it is better to trade more at the beginning to exploit the initial lower trading costs.

Features Q,T As shown in Table 3.6, when only inventory and time are used as features, the DDQL algorithm is not able to really beat the TWAP strategy in terms of costs. In fact, the $\Delta P\&L$ between the DDQL and the TWAP strategy is very small. As a consequence, the $\Delta P\&L$ between the DDQL and the theoretical optimal cost is significantly negative. Looking at the blue line in Figure 3.4 it appears that the DDQL agent with two features learns to trade more at the beginning, but not enough. A similar conclusion is obtained by looking at the full strategy (see left panel of Figure 3.5).

[4]We solved the problem using the **.optimise()** method in SciPy and applying Quadratic programming to Equation (3.4).

Features Q,T,S Adding the mid-price to the features leads to better results in terms of $\Delta P\&L$. As reported in Table 3.6, the IS of the DDQL agent is significantly smaller than the one obtained with TWAP. Comparing our results with the theoretical solution, we conclude that adding the mid-price to the feature set lowers the $\Delta P\&L$ to just 2 basis points of difference. Although not performing as good as the theoretical solution, adding a feature significantly increases the learning capabilities of the algorithm. This conclusion is supported by the average inventory holding and average trading schedule reported in Figure 3.4 and right panel of Figure 3.5, respectively. It can be seen that the agent has learned to trade too aggressively within the first half of the execution window. This is compatible with the increasing nature of the underlying impact parameters.

	Increasing Impact				
inputs	**Theo.** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	$\Delta P\&L$ vs. Theo. **avg.**	$\Delta P\&L$ vs. Theo. **std.dev.**	**TWAP** $\mathbb{E}[IS]$
Q,T	0.1449	0.2401	-4.76	1.17	0.2326
Q,T,S	0.1449	0.1944	-2.42	1.10	0.2326

Table 3.6: Increasing impact. Comparison between the costs of the DDQL agent, of the theoretically optimal solution, and of a TWAP strategy. ΔP&L values are reported in basis points.

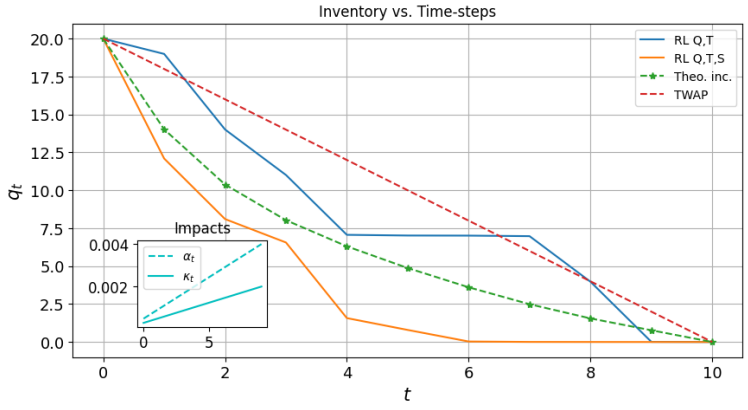


Figure 3.4: Increasing impact. Average inventory holdings per time step with different features, compared to the optimal inventory holding if the strategy was known from the beginning, and to the inventory holding if the TWAP strategy was used. In the inset graph we have reported the time-dependent values for the impacts.

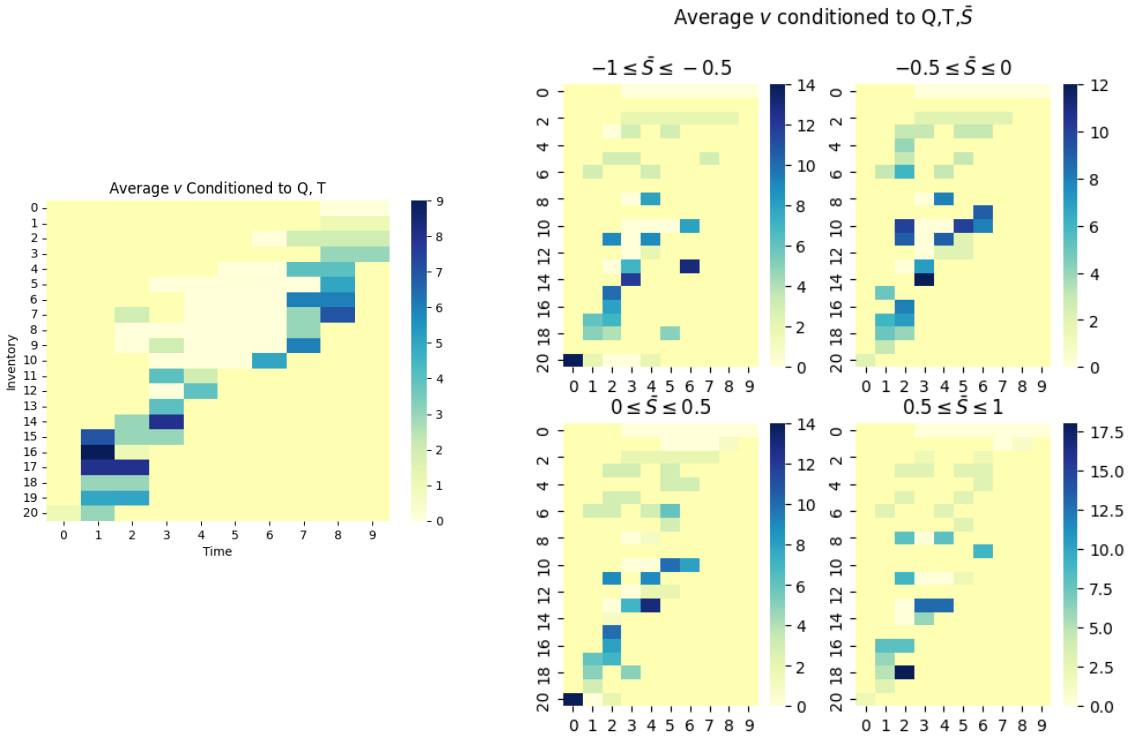


Figure 3.5: Increasing impact. Left panel: Average number of shares sold per time-step t and inventory level q_t. Right panel: Average number of shares sold per time-step t, inventory level q_t and normalised price $\bar{S}$

Testing and training with decreasing impacts

The impacts dynamics is $g(\frac{v_t}{\tau}) = \kappa_t \frac{v_t}{\tau}$, where $\kappa_t = \kappa_0 - \beta_\kappa \times t$ and $h(v_t) = \alpha_t v_t$, where $\alpha_t = \alpha_0 - \beta_\alpha \times t$ and the parameters for the impacts are reported in Table 3.5b. As before, the parameters are chosen such that the average value of the impact parameter during the trading period are equal to the fixed impacts in the experiments of the previous section. Figure 3.6 shows the found solution for the two set of features and the theoretically optimal trading schedule. Since impacts are decreasing, it is optimal to trade less than in a TWAP at the beginning to avoid the initial high trading costs.

Features Q,T Similarly to the increasing impact case, when the features are only inventory and time, the DDQL algorithm has a sub-optimal performance. In fact, as reported in Table 3.7, the IS is larger than the theoretical optimum, even if the learnt strategy outperforms the TWAP. As in the previous case, the DDQL algorithm manages to understand the direction of the trend. In fact, Figure 3.6 (blue line) and the left panel of Figure 3.7 show that the average actions per inventory level and time-step are consistent with increasing impact parameters, since the algorithm trades more toward the end of the period when costs are lower. It can be concluded

that although correct in the estimation of the trend, the algorithm did not manage to calibrate its own actions to the magnitude of the impacts.

Features Q,T,S When including the mid-price among the features, the algorithm's performance gets significantly better, (see Table 3.7) having a cost slightly worse than the optimal trading strategy. Again, adding the mid-price as a feature adds more power to the DDQL agent when it comes of estimating magnitude and direction of the impacts. This is confirmed by the average actions chosen per level of inventory, time step and mid-price level S, see Figure 3.6 (orange line) and the right panel of Figure 3.7. The strategy is again skewed to the right hand side of the heat-map implying that the algorithm has understood the direction and magnitude of the impacts.

	Decreasing impact				
inputs	**Theo.** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	$\Delta P\&L$ vs. Theo. **avg.**	$\Delta P\&L$ vs. Theo. **std.dev.**	**TWAP** $\mathbb{E}[IS]$
Q,T	0.2566	0.3080	-2.58	1.17	0.3588
Q,T,S	0.2566	0.2877	-1.51	0.52	0.3588

Table 3.7: Decreasing impact. Comparison between the costs of the DDQL agent, of the theoretical solutions, and of a TWAP strategy. ΔP&L values are reported in basis points.

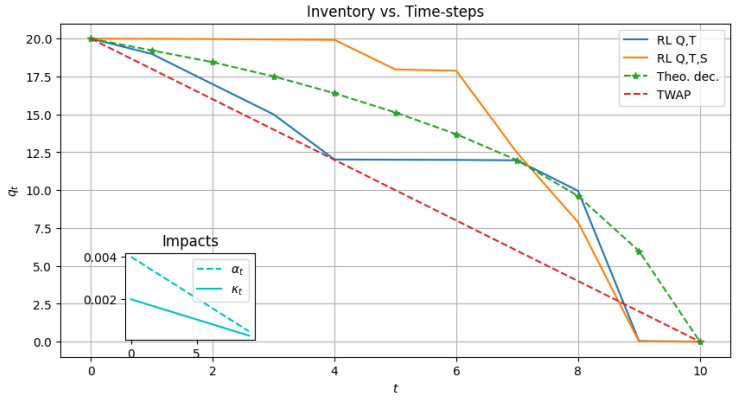


Figure 3.6: Average inventory holdings per time step with different features, compared to the optimal inventory holding if the strategy was known from the beginning, and to the inventory holding if the TWAP strategy was used. In the inset graph we have reported the time-dependent values for the impacts.

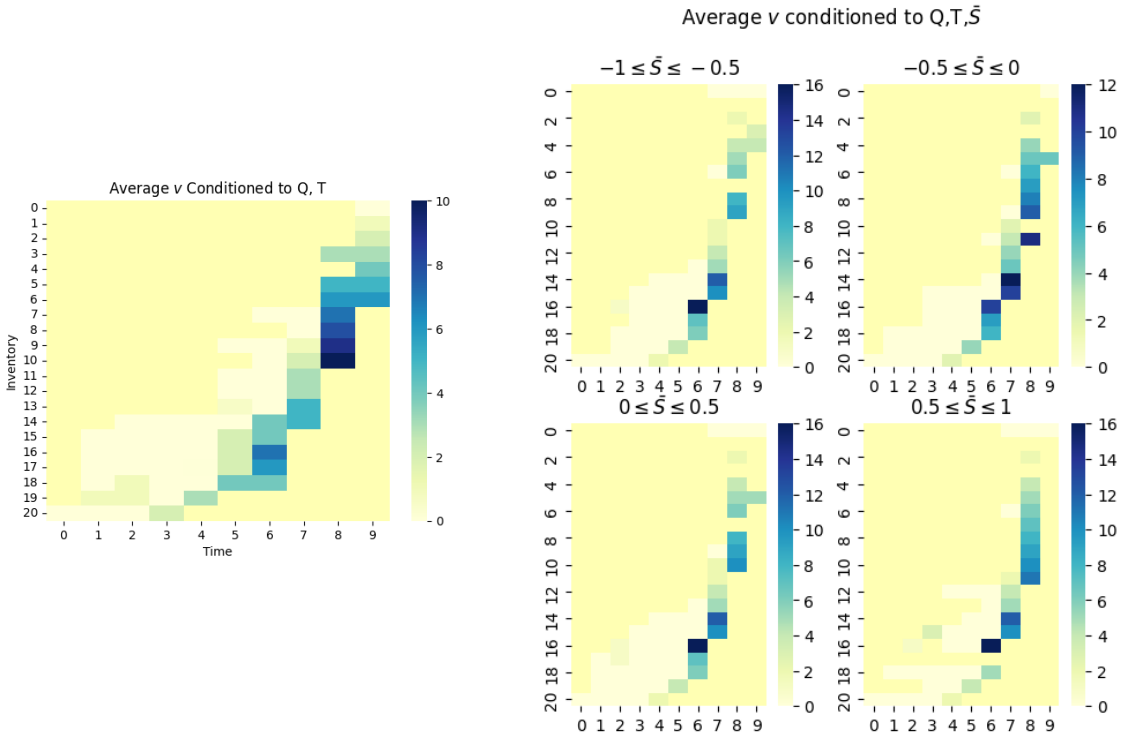


Figure 3.7: Decreasing impact. Left panel: Average number of shares sold per time-step t and inventory level q_t. Right panel: Average number of shares sold per time-step t, inventory level q_t and normalised price $\bar{S}$

Mixed train with increasing and decreasing impacts

While the previous results were obtained by using the same dynamics in the train and test phase, here we consider a more realistic situation where the train phase contains different scenarios and in the test phase only one of them is actually realised. More specifically, the DDQL agent is now trained with $10,000$ trajectories using increasing impacts and $10,000$ trajectories using decreasing impacts. It is subsequently tested on $5,000$ trajectories with either increasing or decreasing impacts trends. It is worth to study such experimental setup since it is functional for an eventual deploy in real market situations of an algorithm based on this RL paradigm. In fact, if the algorithm is used in a real trading setting, and especially without any prior knowledge of the data at hand, one would train such an algorithm with several scenarios for the stocks intended to be sold and then, using such strategy, one would like the algorithm to be able to detect the specific liquidity pattern and simultaneously adapt the trading strategy. We mimic this situation by training with both regimes, then we test with either of the two, in order to assess whether the DDQL agent is able to recognise the direction and the magnitude of the impact coefficients.

In this case the TWAP becomes the natural benchmark. In fact, since the training set includes both increasing and decreasing impact while the in the testing is present only one of them, the obvious choice for a agnostic trader would be to use a TWAP strategy. For this reason, in the tables we report also the $\Delta P\&L_{\ TWAP}$ with respect to the TWAP strategy. The standard $\Delta P\&L$ is also reported but should be considered as a kind of unfeasible benchmark, since its calculation requires to know whether the impacts in the test set is increasing or decreasing.

Increasing impacts - Features Q,T Using as features only inventory and time, the agent is again unable to find the optimal strategy. As reported in Table 3.8 the performance in terms of $\Delta P\&L$ is still poor and the overall IS is similar to the one obtained with a TWAP strategy. As it can be seen in the left panel of Figure A.3 the strategy achieved by the agent does not appear in line with the one that would be expected with such impact dynamics.

Increasing impacts - Features Q,T,S When the mid-price is included into the features, the performance of the DDQL agent is dramatically better. In fact, as reported in Table 3.8 the performance in terms of $\Delta P\&L$ is almost zero, meaning that the agent performs as good as the optimal solution in terms of costs. In turn, this implies that the agent has correctly estimated both the direction and the magnitude of the impacts used in the test phase. In fact, as it can be seen in the right panel of Figure A.3, the strategy found, conditioned to the price levels, is in line with the increasing magnitude in the impact dynamics.

Table 3.8: Increasing impact. Comparison between DDQL agent, theoretical solutions and TWAP strategy. ΔP&L values are reported in basis points. The standard errors of the nominal raw differences between theoretical and RL cost of the strategies are in parenthesis.

	Mixed training and increasing impact testing					
inputs	**Theo.** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	$\Delta P\&L$ vs. Theo. **avg.**	$\Delta P\&L$ vs. Theo. **std.dev.**	**TWAP** $\mathbb{E}[IS]$	$\Delta P\&L$ **TWAP avg.**
Q,T	0.1449	0.2554 (0.00062)	-5.34	1.1	0.2326	-0.92
Q,T,S	0.1449	0.1319 (0.00048)	0.65	0.9	0.2326	5.2

Decreasing impacts - Features Q,T If just the inventory and the time-step form the state seen by the agent, the estimation of the parameters is noisy as in the other cases. Looking at Table 3.9 it can be seen that the resulting strategy performs slightly worse than the TWAP and

it under-performs the optimal selling schedule suggested via optimisation of the cost functional. This result is in line with the strategy outlined in the left panel of Figure A.4.

Decreasing impacts - Features Q,T,S If the mid-price features the states seen by the agent the performance is fairly good if compared to the previous case. In fact, adding the price to the features of the algorithm increases the estimation power of DDQL and delivers a performance in line with the optimal one. This can be seen in Table 3.9 where the $\Delta P\&L$ is very low, suggesting that the impacts dynamics are estimated quite efficiently leading to a strategy consistent with the decreasing nature of the impacts' dynamics in the test set. This result is confirmed by the average action chosen per level of normalised mid-price, inventory and time-step shown in the right panel of Figure A.4.

Table 3.9: Decreasing impact. Comparison between DDQL agent, theoretical solutions and TWAP strategy. ΔP&L values are reported in basis points. The standard errors of the nominal raw differences between theoretical and RL strategies are in parenthesis.

	Mixed training and decreasing impact testing					
inputs	**Theo.** $\mathbb{E}[IS]$	**DDQL** $\mathbb{E}[IS]$	$\Delta P\&L$ vs. Theo. **avg.**	$\Delta P\&L$ vs. Theo. **std.dev.**	**TWAP** $\mathbb{E}[IS]$	$\Delta P\&L$ **TWAP avg.**
Q,T	0.2566	0.3696 (0.00030)	-5.62	0.32	0.3588	-0.51
Q,T,S	0.2566	0.2456 (0.00051)	0.86	0.03	0.3588	6.5

Although the results above refer to a risk neutral agent, a similar behaviour is observed for a risk averse agent. We considered an experiment with mixed training of a risk-averse DDQL and then we tested the performance against a risk-averse A&C strategy in a decreasing impact scenario (similar results are obtained for testing in an increasing impact scenario). We present the outcomes of an experiment where q_t, t and S are used as features, since we showed that this is the most effective set of features for high level of volatility. Setting $\lambda = 5$ and $\sigma = 0.01$, as above, we obtain the results displayed in Table 3.10. As in the risk neutral case, we notice that also for a risk averse agent the training in mixed pattern conditions provides a policy which is able to dynamically identify the impact pattern in the testing set and to outperform the agnostic TWAP strategy obtaining a $\Delta P\&L = 5.40$ bps

Table 3.10: Decreasing and $\lambda = 5, \quad \sigma = 0.01$. Comparison between the costs of the DDQL agent and of the A&C model is the impacts are decreasing in time. ΔP&L values are reported in basis points. The standard errors of the nominal raw differences between TWAP and RL cost of the strategies are in parenthesis.

Mixed training and decreasing impact with risk aversion testing				
	A&C	**DDQL**	ΔP&L	ΔP&L
features	$\mathbb{E}[IS]$	$\mathbb{E}[IS]$	**avg.**	**std.dev.**
Q,T,S	0.5314	0.42385 (0.0076)	5.40	22.11

3.3.3 Stochastic impacts

Lastly, we consider stochastic impact coefficients. These experiments allow us to consider much more complex dynamics and give the opportunity to thoroughly test the capabilities of the DDQL agent to identify optimal strategies. We will consider two different experiments. The first, leveraging on the contribution by Ma et al. [2020], will train and test the DDQL agent in an environment where both the permanent and temporary coefficient follow a regime switching dynamics. In this first case there is a closed form solution, found by Ma et al. [2020], to the optimal execution problem, and thus the benchmark is the cost that would be achieved if the agent, knowing the model parameters, follows this optimal strategy. In the second experiment, the permanent and temporary impact coefficients follow a mean reverting stochastic processes, as proposed in Barger and Lorig [2019]. As discussed in Section 3.2.1, an approximate solution exists for small impact parameter fluctuations. We aim at assessing whether the DDQL agent is able to recover this proposed approximate solution and, when the impact fluctuations are large, to improve in terms of costs, thus implying that the DDQL has indeed found a better approximation to the optimal trading strategy.

Stochastic regime switching impacts

The model of Ma et al. [2020] considers a A&C setting where the temporary and permanent impact coefficient α_t and κ_t follow two independent Markov chains with two regimes corresponding to high and low impact. We choose $\kappa^L = 0.001$ and $\kappa^H = 0.005$ for the permanent impact, and $\alpha^L = 0.002$ and $\alpha^H = 0.006$ for the temporary impact and the transition matrix of the chain is: $A = \begin{bmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{bmatrix}$ For the sake of conciseness we report results obtained using as features the inventory level q_t, the time-step t and the price S. The average trading strategy is reported in Figure 3.8. The strategy is an average across the testing set, since the optimal action strongly depends on price. It is interesting to note that the optimal solution is front loaded, even if the

agent is risk neutral. The average DDQL strategy is quite close to the theoretically optimal one obtained by using the true values of the model parameters. We also report the cost results in

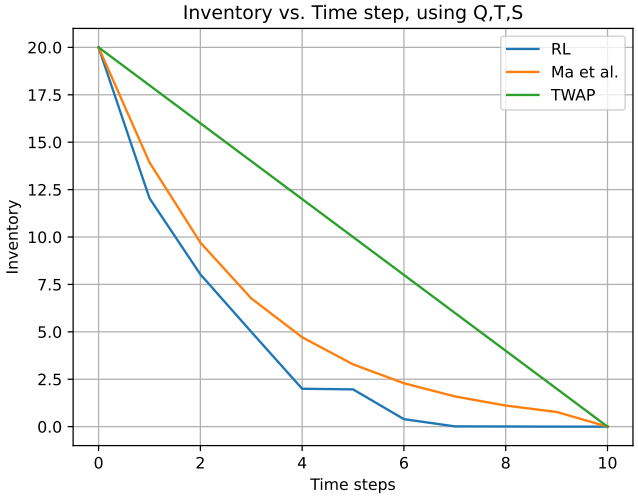


Figure 3.8: Comparison of the optimal inventory holding in Ma et al. [2020] with those found by the DDQL agent. The trading strategy is the average trading strategy over the 5,000 trajectories where the impacts can change from high or low levels independently.

Table 3.11, showing how the proposed algorithm is able to retrieve a cost very close to that of the optimal strategy since $\Delta P\&L = 0.25$ bps.

Table 3.11: Comparison between the costs of the DDQL agent and of the model in Ma et al. [2020]. ΔP&L values are reported in basis points, the standard error of the nominal raw differences between the cost of the strategies is in parenthesis.

	Regime switching impact coefficients			
	Theo.	**DDQL**	ΔP&L	ΔP&L
features	$\mathbb{E}[IS]$	$\mathbb{E}[IS]$	**avg.**	**std.dev.**
Q,T,S	0.4794	0.4744 (0.0041)	0.256	1.12

Stochastic mean reverting impacts

Lastly, we want to consider the most complex model for the impacts dynamics. Hence, as an analytical point of comparison, we use the third model of Section 3.2.1, presented in continuous form but used for the purpose of our experiments in its discretised formulation, for which an approximated solution can be obtained as a Taylor expansion on the long run means of the temporary and permanent impact dynamics as outlined in Barger and Lorig [2019]. In this specific model, the resulting optimal execution of Equation (3.6) is in fact a perturbation of the

TWAP strategy that accounts for deviation of the impacts with respect to their long run mean values.

The aim of the experiment is to test whether the agent is able to replicate, or even beat, the performance that would be obtained if the parameters for Equation (3.6) were known. In fact, the difficulty in considering stochastic impacts when implementing an optimal execution strategy lies not just in the correct equation to be used, but in the correct parameters estimation for the impact model considered. By using the DDQL agent we aim at tackling the problem in a model agnostic way, i.e. we do not estimate or make assumptions on either the model or the impacts' magnitudes. We let the agent learn them via exploration-exploitation.

The parameters chosen for the simulations are reported in Table 3.12. We assume that both impacts hover over their long run averages and are correlated, with correlation coefficient [5] $\omega = 0.9$. Thus, the parameters are chosen such that the Feller condition holds and the values for the impact coefficients are almost surely positive[6]. We implement two types of experiments, we test for strong mean reversion ($\lambda_\alpha^{(\text{high})}$ and $\lambda_\kappa^{(\text{high})}$) and for weak mean reversion ($\lambda_\alpha^{(\text{low})}$ and $\lambda_\kappa^{(\text{low})}$) parameters, leaving long run levels and volatility of the impacts untouched. Thus testing for either strong or weak memory properties in the dynamics for the impacts.

In the experiments, for each train and test episode we simulate one path for α_t and one for κ_t. Sample paths for both α_t and κ_t are reported in Fig 3.10.

Table 3.12: Parameters used for the simulation of stochastic impacts. The parameters are chosen such that the starting values correspond to the fixed impacts in the A&C experiment above.

Parameter	Value	Parameter	Value
$\lambda_\alpha^{(\text{low})}$	1	θ_κ	0.001
$\lambda_\kappa^{(\text{low})}$	1	θ_α	0.002
$\lambda_\alpha^{(\text{high})}$	5	σ_κ	0.002
$\lambda_\kappa^{(\text{high})}$	5	σ_α	0.002
ω	0.9		

Features - Q,T When the agent can only access the level of inventory and time-steps as features, when considering the $\lambda_{\alpha,\,\kappa}^{(\text{low})}$ setup, the resulting IS is similar to the one obtained using

[5] This high value reflects the fact that different liquidity measures are typically highly correlated. We expect the same happens for temporary and permanent impact

[6] It is worth noticing that in the space of solutions of Barger and Lorig [2019] both buys and sells are allowed for a sell program, while our RL approach considers only sells. In order to provide a fair comparison between the two methods, we selected the parameters of the model in such a way that, using the solution proposed by Barger and Lorig [2019], the fraction of buys in a sell program is negligible. This parameter choice has been achieved via extensive numerical simulations.

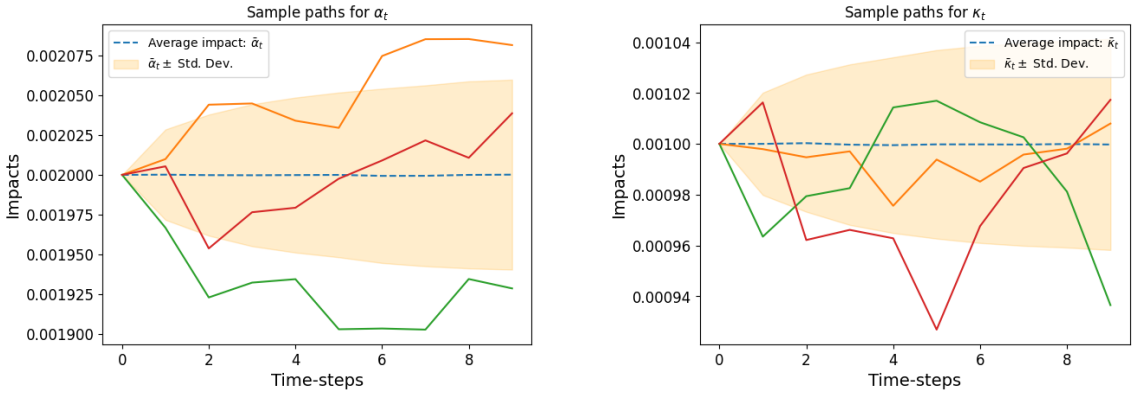


Figure 3.9: Stochastic impact. Left panel: three sample paths for α_t with $\lambda_{\alpha}^{(\text{low})}$. Right panel: three sample paths for κ_t with $\lambda_{\kappa}^{(\text{low})}$. Solid lines: three different realisations for the temporary and permanent impacts. Yellow area is the standard deviation around the average value of the impact.

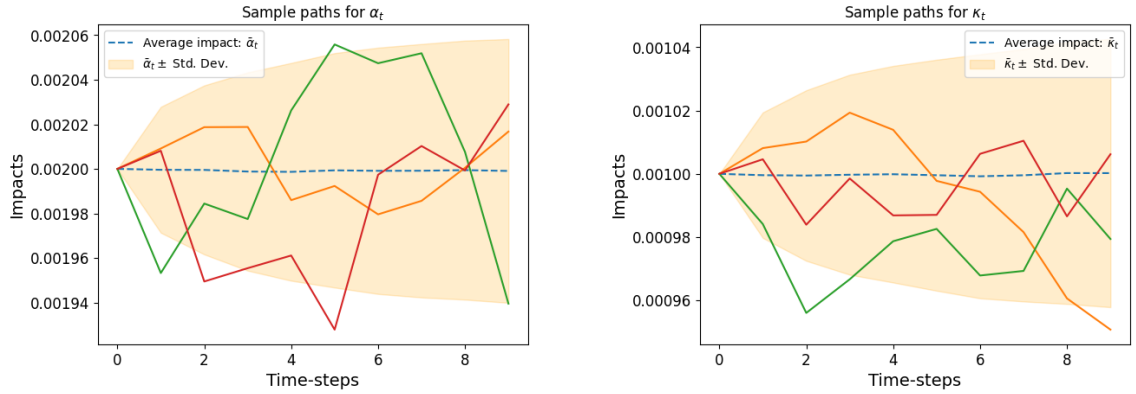


Figure 3.10: Left panel: sample paths for α_t with $\lambda_{\alpha}^{(\text{high})}$. Right panel: sample paths for κ_t $\lambda_{\kappa}^{(\text{high})}$. Solid lines: three different realisations for the temporary and permanent impacts. Yellow area is the standard deviation around the average value of the impact.

Equation (3.6) as shown in Table 3.13a. The $\Delta P\&L$ is, in fact, slightly positive but less than 2 basis points. The performance increases in the case of $\lambda_{\alpha,\,\kappa}^{(\text{high})}$ setup, as shown in Table 3.13b it seems that since the impact dynamics mean revert faster to their long-run average the agent, trained in this way, takes advantage and trades accordingly, thus obtaining lower IS and better $\Delta P\&L$.

Features - Q,T,S Adding the price to the set of features does bring significant benefits to the optimal selling schedule. Considering the $\lambda_{\alpha,\,\kappa}^{(\text{low})}$ setup, the $\Delta P\&L$ is now significantly positive, due to the increased amount of cash generated by training a DDQL agent with these features. Thus, adding the price to the set of features greatly improves the trading performance overall as seen in Table 3.13a. When the $\lambda_{\alpha,\,\kappa}^{(\text{high})}$ setup is considered, the performance of the agent is still extensively better than the approximated solution, but in line with those obtained with just

inventory and time-step as features as shown in Table 3.13b. We interpret this to be the result of the increased mean reversion rate, the agent learns that the dynamics tend to revert faster to the long run average and takes advantage with its trading. Notably, the agent trained in this way has no notion of the stochastic dynamics of the impacts parameters used in this setting, the algorithm in fact, *learns* the model and, implicitly, its parameters from the rewards obtained at each time-step and for each train episode. Thus, it is capable of performing better than an approximate closed-form solution, with the advantage that the parameters used to obtain the sale schedule are neither assumed nor estimated in any way, they are learned *implicitly* and directly from the few features observed by the agent and fed to the neural nets.

Table 3.13: Stochastic impact. Comparison between DDQL agent and theoretical solutions. ΔP&L values are reported in basis points.

(a) Stochastic mean reverting impacts: $\lambda^{(\text{low})}_{\alpha,\,\kappa}$.

Low volatility impact						
inputs	**Theo.** $\mathbb{E}[IS]$	**Theo. std.dev.**$[IS]$	**DDQL** $\mathbb{E}[IS]$	**DDQL std.dev.**$[IS]$	$\Delta P\&L$ **avg.**	$\Delta P\&L$ **std.dev.**
Q,T	0.3129	0.63	0.2789	0.011	1.8	3.19
Q,T,S	0.3129	0.63	0.2572	0.007	2.5	3.27

(b) Stochastic mean reverting impacts: $\lambda^{(\text{high})}_{\alpha,\,\kappa}$.

High volatility impact						
inputs	**Theo.** $\mathbb{E}[IS]$	**Theo. std.dev.**$[IS]$	**DDQL** $\mathbb{E}[IS]$	**DDQL std.dev.**$[IS]$	$\Delta P\&L$ **avg.**	$\Delta P\&L$ **std.dev.**
Q,T	0.5017	1.83	0.3200	0.0084	9.2	73.3
Q,T,S	0.5017	1.83	0.3158	0.0112	9.4	73.4

3.4 Conclusions

In this paper we developed a DDQL algorithm able to learn the optimal liquidation strategy when the liquidity is time-varying. By considering an Almgren-Chriss linear market impact model, we investigated several deterministic and stochastic dynamics of the temporary and the permanent impact coefficient. These include constant, linearly varying, and square-root mean reverting stochastic processes for the parameters. We also investigate the case when coefficients are linearly varying but the slope can be either positive or negative with equal probability.

We find that, when considering settings where the optimal solution is known, the algorithm is able to learn strategies with an expected cost comparable with the optimal one. More interestingly, when we consider settings where the optimal solution is not available or is available

in approximate form, we find that the DDQL algorithm finds better solutions. This indicates that RL could be successfully used in real settings where liquidity is naturally latent and time-varying.

Moreover, we empirically show how the algorithm is sensitive, relative to the achieved IS cost, to increasing levels of volatility. We show how adding the price as a feature, helps overcoming the problem of increased costs when the number of training episodes is limited. Furthermore, we analyse how increasing the number of training episodes helps achieving costs in line with the theoretical optimum, once again confirming the fact that static strategies are equivalent to dynamical ones. At the same time, our methodology provides a way around a-possibly-lengthy training phase by just adding the observed mid-price as feature to the algorithm.

There are several possible extensions of our work. First of all, we considered a linear impact model, whereas it is known that temporary market impact is non-linear as shown in Almgren et al. [2005]. Second, more sophisticated market impact models, such as the Transient Impact Model by Bouchaud et al. [2003] and its extension by Bouchaud et al. [2009], better describes the reaction of prices to trades. Finally, other dimensions of liquidity, e.g. the bid-ask spread, are important components of transaction cost and are known to be time varying with non-trivial dynamics. It is certainly interesting to explore if RL based approaches are capable to find optimal trading schedules also in these more complex and realistic environments. We leave these interesting questions for future research.

Appendix A

Learning process

A.1 Constant impacts

In this Appendix we study the convergence of the learning process of the DDQL algorithm. When volatility, the only source of noise in our experiments, is small, we observe that the convergence is typically fast. On the contrary, when volatility is high, the convergence can be quite small and the sample size of the training set might be insufficient to reach a fixed point. Notice that, in the A&C setting the IS should be independent of the volatility of the price process and in general the optimal strategy does not depend on price when impact is constant or deterministically time-varying.

We use the same setting of Section 3.1.1 with constant impact parameters and three different levels of volatility. Considering the case of a training set composed by 10,000 episodes, we plot in Figure A.1 the loss as a function of the iterations of the learning process. We observe that under high volatility conditions, the DDQL agent fails to converge to the optimal policy. This is evidenced by the persistent discrepancy between the target Q-values and the actual Q-values, as reflected by a high and fluctuating loss in the final 50 iterations in the case of $\sigma = 0.1$. By contrast, in lower volatility scenarios, the loss is significantly lower, indicating improved convergence. This behaviour directly translates into higher costs for the $\sigma = 0.1$ case and lower IS costs for the remaining two cases. Furthermore, analysing the costs reported in Tables 3.3a and 3.3b, we note that costs are even lower when Q, T, S are used as features. At first glance, this may seem contradictory to the principle that static strategies should be equivalent to dynamic ones in a deterministic impact model.

To further investigate this, we conducted an additional experiment with very large volatility,

$\sigma = 0.1$, increasing the number of training episodes to 20,000 while adjusting the reset rate to $m = 200$ to maintain the rate of decrease of the ϵ parameter. This adjustment preserved the balance between exploration and exploitation while allowing the agent to explore more effectively. As reported in Table 3.3c, this modification led to lower costs, confirming that deterministic strategies are sufficient to achieve optimal costs. This is further supported by the decrease in loss levels observed in Figure A.2, where the loss for $\sigma = 0.1$ declines and remains low towards the end of training. This suggests that the agent successfully approximated an optimal Q-function.

A.2 Time varying impacts

The same result is achieved for the decreasing or increasing impact case when both training and testing are performed on the same time dependent structure of the impacts. In fact, considering 20,000 iterations instead of 10,000 and adjusting the batch size to 16 instead of 32, we achieve a $\Delta P\&L = -0.353$ bps for the Q, T case as reported in Table A.1. This means that the DDQL algorithm's learning phase is sensitive not to the environment's parameters, such as the volatility level or the impact magnitudes, but to the number of scenarios that the agent is able to learn upon, ultimately confirming how in the case of deterministic impacts static strategies are as effective a dynamic ones. However, in real life scenarios one usually lacks the amount of data needed to train the algorithm, ultimately implying that having more features do achieve satisfactory results with much less training iterations.

Table A.1: Decreasing impact. Comparison between the costs of the DDQL agent, of the theoretical solutions, and of a TWAP strategy. ΔP&L values are reported in basis points. The standard error of the raw differences between the theoretical solution and the RL based one is in reported in parenthesis.

	Decreasing impact with 20,000 training episodes				
	Theo.	**DDQL**	$\Delta P\&L$ vs. Theo.	$\Delta P\&L$ vs. Theo.	**TWAP**
inputs	$\mathbb{E}[IS]$	$\mathbb{E}[IS]$	**avg.**	**std.dev.**	$\mathbb{E}[IS]$
Q,T	0.2566	0.2653 (0.00025)	-0.3539	0.03155	0.3588

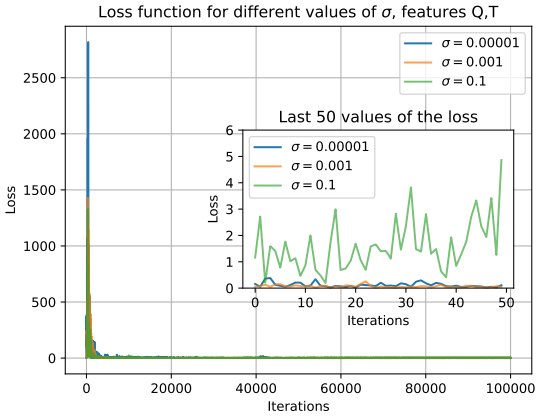


Figure A.1: Loss values for a training of 10,000 episodes of 10 time steps for different volatility levels. The figure refers to a standard A&C setting with constant parameters.

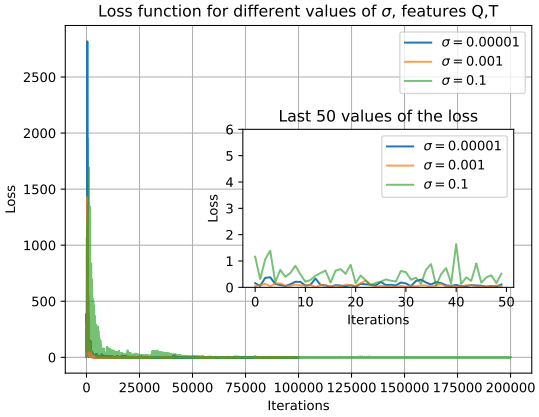


Figure A.2: Loss values for a training of 10,000 episodes of 10 time steps for $\sigma = 0.00001$ and $\sigma = 0.001$, and loss values for a training of 20,000 episodes of 10 time steps for $\sigma = 0.1$. The figure refers to a standard A&C setting with constant parameters.

A.3 Additional figures

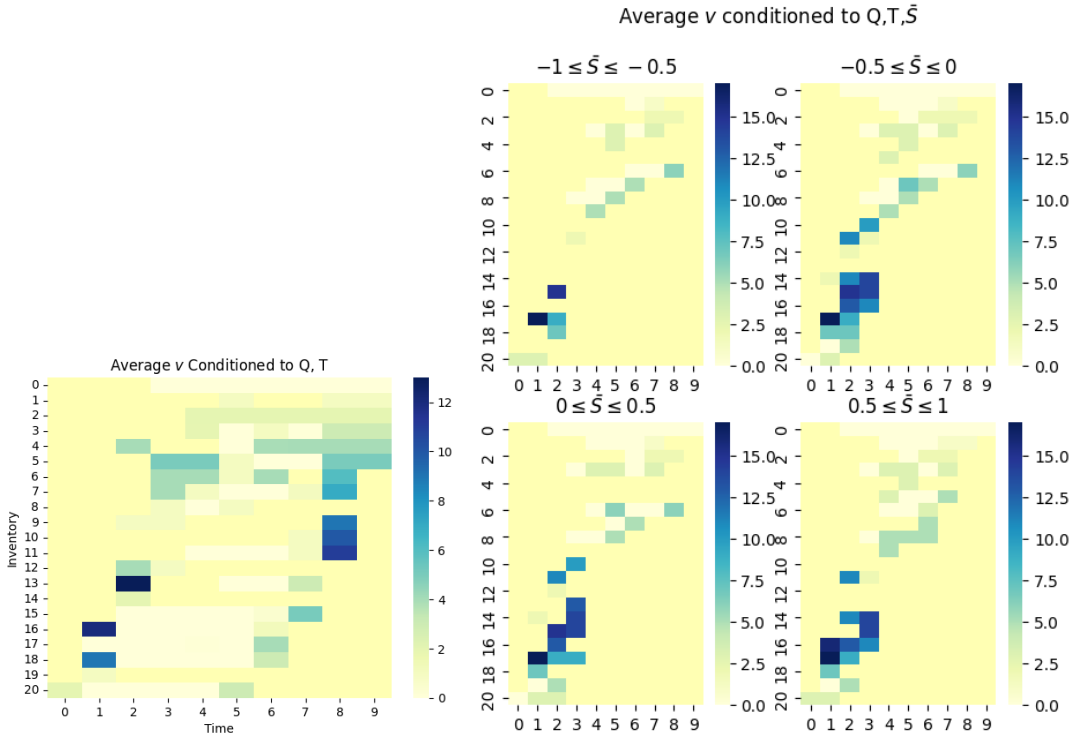


Figure A.3: Mixed train, test with increasing impact. Top panel: average number of stock sold per time-step t and inventory level q_t. Bottom panel: average number of stock sold per time-step t, inventory level q_t and normalised price $\bar{S}$

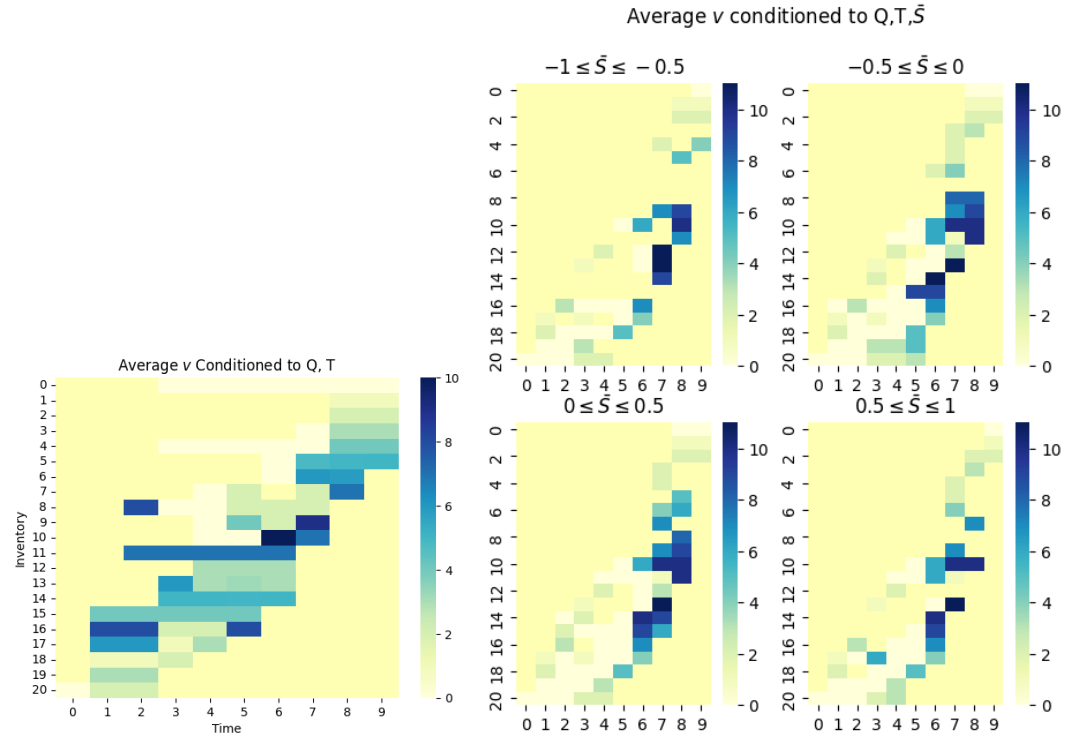


Figure A.4: Mixed train, test with decreasing impact. Top panel: average number of stock sold per time-step t and inventory level q_t. Bottom panel: average number of stock sold per time-step t, inventory level q_t and normalised price $\bar{S}$

Chapter 4

Optimal trading with reinforcement learning

> "It always takes longer than you expect, even when you take into account Hofstadter's Law."
>
> Hofstadter's Law

The contents of this chapter are available as a preprint, and are currently under review in Macrì et al. [2025].

Abstract Reinforcement Learning (RL) applied to financial problems has been the subject of a lively area of research. The use of RL for optimal trading strategies that exploit latent information in the market is, to the best of our knowledge, not widely tackled. In this paper we study an optimal trading problem, where a trading signal follows an Ornstein–Uhlenbeck process with regime-switching dynamics. We employ a blend of RL and Recurrent Neural Networks (RNN) in order to make the most at extracting underlying information from the trading signal with latent parameters. The latent parameters driving mean reversion, speed, and volatility are filtered from observations of the signal using RL, and trading strategies are derived via RL. To address this problem, we propose three Deep Deterministic Policy Gradient (DDPG)–based algorithms that integrate Gated Recurrent Unit (GRU) networks to capture temporal dependencies in the signal. The first, a one-step approach (hid-DDPG), directly encodes hidden states from the GRU into the RL trader. The second and third are two-step methods: one (prob-DDPG) makes use of posterior regime probability estimates, while the other (reg-DDPG) relies

on forecasts of the next signal value. Through extensive simulations with increasingly complex Markovian regime dynamics for the trading signal's parameters, as well as an empirical application to equity pair trading, we find that prob-DDPG achieves superior cumulative rewards and exhibits more interpretable strategies. By contrast, reg-DDPG provides limited benefits, while hid-DDPG offers intermediate performance with less interpretable strategies. Our results show that the quality and structure of the information supplied to the agent are crucial: embedding probabilistic insights into latent regimes substantially improves both profitability and robustness of reinforcement learning–based trading strategies.

4.1 Introduction

Optimal trading strategies have been the subject of an active and flourishing area of research since the advent of electronic markets several decades ago. The development of more efficient quoting systems and the faster spread of information have led major market participants, on both sell and buy sides, to embed ever more granular information into their trading algorithms.

In model-based approaches to algorithmic trading, recent work includes the incorporation of trading signals into the classical optimal stochastic control formulation of trading, starting with the works of Gârleanu and Pedersen [2013] in discrete time and Cartea and Jaimungal [2016] along with Casgrain and Jaimungal [2019] (with latent signals) in continuous time. Thanks to the advent of machine learning (ML), research has begun to focus on model-agnostic approaches to incorporate trading signals. In this regard, among the many methods proposed in literature, those that propose forecasting algorithms that use Recurrent Neural Networks (RNNs) show the greatest promise. For example, Tsantekidis et al. [2020], investigate many architectures for predicting future price levels starting from Limit Order Book (LOB) information; in Sirignano and Cont [2021] the authors use high frequency data containing all orders, transactions and order cancellations for approximately 1000 stocks traded on NASDAQ to predict their returns; in Zhang et al. [2019b], the authors propose a new convolutional neural network on top of an LSTM layer to forecast stock returns at high frequency and, finally, in Kolm et al. [2023] where the authors compare the performance of several machine learning architectures in extracting excess return information for multiple horizons. Extracting market signals, however, is only one side of the problem, the other side is how to use those signals for optimal trading.

Another stream of literature employs deep reinforcement learning to directly seek optimal trading policies, or strategies, that maximise profits. For example, Ning et al. [2021] use double deep Q-networks (DDQN) for optimal execution problems and Casgrain et al. [2022] extends it to the multi-agent RL paradigm[1]; Zhang et al. [2019a] compare different paradigms of RL against classical time series momentum strategies on optimal futures trading, showing how RL-based methods outperform such baseline models; Yang et al. [2020] train an RL agent that makes use of an 'ensemble' strategy - an ensemble of trading strategies that uses three actor-critic based algorithms: Proximal Policy Optimisation (PPO), Advantage Actor Critic (A2C), and Deep Deterministic Policy Gradient (DDPG). This particular combined architecture enables the

[1]The first versions of these works appeared online in 2018 and 2019.

agent to inherit and integrate the best features of all three algorithms, thus robustly adjusting to different market situations. Cartea et al. [2023] use DDQN and develop a new reinforced deep Markov model (RDMM) for statistical arbitrage. Finally, Briola et al. [2021] start from an environment based on LOBs for a particular stock, and train a PPO agent to trade one unit of asset per time step, leading the agent to learn trading strategies that deliver stable positive returns in a highly stochastic and non-stationary environment. This stream of literature is quite active and thus many other methods have been developed to tackle other kind of problems when it comes to asset trading. For a more complete review of the methods available please refer to the contributions of Millea [2021], Zou et al. [2023], Hambly et al. [2023], among the many.

To the best of our knowledge, however, none of the existing literature makes use of latent information embedded in a signal process as part of training for an RL agent and to use this to obtain optimal trading strategies. Thus, in this contribution, we fill this gap in the literature by training an RL agent whose goal is to optimally trade a signal, hence maximizing their future discounted rewards from trading by making use of hidden information coming from the signal itself. We model the trading signal as a mean-reverting process; in this context, the information embedded in the training of the agent comes in the form of the probability of mean reversion to one of the different long-run regimes to which the signal mean reverts to. We progressively consider more complicated signal dynamics and show how embedding information in the form of the probability of mean reversion to a regime improves dramatically the performance of the RL agent both in a simulated environment and with real data. In this contribution, we develop and test three DDPG-based algorithms to solve an optimal trading problem with mean-reverting signals governed by Markov chains. We incorporate GRU networks to capture the time-dependent structure of the trading signal, leveraging on this we explore different ways of feeding information to the RL agent. Our findings show that providing posterior probability estimates of the underlying mean-reversion regimes consistently yields the highest rewards, both in synthetic environments with multiple regime dynamics and in real market pair-trading data. In contrast, supplying next-step signal forecasts adds little benefit, while using GRU hidden states provides intermediate performance. Overall, the results highlight that the quality and type of information provided to the agent is crucial: interpretable structured insights into the data-generating process substantially improve the effectiveness and robustness of the learning.

The chapter is organised as follows, in Section 2 we define the trading problem tackled throughout the remainder of the paper, in Section 3 introduces the methods used to model the

trading problem, Section 4 discusses the results for different model scenarios with simulated data, Section 5 discusses the results obtained by applying the algorithms to real data, and finally, Section 6 provides conclusions and outlines further research directions.

4.2 Optimal trading problem

4.2.1 Market model

A risk-neutral investor aims to maximise her expected discounted profits from trading a signal $(S_t)_{t\geq 0}$. We assume that S_t follows an Ornstein-Uhlenbeck process with time varying parameters, thus it satisfies the stochastic differential equation (SDE):

$$dS_t = \kappa_t(\theta_t - S_t)\,dt + \sigma_t\,dW_t\,, \tag{4.1}$$

where $W = (W_t)_{t\geq 0}$ is a Brownian motion in a suitable probability space and the parameters $\kappa_t \geq 0$, $\theta_t > 0$, and $\sigma_t > 0$ are, respectively, the long run mean at which the process mean reverts to, the velocity of mean reversion, and the volatility of the process at time t.

The existence of mean reversion creates opportunities for signal forecasting and therefore for profitable trading strategies, because when the signal is above/below the long term value θ_t, then for the agent it is optimal to go short/long on the signal. The parameters, however, are generally latent and must be estimated dynamically together with the optimization of the strategy. To overcome this issue, we take the approach from partial information stochastic control and filter θ_t, κ_t, σ_t from the observations of the dynamics of $(S_u)_{u\in[0,t]}$.

To gain some insight, consider the case where κ_t and σ_t are constant, while θ_t follows some unknown dynamics. In this case, the solution to SDE (4.1) may be written as

$$S_{t+\Delta t} = e^{-\kappa\,\Delta t}\,S_t + \kappa\int_t^{t+\Delta t} e^{-\kappa(t-u)}\,\theta_u\,du + \sigma\int_t^{t+\Delta t} e^{-\kappa(t-u)}\,dW_u\,. \tag{4.2}$$

As expected, there is a strong dependence of $S_{t+\Delta t}$ on the level of θ_t. Moreover, the expected change in the price is given by

$$\begin{aligned}\mathbb{E}[S_{t+\Delta t} - S_t \,|\, \mathcal{F}_t] &= (e^{-\kappa\,\Delta t} - 1)\,S_t + \kappa\int_t^{t+\Delta t} e^{-\kappa(t-u)}\,\mathbb{E}[\theta_u \,|\, \mathcal{F}_t]\;du \\ &\approx \kappa\,\Delta t\;(\mathbb{E}[\theta_t \,|\, \mathcal{F}_t] - S_t) + o(\Delta t).\end{aligned} \tag{4.3}$$

The relationship in Equation (4.3) shows how the expected increase in S_t is related to the expected level of θ_t conditioned on the filtration $\mathcal{F}_t$ generated by S_t. In this paper, we consider a setting where a risk neutral agent maximizes the expected discounted profit from trading over a finite time horizon when the dynamics of the signal is given by Equation (4.1) and the parameters θ_t, κ_t, and σ_t are driven by a regime switching Markov chain. Moreover, we implement a Deep Reinforcement Learning (RL) approach to solve the optimization problem by considering a discretized version of the dynamics in Equation (4.1).

More specifically, we discretize the time in length intervals τ and we index the discrete time steps with the integer numbers, $t = 0, 1, 2, ...$ At the start of the trading window ($t = 0$), the agent has an initial endowment of inventory I_0. The agent's action is identified with the new position she wishes to hold, i.e. $a_t = I_{t+1}$, and the volume of trades is[2] $q_t = I_{t+1} - I_t,\ \forall\ t \in \mathbb{N}$. Whenever the trader performs a trade of volume q_t, she pays a transaction cost per unit volume of $\lambda \geq 0$. As a result, the reward r_t from the trading action at time t is

$$r_t(q_t, S_t, S_{t+1}, \lambda) := I_{t+1}\left(S_{t+1} - S_t\right) - \lambda\left|q_t\right|. \tag{4.4}$$

This reward represents the change in the trader's gains process accounting for transaction fees.

The trader's performance criterion is the expected future discounted sum of rewards

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t\, r_t(q_t, S_t, S_{t+1}, \lambda) \;\Big|\; I_0, S_0\right],$$

where $\gamma \in (0, 1)$ is a discount factor. The trader aims to optimise this criterion over $\mathcal{F}$-adapted admissible strategies $(q_t)_{t\geq 0}$ that are square integrable, denoted $\mathcal{A}$, i.e., the trader aims to find the strategy that optimizes

$$\max_{(q_t)_{t\geq 0}\in\mathcal{A}} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t\, r_t(q_t, S_t, S_{t+1}, \lambda) \;\Big|\; I_0, S_0\right]. \tag{P1}$$

Problem (P1) is an infinite time horizon problem, and we seek stationary strategies that depend only on the state of environment at that time, and not explicitly on time.

As mentioned above, each parameter θ_t, κ_t, and σ_t follows an independent regime switching Markov chain. More precisely, we consider increasingly more complex dynamics for S_t. Where only θ_t follows a regime switching Markov chain dynamics; next, where both θ_t, κ_t follow regime

[2]Positive (negative) values of q_t correspond to purchases (sales).

switching dynamics; and finally, where all θ_t, κ_t, and σ_t follows regime switching dynamics.

In the first setting, θ_t has three regimes $\theta_t \in \{\phi_1, \phi_2, \phi_3\}$ and the switching between the regimes is modelled using a Markov chain with transition rate matrix A and τ being the length of the interval between t and $t-1$, so that:

$$\mathbb{P}(\theta_t = \phi_j | \theta_{t-1} = \phi_i) = [e^{A\tau}]_{ij} \tag{4.5}$$

We next consider the case where also the speed of mean reversion κ_t follows an independent two state Markov chain with $\kappa_t \in \{\psi_1, \psi_2\}$, corresponding to slow and fast relaxation to the mean. Finally, we allow for regime switching volatility, with $\sigma_t \in \{\xi_1, \xi_2\}$, and again the Markov chain is independent from the other two. Such dynamics is quite complicated and difficult to filter from the mere observation of S_t levels.

To simplify the presentation, in the following we detail the case where only the mean reversion level θ_t follows a Markov chain, while κ and σ are constant. The generalization to the more complex cases is straightforward and the numerical results are shown for all models.

4.3 Learning Algorithms

As in any optimization setting with latent and time-varying parameters, the problem faced by the algorithm is two-fold. The first consists in using the observed data to filter the latent parameters, while the second consists in finding the optimal action given the current estimate of the parameters. Here we consider and compare two approaches – one where we optimize the criterion without directly filtering the states of the system, and the second where we first develop a filter and then use the filtered states as part of the optimization. The optimal trading problem can, therefore, be cast either as a unique optimization or as two consecutive optimizations

Specifically, we first propose a one-step algorithm, using as features the path of the signal process and the current inventory and returns the optimal action – which treats the filtering and optimal trading problems simultaneously. We then consider a two-step approach, where we first train a model to determine posterior probabilities for the latent θ_t parameter from the paths of the observable signal process S_t, and then use the posterior probabilities, together with the current signal and inventory, to find the optimal trading action. As a third benchmark case, we also consider the case where the filtering part forecasts the next value of the trading signal, instead of the posterior probabilities, which is then used as feature in the part of the algorithm

responsible of the trading action. In these last two settings the filtering and optimal trading are performed separately. Moreover, the setting where posterior probabilities are initially learned requires the knowledge of the true state of the latent process, a characteristic that is typically not available in real settings.

From an architectural point of view, we employ a Gated Recurrent Unit (GRU) network (introduced by Cho et al. [2014]). The GRU network is a Recurrent Neural Network able to to encode time dependency and thus deal with time series. One could also employ, e.g., long-short term memory processes or self-attention and other variants.

To approximate the optimal trading policy we employ, as anticipated, Deep Reinforcement Learning, more specifically we use a Deep Deterministic Policy Gradient (DDPG) algorithm, first introduced by Lillicrap et al. [2015], which employs an Actor-Critic approach. The algorithm makes use of two distinct neural networks, an Actor network π that is responsible to choose the action to perform, based on the state of the environment, and a Critic network Q that evaluates the 'quality' of the action chosen using the network π given the state of the environment. For each of the algorithm proposed we train the RL agent over a number N of training episodes. In our setting, the agent is tasked with the objective of maximising the rewards obtained by rebalancing the inventory holdings I as defined in Equation (4.4). The agent's actions consist of the new level of inventory to be held. Thus, the agent seeks to learn the optimal rebalancing their level of inventory depending on the signal process S_t. After the N trading episodes we test what the agent has learnt by feeding other M episodes, where the agent trades using the policy learnt during the training phase.

4.3.1 States, environment, actions, and rewards

While training, the agent may be at some time t with an inventory I_t and must decide how much inventory to hold at time $t+1$, i.e., I_{t+1}. The agent has access to the signal value at that time S_t and to the $t-W$ past observations of the signal, where $W+1$ is the length of the signal window, which we denote $\{S_u\}_{u=t}^{t-W}$. A visual representation of the information available to the agent is shown in Figure 4.1. These features/information are used to train an agent to optimize their sum of discounted rewards using a GRU network and two variants of the DDPG algorithm – i.e., we consider two approaches to the joint problem of filtering the signal and finding the optimal trading strategy employing this set of information about the environment available to the agent.

For training purposes, we simulate batches of size b of time series for the signal $\{S_u\}_{u=t}^{t+W+1}$ (i.e., signal time-series of length $W+2$) and inventories I_t. Inventories I_t are randomly sampled as $I_t \sim \mathcal{U}_{[I_{\min}, I_{\max}]}$.[3]

For the signal, we simulate the time series of the signal S_t of length $W+2$, with parameters according to the setting being considered. The starting value for the signal's time series simulation is $S_{t-W} \sim \mathcal{N}(\mu_{\text{inv}}, 3\,\sigma_{\text{inv}})$ where $\sigma_{\text{inv}} = \frac{\sigma}{2\kappa}$ which is the invariant volatility for the Ornstein-Uhlenbeck process with time varying parameters in Equation (4.1), while μ_{inv} is the invariant mean of the trading signal[4].

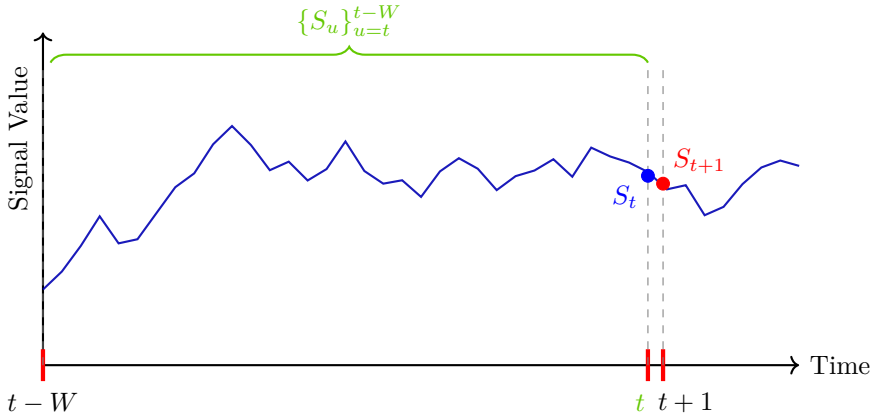


Figure 4.1: Information about the signal that can be used by the agent. The agent finds itself at some time t, has access to the past values of the signal from time t back to time $t-W$ and decides how much inventory to be held at time $t+1$.

In the testing phase, where we assess the trading policy learnt by the agent, we assume that the agent trades over a time-window of n time-steps. We assume that the agent starts their trading at time $t=0$ and has access to information on the past W values of the signal S. As time t progresses, the agent uses the observations of the signal from time t back to time $t-W$, in a rolling window fashion, to decide the inventory to be held at time $t+1$. The agent can hold inventories within the interval $I \in [-I_{\max}, I_{\max}]$ and we always start the testing episodes of trading with $S_0 = 1$ and $I_0 = 0$.

As mentioned, we employ two main approaches: In the *one-step approach* we set the states, or information about the environment, available to the agent as tuples of (S_t, I_t, o_t). Here S_t is the signal value at time t, I_t is the value of the inventory held at time t, and o_t is the output of a Recurrent Neural Network that takes as input a collection $\{S_u\}_{u=t}^{t-W}$ of past values of the

[3] Where $\mathcal{U}$ is the uniform distribution.

[4] In the settings where σ and κ are time varying in order to compute the initial value of S_0, we choose the minimal value according to the regimes used.

signal from time $t - W$ up to time t (see below for more details).

In the *two-step approach* we decouple the filtering from the optimal trading task and consider two different settings. In the first setting, we first train the algorithm to learn the regimes from the past and current value of the signal. Specifically, the posterior probabilities $\Phi_{t,k} := \mathbb{P}(\theta_t = \phi_k | \{S_u\}_{u=t}^{t-W})$ and $k = 1, 2, 3$ are learnt offline using a GRU network followed by a feed-forward network with soft-max activation output layer (to provide estimates of the posterior probability of the regime based on the historical price signal). Then the DDPG uses the tuple $(S_t, I_t, \{\Phi_{t,k}\}_{k=1,2,3})$ as features to learn the optimal trading. In the third setting, we first train the algorithm to forecast the next value of the signal $\tilde{S}_{t+1}$ and then the DDPG takes as input features the tuple $(S_t, I_t, \tilde{S}_{t+1})$.

The action that the agent can take at each time step t consists of a rebalancing of her inventory, which can be a long ($I_t > 0$) or short ($I_t < 0$) position, and is chosen according to the algorithms described below. As a constraint, we impose that at each time, the agent holds $I_t \in [I_{\min}, I_{\max}]$, i.e. the actions that can be chosen limit I_t between a maximum and a minimum inventory.

After the action has been taken by the agent in a state of the environment at time t, the reward is calculated considering the subsequent change in the signal process and measuring the gain obtained by the agent trading. The reward used is given by Equation (4.4).

In all algorithms, we let the agent explore a wide range of combinations of inventory holdings I_t and possible S_t values or, in the case of the two-step procedures, multiple combinations of states $(I_t, S_t, \Phi_{t,k})$ or $(I_t, S_t, \tilde{S}_{t+1})$. This is done through an exploration-exploitation scheme. Thus, during the learning procedure, the agent employs randomized actions, which is a randomized perturbation of the current policy, the exploration process controlled by an exploration parameter ε that decays as the training unfolds.

4.3.2 The one-step approach

To approximate the optimal strategy for Problem (P1) when the dynamics of the trading signal are described by Equation (4.1) with regime switching parameters, we first propose a one-step procedure that employs the signal's past history as part of the state of the environment that feeds into a DDPG algorithm, employing the Actor-Critic approach. To account for the time dependency of the trading signal, we encode this information using a GRU network whose output is then fed into the DDPG algorithm. A directed graph representation of the one-step

approach is reported in Figure 4.2.

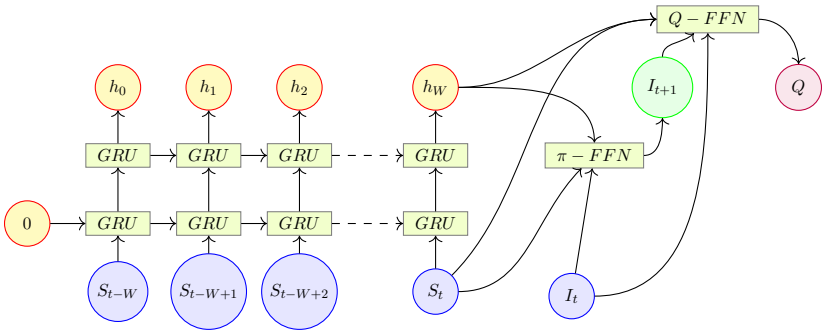


Figure 4.2: Directed graph representation of the neural network architecture for the one-step architecture. In the figure $d_\ell = 2$.

The training loop

The algorithm is trained over N iterations, for each iteration we sample b batches of input data that are used to train the algorithm. In each iteration, both the Agent and the Critic network are updated by feeding to the networks independent batches of states of the world represented by a matrix $\mathbf{F} \in \mathbb{R}^{b\times(W+2)}$, where each of the b rows contains a tuple of the form $(\{S_u\}_{u=t}^{t-W}, I_t)$.

The GRU network. The architecture, shown in Figure 4.2, is composed by $d_l \times (W+1)$ GRU units, where d_l is the number of layers. The units of the first layer receive input from the data, whereas the subsequent layers receive the hidden states generated by the previous layers. For each iteration $m = 1, ..., N$ the data are contained in the sub-matrix $\mathbf{Z} \in \mathbb{R}^{b\times(W+1)}$ of $\mathbf{F}$ containing the past and present $W+1$ observations of S, which will be encoded using a GRU network. The GRU has fewer parameters than the LSTM (which has three gates), leading to lower memory consumption and faster training, making it suitable for integration in larger learning architectures.

The GRU iteratively applies a function to the input sequence $\{S_u\}_{u=t}^{t-W}$. For the first layer of GRUs, at each time step $k \in \{0, W\}$ of the input sequence, for each column $Z_k \in \mathbb{R}^b$ of the matrix $\mathbf{Z}$ and for each layer in the net, the GRU first updates the *reset gate*:

$$p_k = \sigma(\mathbf{H}_p Z_k + \mathbf{U}_p h_{k-1} + \tilde{b}_p) \tag{4.6}$$

The *update gate* is computed similarly:

$$z_k = \sigma(\mathbf{H}_z Z_k + \mathbf{U}_z h_{k-1} + \tilde{b}_z) \tag{4.7}$$

Then, the *candidate hidden* state is:

$$\tilde{h}_k = \sigma(\mathbf{H}_h Z_k + \mathbf{U}_h(p_k * h_{k-1}) + \tilde{b}_h) \tag{4.8}$$

And the *final hidden* state update follows:

$$h_k = (1 - z_k) * h_{k-1} + z_k * \tilde{h}_k \tag{4.9}$$

Here $*$ denotes the element-wise product and we set h_{-1} equal to the zero vector. The reset gate $p_k \in \mathbb{R}^{d_h}$ allows the GRU to discard or retain past information selectively, while the update gate $z_k \in \mathbb{R}^{d_h}$ controls how much of the past hidden state is carried forward. Denoting the dimension of the hidden layer with d_h, we notice that $\mathbf{U}_p, \mathbf{U}_z, \mathbf{U}_h \in \mathbb{R}^{d_h \times d_h}$, $\mathbf{H}_p, \mathbf{H}_z, \mathbf{H}_h \in \mathbb{R}^{d_h \times b}$, $h_{k-1} \in \mathbb{R}^{d_h}$ is the hidden state from the previous time step, the bias vectors are $\tilde{b}_p$ and $\tilde{b}_h \in \mathbb{R}^{d_h}$. Finally, σ is the hyperbolic tangent (`tanh`) activation function.

The GRU units for subsequent layers take as input the sequence of hidden layers generated by the previous layer. In other words, the vector Z_k in Equations (4.6)-(4.8) is replaced by the vector h_k of the previous layer. For this reason, for the layers different from the firs one we have $\mathbf{H}_p, \mathbf{H}_z, \mathbf{H}_h \in \mathbb{R}^{d_h \times d_h}$.

From the GRU network we take as output the hidden state for $k = W$ of the last layer ℓ. Therefore the dimension of the output is $o_t \in \mathbb{R}^b$ where we have added the subscript t to indicate the time index of the operations performed by the Actor-Critic part of the algorithm.

The final matrix $\mathbf{G}_t \in \mathbb{R}^{(b \times 3)}$, which is used as input to the Actor and Critic networks, contains the GRU output vector o_t as well as the inventory and the current signal value from the matrix $\mathbf{F}$. Note that the GRU output is not used to predict the next signal value, but rather to encode the temporal dependence structure of the signal process.

Before updating the parameters of the Critic network Q and of the Actor network π, we first optimise those of the GRU. To do so, we produce an estimate $\tilde{S}_{t+1}$ of the next signal by passing the final GRU hidden states through a linear layer with a `LeakyReLU` activation. The GRU parameters are trained by minimising the mean squared error between S_{t+1} and $\tilde{S}_{t+1}$. The GRU parameters are therefore updated at each iteration of the algorithm, similarly to the Q and π networks, but the GRU training per iteration occurs before the training of the Actor and the Critic networks.

The Actor and Critic networks. The Critic and Actor neural networks are feed-forward nets with four and three input neurons, respectively, a number of l_{NN} layers of d_{NN} hidden nodes, each with `SiLu` activation function, in the Actor network, the final layer has `tanh` activation function, whose output is then scaled by $I_{\max}$.

As we are not assuming any initial or final inventory goals or penalties, in our DDPG implementation, while training we choose not to keep memory of the states that the agent has visited during training. Rather, we feed randomly generated initial states of the world in order to let the algorithm explore as many states as possible, as discussed in Section 4.3.1.

This is done to provide the agent with as many different states as possible when training. Moreover, since we do not consider any form of permanent impact generated by the agent when trading, the buy and sell actions performed by the agent have no direct effect on the signal process, and therefore no memory needs to be kept.

At the beginning of the training loop we initialise the networks π, Q with random weights μ_π, μ_Q. As training unfolds, $(\mu_\pi, \mu_Q) \to (\mu_\pi^*, \mu_Q^*)$, where (μ_π^*, μ_Q^*) are the optimal weights that correspond to the optimal policy that maximises the the sum of discounted rewards.

During training the Q network, in order to avoid an overestimation of the Q-values for the considered state-action pairs, we use a Q_{tgt} network that is initialised at the beginning of the training routine as a copy of the Q network with weights $\mu_{Q_{\text{tgt}}} = \mu_Q$, thus employing double deep Q-learning for the Critic part of the algorithm, the weights of the $\mu_{Q_{\text{tgt}}}$ are then periodically set equal to μ_Q. In order to do so, we have adopted the soft update technique as described in the original DDPG paper by Lillicrap et al. [2015], with the soft update parameter set to 0.001.

The Critic network. For each training iteration, we train a Critic network Q. To this end, we add an additive exploration zero mean noise to the output of the Actor network π (whose training is described below), whose variance declines at each iteration[5]. The additive noise, ε, helps the Actor neural network to explore the range of inventory holdings to be held depending on the state. Thus, while training the Q network, $I_{t+1} = \pi(\mathbf{G}_t|\mu_\pi) + \mathcal{N}(0, \varepsilon)$, where $\pi(\mathbf{G}_t|\mu_\pi)$ is the output of the Actor network. Clearly, as the training unfolds and ε becomes smaller, the exploration noise diminishes and the policy chosen by the network π becomes deterministic. In

[5]Specifically, at the learning loop start we set $a > 0$ and $\varepsilon_{\min} < a$ Then, for each train iteration $m \in \{1, N\}$ we set $\varepsilon = \max(a/(a+m), \varepsilon_{\min})$. In this way, the more we iterate in the learning routine the lower the ε value. ε is then used as an additive noise in the Actor-Critic architecture used in both the learning algorithms.

this way, given the weights μ_π^*, found at the end of the training, the Critic network directly maps a state into an action such that:

$$I_{t+1}^* = \pi(\mathbf{G}_t|\mu_\pi^*) \tag{4.10}$$

Once the new level of inventory is obtained, it is used to calculate the reward for each state in the batch as in Equation (4.4).

To train the Q network, we update the weights μ_Q such that $\mu_Q = \arg\min_{\mu_Q} \mathcal{L}_1(\mu_Q; \mu_{Q_{\text{tgt}}})$, by taking a single gradient step on the loss

$$\mathcal{L}_1 = \frac{1}{b}\sum_{i=1}^{b}(Q(\mathbf{G}_t^{(i)}, I_{t+1}^{(i)}|\mu_Q) - y_t^{(i)})^2 \tag{4.11}$$

$$y_t^{(i)} = r_t^{(i)} + \gamma Q_{\text{tgt}}(\mathbf{G'}_{t+1}^{(i)}, \pi(\mathbf{G'}_{t+1}^{(i)}|\mu_\pi)|\mu_{Q_{\text{tgt}}}) \tag{4.12}$$

where the superscript i indicates the i-th row of the matrix $\mathbf{G}_t$ and $\mathbf{G}'_t \in \mathbb{R}^{b\times(W+2)}$ is the matrix of inputs $\mathbf{G}'_{t+1} = (o_{t+1}, S_{t+1}, I_{t+1})$, where o_{t+1} is obtained by passing to the GRU network the series of $\{S_u\}_{u=t+1}^{t-W+1}$ of past values of the signal from time $t - W + 1$ up to time $t + 1$. The weights of the Critic network μ_Q are then updated using gradient descent.

Within the training loop we iterate this training process for the Critic network a number ℓ of times to explore the space of actions given the states more thoroughly, in order to obtain better estimates for the Q-values.

The Actor network. For each training iteration we train the Actor network π, which is the neural network responsible for choosing the action I_{t+1}. To do this, we feed the π network with a batch of b states $\mathbf{G}_t$. The π network returns a new level of inventory to be held I_{t+1}, which is in turn fed to the Critic Q network, along with the state $\mathbf{G}_t$. The output of the Q network is the Q value which is maximized during training. Specifically, the weights μ_π of the Actor network are found by maximizing the output of the Critic network with fixed weights μ_Q. As before we perform a single gradient step to minimise the loss:

$$\mathcal{L}_2 = -\frac{1}{b}\sum_{i=1}^{b} Q(\mathbf{G}_t^{(i)}, \pi(\mathbf{G}_t^{(i)}|\mu_\pi)|\mu_Q) \tag{4.13}$$

where $\pi(\mathbf{G}_t^{(i)}|\mu_\pi)$ is the output of the Actor network with current weights μ_π. The weights μ_π are updated using the policy gradient theorem. The derivative of the loss function $\mathcal{L}_2$ with respect to the weights of the Actor network is:

$$\nabla_{\mu_\pi}\mathcal{L}_2 = -\frac{1}{b}\sum_{i=1}^{b}\left[\nabla_a Q(\mathbf{G}_t^{(i)}, a^{(i)}|\mu_Q)|_{a^{(i)}=\pi(\mathbf{G}_t^{(i)}|\mu_\pi)}\nabla_{\mu_\pi}(\mathbf{G}_t^{(i)}|\mu_\pi)\right] \tag{4.14}$$

where the first part in the square brackets tells us how sensitive the Q network is to changes in the action chosen by the Actor π, this is the gradient of the Q-value with respect to the action and evaluated at the action suggested by the Actor network, in this context for ease of notation we use $a^{(i)} = I_{t+1}^{(i)}$ to denote the action for the i state among the b considered in the batch. The second part inside the brackets is the gradient of the the actor's policy and accounts for how much the action I_{t+1} changes with respect to changes in the π network parameters. We update the weights for the π network with gradient descent. We repeat this training routine a number l of times in order to assess how the quality of the actions chosen by the Actor changes and, in this way, to thoroughly explore weights combinations for the π network.

The complete training algorithm for both the Critic and the Actor neural networks is reported in Algorithm 6. Both the Actor and the Critic neural networks are feed-forward fully connected neural networks, for the gradient descent we use the Weighted ADAM optimiser with a scheduler. The features for the DDPG algorithm are normalised in the domain $[0, 1]$. Relevant parameters for the networks and the market model are reported in Table 4.1. We will refer to this model as *hid-DDPG* throughout the paper.

4.3.3 The two-step approach

To approximate the optimiser of Problem (P1) when the trading signal follows the dynamics in Equation (4.1) with regime switching parameters, we further propose a two-step procedure in two different setups. For the first setup, in the first step, we approximately solve the filtering problem, aiming at estimating the posterior probability of being in one of the regimes of θ_t. In the second step, we approximate the optimal trading policies that use the estimated posterior distribution as an additional feature. These estimates, together with the signal S_t and the inventory level I_t, constitute the features used in the RL algorithm. The directed graph representation of the training for this algorithm is reported in Figure 4.3. In the third setup we repeat the procedure but, instead of filtering the posterior probabilities for the θ_t regimes,

we train a GRU to obtain estimates of the next value of the trading signal, i.e. $\tilde{S}_{t+1}$, we will use this estimate along with the signal S_t and the inventory level I_t as features to the DDPG algorithm. The directed graph representation of the training for this algorithm is reported in Figure 4.4. The DDPG step for both the two-step procedure remains the same, except for the set of features used, as discussed at the end of Section 4.3.1.

First-step: regime filtering with classification

Concerning the first step, we approximately solve the filtering problem, aiming at estimating the posterior probability of being in one of the regimes for θ_t. Further, we approximate the optimal trading policies using the estimated posterior distribution as an additional feature.

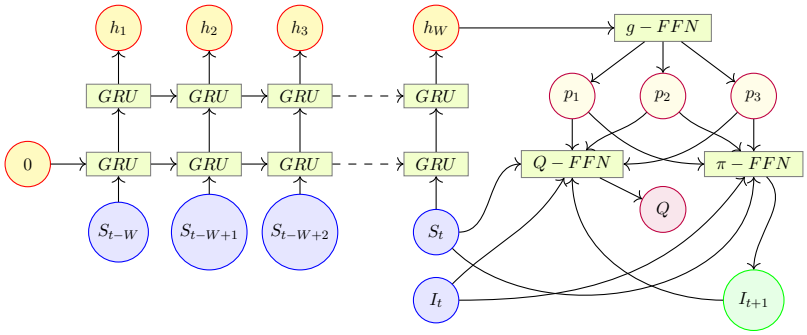


Figure 4.3: Directed graph representation of the neural network architecture for the two-step architecture with regime filtering.

In this setup, we provide a deep learning approach to estimate the probability that θ_t is in one of the regimes, conditional on the history of the path of the signal process S. That is, we aim to find an approximation

$$\Phi_{t,k} := \mathbb{P}(\theta_t = \phi_k | (S_u)_{u \leq t}) \approx g_k \left(S_t, \ldots, S_{t-W}\right) , \qquad k = 1, 2, 3 \tag{4.15}$$

The vector valued function $g(\cdot) = (g_1(\cdot), g_2(\cdot), g_3(\cdot))$ is approximated as an Artificial Neural Network (ANN) with W being the look-back window used. The input, $\{S_u\}_{u=t}^{t-W}$, to the ANN $g(\cdot)$ is a sequence of trading signals that goes from time-step t backwards to time $t - W$.

Once again, as the signal S_t is a time series, a natural choice is to employ RNNs in the approximation consistently with the one-step method outlined in Section 4.3.2 - specifically, we employ a GRU network as our function approximant $g(\cdot)$. On top of the GRU architecture, we nest additional fully connected layers to return the probability for the signal S_t to be mean reverting to a specific regime $\theta_t \in \{\phi_1, \phi_2, \phi_3\}$.

More specifically, the last hidden state is passed to a second architecture equipped with additional fully connected layers. In each hidden layer of the fully connected neural net we apply the Sigmoid Linear Unit (`SiLu`) activation function to add non-linearity. In the end, since this is a classification problem, we use the `SoftMax` activation function to retrieve the probability $\mathbb{P}\left(\theta_t = \phi_j | \{S_u\}_{u=t}^{t-W}\right)$. The loss used to train the architecture is the categorical cross entropy.

The parameters for the GRU and the additional layers are reported in Table 4.1.

First-step: regression setup

As a point of comparison we develop a third setup to the two-step procedure. In this third experiment, instead of approximating the probability of being mean reverting at each of the regimes of θ_t, the first step consists in training a GRU net on a regression problem to directly estimating the next value for the trading signal, i.e. $\tilde{S}_{t+1}$. Similarly as in the filtering problem, in order to obtain the next value estimate, we provide as input to the GRU $g(\cdot)$ a sequence $\{S_u\}_{u=t}^{t-W}$. The architecture of the GRU net remains the same but the last layer has `SiLu` activation function and we minimize the loss consisting of the squared error between the estimate $\tilde{S}_{t+1}$ and the actual next value of the trading signal S_{t+1}.

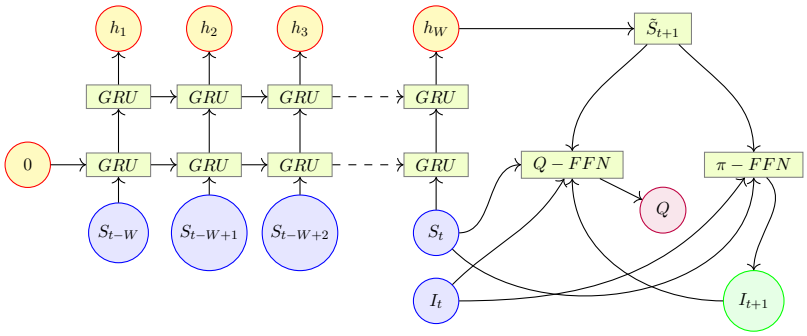


Figure 4.4: Directed graph representation of the neural network architecture for the two-step architecture with price prediction.

Second-step: optimal trading

Once training in the first-step is completed, and the GRU models are trained for either of the setups in Section 4.3.3 or Section 4.3.3, we consider the problem of a risk neutral trader who maximises the expected discounted profits from trading. We assume that the agent re-balances the inventory according to the level of the observed trading signal and on their current inventory in order to maximise the rewards obtained from trading. To do so, we again model the agent

with a DDPG algorithm similarly as in Section 4.3.2. Hence, for the optimal trading part of the algorithm we again adopt an Actor-Critic approach where we make use of two distinct neural networks, an Actor network π that is responsible for choosing the action to perform, based on the state of the environment where the agent is, and a Critic network Q that critically evaluates the 'quality' of the action chosen using the network π given the state of the environment. This is analogous to the one-step procedure with the important difference that, in the regime filtering approach, the posterior probabilities estimates on the current mean reversion regime θ_t are added to the feature set and the model that approximately solves the filtering model is trained in the previous step, while in the regression approach the estimate for the next value of the trading signal S_t is added to the set of features fed to the RL agent.

Hence, the first setup (and second method proposed) of the two-step approach is termed *prob-DDPG* and makes use of features $\mathbf{G}_t = (S_t, I_t, \Phi_{t,k})$ for the DDPG part, while the second setup (and third method overall) is termed *reg-DDPG* throughout the paper, and employs $\mathbf{G}_t = (S_t, I_t, \tilde{S}_{t+1})$ as features to the DDPG.

For the DDPG part we again set both Actor and Critic ANNs to be feed-forward fully connected neural networks as in the one-step approach. For the gradient descent steps we use the Weighted ADAM optimiser with a scheduler. The features of the DDPG algorithm are normalised in the domain $[0, 1]$. The relevant parameters for the networks and the market model are reported in Table 4.1.

4.4 Results

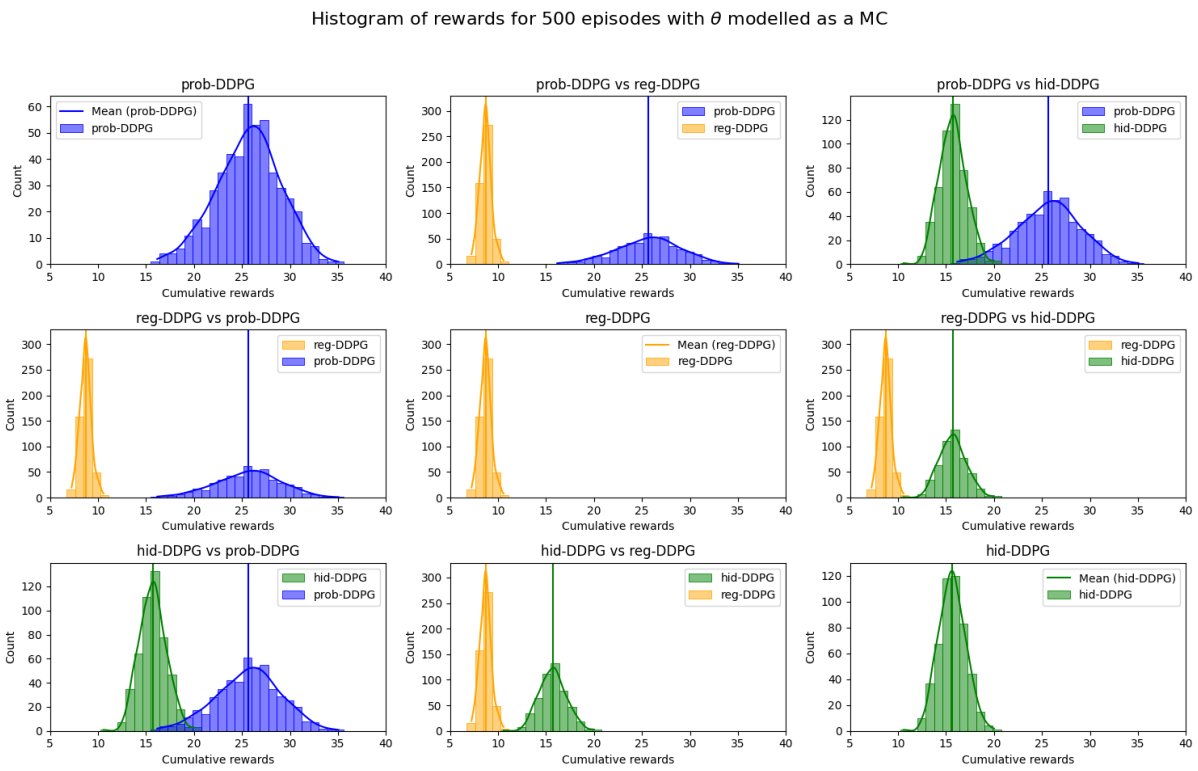


Figure 4.5: Comparison of the cumulative rewards for the different DDPG approaches when θ_t is a Markov chain.

We now test the capabilities of the approaches outlined above on synthetic data and for increasing levels of complexity for the data generating process as defined in Equation (4.1). To this end, as anticipated, we test our algorithm on environments where just θ_t is a Markov chain, where θ_t, κ_t are independent Markov chains, and finally, for the most complete case, where θ_t, κ_t and σ_t are modelled via independent Markov chains. We employ both the two-step approaches and the one-step approach over a set of testing episodes, and then we subsequently compare the results, in terms of cumulative rewards, obtained by employing the trading strategies found by the DDPG agent.

In the testing phase, we compute the total reward for each of the M testing episodes as $R_n = \sum_{t=1}^{n} r_t$ where n is the number of time steps in each iteration M where the agent trades and r_t is as defined in Equation (4.4). For each experiment, we report the average and the standard deviation of the cumulative rewards over the M test iterations. Clearly, the method that shows a higher average reward is the one that delivers the best optimal trading policy for the analysed problem. The simulation parameters for each experiment are reported in Table 4.1.

The transition rate matrices for the time varying parameters are $A_\theta = \begin{bmatrix} -0.1 & 0.05 & 0.05 \\ 0.05 & -0.1 & 0.05 \\ 0.05 & 0.05 & -0.1 \end{bmatrix}$, $A_\kappa = \begin{bmatrix} -0.1 & 0.1 \\ 0.1 & -0.1 \end{bmatrix}$ and $A_\sigma = \begin{bmatrix} -0.1 & 0.1 \\ 0.1 & -0.1 \end{bmatrix}$.

Table 4.1: Simulation parameters for the three experiments and the for the DDPG algorithm.

DDPG parameters					
lr= 0.001	$\mu_{\text{inv}} = 1$	train eps. $N = 10,000$	b = 512	$I_{\max} = 10$	$\lambda = 0.05$
l = 5	$\ell = 1$	test eps. $M = 500$	$I_{\min} = -10$	$a = 100$	$\Delta t = 0.2$, $n = 2,000$
hid-DDPG		prob-DDPG		reg-DDPG	
$l_{\text{NN}} = 4$	$d_{\text{NN}} = 20$	$l_{\text{NN}} = 5$	$d_{\text{NN}} = 64$	$l_{\text{NN}} = 5$	$d_{\text{NN}} = 64$
Simulation parameters					
MC pmt.	θ	κ	σ	layers	hidd. nod.
θ_t	$\{0.9, 1, 1.1\}$	5	0.2	5	16
θ_t, κ_t	$\{0.9, 1, 1.1\}$	$\{3, 7\}$	0.2	5	20
$\theta_t, \kappa_t, \sigma_t$	$\{0.9, 1, 1.1\}$	$\{3, 7\}$	$\{0.1, 0.3\}$	6	20

Table 4.2: Parameters used in the GRU algorithm for the two-step approaches.

GRU parameters in the prob-DDPG and in the reg-DDPG two-step approach			
NN layers	$d_l = 5$	N train eps.	$10,000$
Hidden nodes	$d_h = 20$	W-ADAM lr	0.001
$W = 10$ prob-DDPG	($W = 50$ reg-DDPG)	Predict window	1
Linear layers parameters			
NN layers	5	Hidden nodes	64

Table 4.3: Parameters used in the GRU algorithm for the one-step approach.

GRU parameters in the hid-DDPG one-step approach			
NN layers	$d_l = 1$	Hidden nodes	$d_h = 10$
Look-back window	$W = 10$	W-ADAM lr	0.001
GRU parameters in the hid-DDPG one-step approach (θ_t, κ_t, σ_t MCs)			
NN layers	$d_l = 2$	Hidden nodes	$d_h = 10$
Look-back window	$W = 10$	W-ADAM lr	0.001

4.4.1 θ_t is a Markov chain

In the first experiment we test the performance of our approaches in the case when only θ_t while κ, σ are constant. In this case we allow the parameter of mean reversion to be $\theta_t \in \{\phi_1 = 0.9, \phi_2 = 1, \phi_3 = 1.1\}$. The results of this, as well as of the other, settings are summarized in Table 4.4 and in Figure 4.5.

Table 4.4: Average cumulative rewards and their standard deviation after $n = 2,000$ trades from the $M = 500$ test episodes of trading for each approach used and for each data generating process employed.

	θ_t MC		θ_t, κ_t MCs		θ_t, κ_t, σ_t MCs	
Model rewards	Ave.	Std.	Ave.	Std.	Ave.	Std.
hid-DDPG	15.70	1.39	8.08	1.54	1.29	3.49
reg-DDPG	8.69	0.60	2.95	0.30	-0.07	0.11
prob-DDPG	25.65	3.35	15.59	3.83	4.51	3.75

By considering first the one step *hid-DDPG* approach, we notice that feeding the hidden states of a GRU network, trained along with the DDPG algorithm, provides the actor network with a policy capable of generating, on average, positive cumulative rewards. Therefore, GRU hidden states provide valuable information to the DDPG algorithm, helping the Actor network in finding policies which exploit the coded information of the parameter state.

Considering the rewards obtained by the two step *prob-DDPG* approach, which first learns to map signal to posterior probabilities and only afterwards learns the optimal trading, we notice that it generally outperforms the one-step approach in terms of cumulative rewards. This is not surprising, since the two-step approaches have the great advantage of separating the learning from the optimization phase, while in the one-step setting these tasks are performed contemporaneously. In practical setting, the two-step procedure might not be feasible as many training episodes are necessary for this task.

However, the two-step approach does not always outperforms the one-step algorithm. In fact, considering the two-step *reg-DDPG* approach, where we first train a GRU network to predict the value of the signal S at time $t + 1$, we can notice that the average rewards although still positive, are significantly lower than those obtained with the one-step *hid-DDPG*. We postulate that this effect on rewards is due to the difficult task of predicting the exact next value for the signal without information about the regimes of mean reversion.

We now investigate in more details the action chosen by the different approaches. The actions chosen by the Actor network are reported in the left column of Figure B.1a, B.1b and B.1c in the Appendix A, for the three approaches used.

Considering the first column of Figure B.1a which corresponds to the *hid-DDPG* approach under a Markovian evolution of θ_t, the inventory rebalancing actions, $q_t = I_{t+1} - I_t$, appear highly clustered, indicating a strong preference for a limited set of trading decisions.

Some actions q_t appear counter-intuitive with respect to the levels of the observed inventory

and signal. For instance, the policy sometimes suggests selling when the signal S_t is low and the inventory I_t is positive. The opposite also occurs: buying when S_t is high and I_t is low. In both cases, some rebalancing decisions still manage to exploit signal values. However, interpreting the resulting trading strategy is difficult, as the GRU hidden states used as input features encode relevant information about the signal process, but in a form that is largely non-interpretable.

Focusing on the *prob-DDPG* policy in the first column of Figure B.1c, we observe that buy and sell actions are predominantly concentrated on the left and right sides of the heatmap, respectively. This points out to the fact that the agent is more likely to buy when the trading signal S_t is low, and to sell when it is high. In the scatter plots on the last row, buy and sell decisions tend to overlap at similar levels of inventory I_t, suggesting a more nuanced policy that takes into account the underlying regimes. The intuition is that providing the posterior probabilities of the latent state θ_t allows the agent to better condition its actions on the most likely regime, the current signal S_t, and the inventory level I_t. This is illustrated in greater detail in the first row of Figure B.2, where actions are shown as a function of S_t and I_t, conditional on the posterior distribution. The shades of yellow represent the probability that S_t is mean-reverting to a specific long-run average, as estimated by the GRU network trained in the first step. The colour of the border of each point indicates whether the action q_t corresponds to a buy or a sell, given the current inventory level I_t and signal S_t.

Since the trading action depends heavily on the expected mean-reversion level of S_t, the posterior distribution vector Φ_t provides crucial information. For instance, looking at the state probability of θ_t in the figure we see how, even if the level of the signal is low, when the agent is long on the signal and the probability of mean reverting to $\theta_t = 0.9$ is high, the agent is going to mostly sell. Conversely, in the lower part of the plot, where the agent is likely short and the signal is expected to revert upwards, the policy favours buying.

Finally, the reg-DDPG policy (first column of Figure B.1b) appears to be less clustered around very few values for I_t, the sell/buy actions seem more reasonable with respect to the various combinations of signal and inventory, meaning that when the signal shows low values and the inventory holdings are low as well, the policy found suggests to the Actor to buy taking advantage of the estimated next value of the signal with respect to the actual value of the inventory holdings, contrary to the hid-DDPG case.

4.4.2 θ_t , κ_t are Markov chains

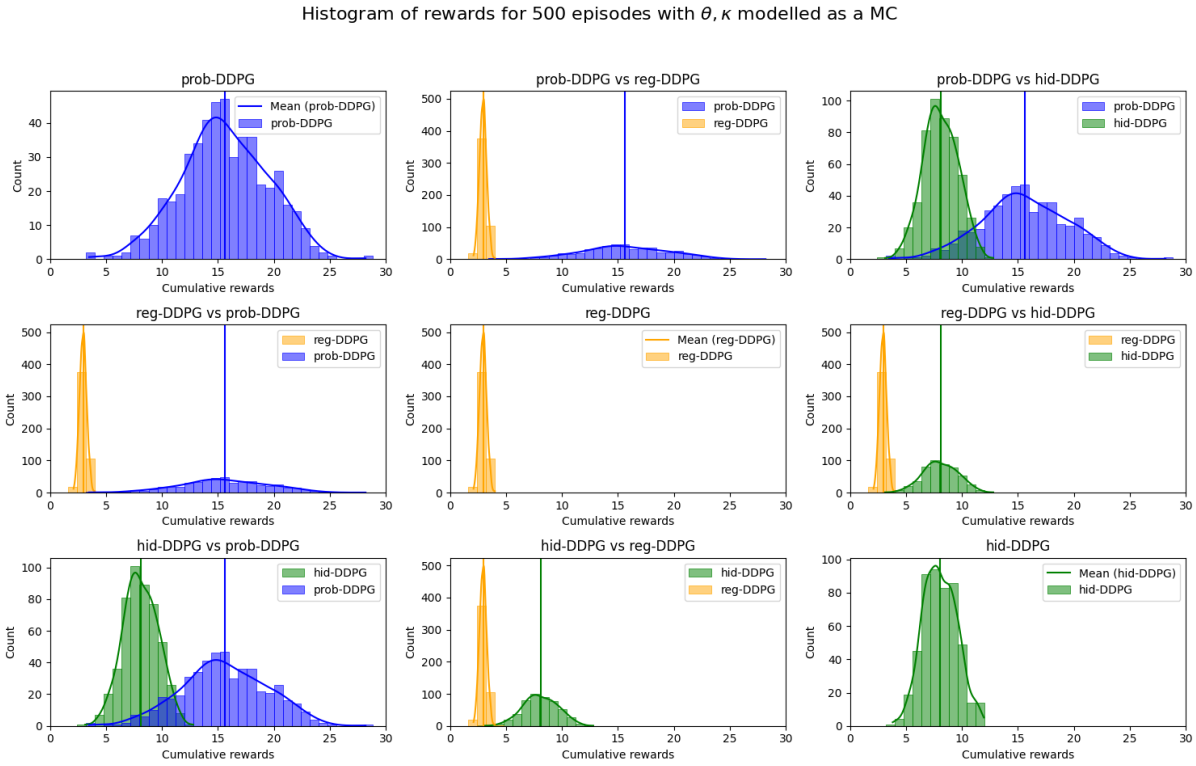


Figure 4.6: Comparison of the cumulative rewards for the different DDPG approaches when θ_t and κ_t follow a Markov chain.

We now consider the case where the speed of mean reversion, κ_t, in Equation (4.1) is also modelled as a Markov chain, which is independent of the Markov chain governing the dynamics of θ_t. In this setting, employing the *hid-DDPG* approach delivers rewards that are positively distributed across the test episodes, as reported in Figure 4.6. Table 4.4 shows that the average reward now is lower than the one obtained with the dynamics where only θ_t follows a Markov chain. This is expected since now the parameter dynamics is more complicated. Considering the *reg-DDPG* approach, it can still be noticed that the rewards, although positive, lie on average below those scored by the *hid-DDPG* approach. This is again due to the more complex signal's dynamics that the agent is trading, and as before the difficulty of predicting the next trading signal value persists, and appears to be stronger in this case due to the non-constant nature of the κ_t parameter. When the *prob-DDPG* approach is used, the algorithm scores lower average rewards than in the previous case, that was less complex, but the average rewards are still higher than those obtained with the other two approaches, as it can be noticed in Figure 4.6.

Analysing the actions chosen by the Actor network per level of inventory I_t and trading signal S_t in the second column of Figure B.1a, B.1b and B.1c for the three approaches used it can still be noticed how the policies do change with respect to the type of information provided

to the DDPG part of the approaches. In fact, it is evident how, for the *reg-DDPG* and for the *hid-DDPG*, in Figure B.1b and in Figure B.1a respectively, the actions employed and the subsequent new levels of inventory seem to be clustered around some of the levels of inventory I_t, thus the inventory holdings do not span much over the allowed inventory domain, $[I_{\max}, I_{\min}]$. We postulate that this partly leads to the lower rewards obtained, as it can be noticed in Figure 4.6. When looking at the policy chosen by the *prob-DDPG* in Figure B.1c, it can be seen how the inventory holdings are more sparse, meaning that providing information on the data generating process' mean reversion level under the form of posterior probabilities on the regimes is more beneficial. In this case we can associate an interpretation to the policy adopted by the RL agent. In fact, in the second row of Figure B.2 it can be understood how the policy of buys and sells q_t does change both with respect of of S_t and I_t and with respect of the probability of being mean reverting to either of the regimes, now with two different speeds of mean reversion κ_t. Clearly, in neither of the approaches we do provide information on the speed of mean reversion, but it seems to be more beneficial to provide circumstantial information on the θ_t values if κ_t follows a Markov chain, rather than providing hidden states or estimates for the next value of the trading signal, from both a cumulative reward and an interpretability point of view.

4.4.3 θ_t , κ_t and σ_t are Markov chains

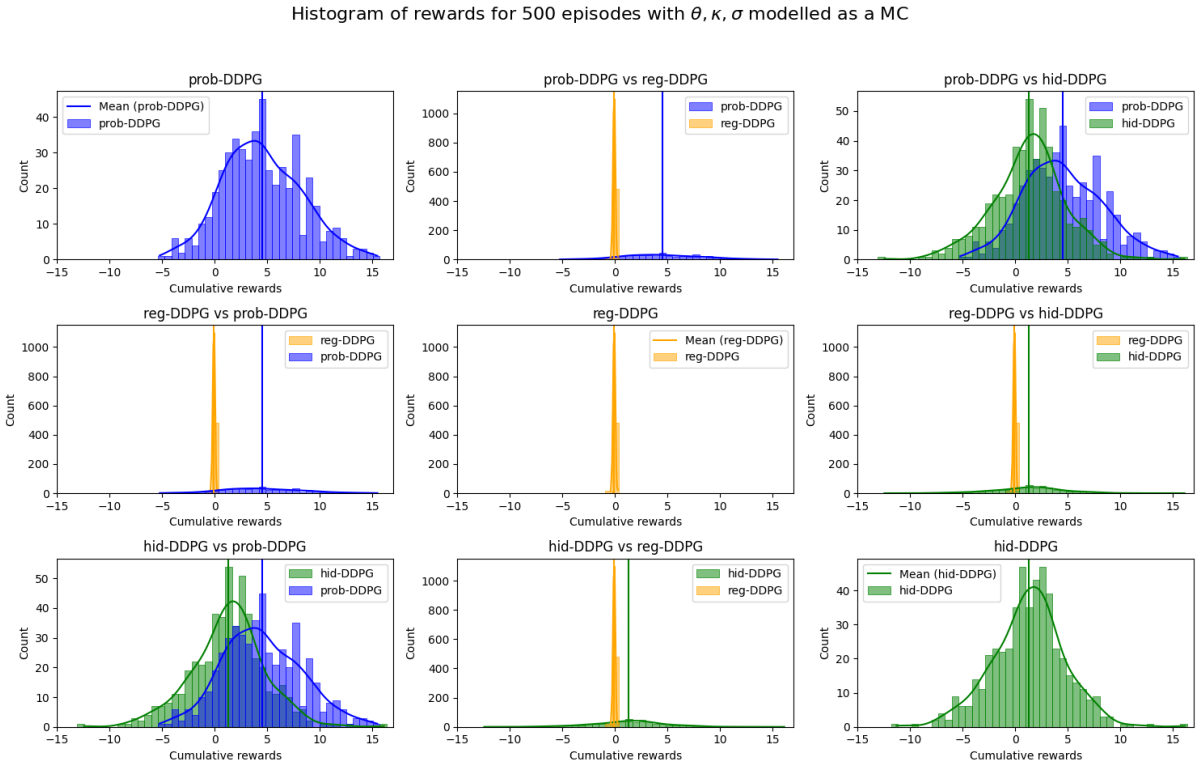


Figure 4.7: Comparison of the cumulative rewards for the different DDPG approaches when θ_t, κ_t, and σ_t follow a Markov chain.

In the last and most complex experiment we allow also the volatility of the signal to transition between two regimes, a high and a low volatility regime, as reported in Table 4.1. The Markov chain that models the transition between the two regimes is independent from those of the other parameters θ_t and κ_t.

From the average cumulative rewards reported in Table 4.4 we see that, once again, the *prob-DDPG* outperforms the other two approaches.

Overall, we notice how average rewards of the *reg-DDPG* approach are near zero, although slightly negative. We postulate that the inefficiency of a DDPG policy based off of information on the next signal level comes from the lower informative content that such an estimate can provide in a complex environment. The standard deviation for the rewards in this case is very low, but this depends on the policy adopted by the agent trained with such a two-step approach. Comparing both the results reported in Table 4.4 and in Figure 4.7 with the policy in Figure B.1b, we see that actions q_t, compared with $I_{\max}$, are small and that inventory levels I_t change very little as the signal S_t changes. This explains the small reward variability: the agent trades very cautiously, which reduces volatility in outcomes but also limits the overall rewards.

Moving to the *hid-DDPG* approach, Table 4.4 and Figure 4.7, we see that *hid-DDPG* underperforms compared to the *prob-DDPG* while over performing with respect to the *reg-DDPG* in terms of rewards. This is to be expected, as the environment has now become even more complex. The trading policy employed by the DDPG agent is still on average positive, meaning that the hidden states of the GRU network provided to the DDPG part of the approach are still informative, although less interpretable. As shown in Figure B.1a, the actions chosen by the agent are of a higher magnitude if compared to those of the two other approaches, this translates into a trading policy that still produces positive cumulative rewards on average, although lower than those obtained using the *prob-DDPG*.

Turning to the *prob-DDPG* approach,[6] the average cumulative rewards form the trading policy are indeed lower in this more complex environment if compared with those obtained by the same approach in the other two simulative set-ups, however, they still remain higher than the other two approaches in this particular set-up, as reported in Table 4.4. Interestingly, the standard deviation does not change much over the course of the three different experiments for this approach. Clearly, similar results would hold if the parameters would move together, uniquely according to the transition matrix A_θ, with parameters $\theta_t = \{0.9, 1, 1.1\}, \kappa_t = \{3, 5, 7\}$ and $\sigma_t = \{0.1, 0.2, 0.3\}$; i.e. the ones used over all the experiments.[7] Therefore, once again, we conclude that first learning the probability of the regime helps the DDPG algorithm choose policies with higher rewards. Due to the interpretability of this learnt feature, we can investigate how the *prob-DDPG* approach takes advantage of the θ_t regime knowledge and chooses buy/sell actions not just based on the current level of S_t and I_t but on the regime probabilities, as displayed in Figure B.2 and in Figure B.1c.

4.5 Pair trading application

Having seen how the three approaches behave with synthetic data, we now turn to testing them on real market data. We focus on the problem faced by an agent whose goal is to trade two co-integrated assets; the classical *pair trading* framework. We thus train a RL agent, using real high-frequency mid-price data, to trade according to a co-integration signal. The signal is, roughly speaking, the process made by a portfolio of the two assets under consideration. The goal is to assess which method produces higher rewards when exposed to real market

[6] In this case we have set the look-back window to $W = 20$.

[7] In this case, for the *prob-DDPG* approach the average reward would be 3.80 with a standard error of 0.1829.

conditions. At the same time, it allows us to investigate how the balance between flexibility and the specificity of the information provided to the agent, affects the overall performance of the RL algorithms used.

For these experiments, we apply a methodology similar to the one in Cartea et al. [2015] to obtain the parameters for the model, and Chan [2013] to obtain the benchmark Z-score strategy.

4.5.1 Dataset, preliminary data analysis and market model

We consider two assets traded on the NASDAQ between the 29th of August 2025 and the 5th of September 2025, the Intel (INTC) stock and the Merril Lynch semi-conductors ETF (SMH) whose holdings are around 20% composed by INTC shares.

We use high frequency observations of the limit order book for both the assets, downloaded from the LOBSTER database, we consider data up to the first level of the order book and we select only trade events, so that each element considered in the time series represents a transaction. Then, calculate the mid-price and filter the mid-prices at a frequency of 1 second, such that we have a clean dataset that displays prices at even intervals of time. Both time series are displayed in Figure 4.8.

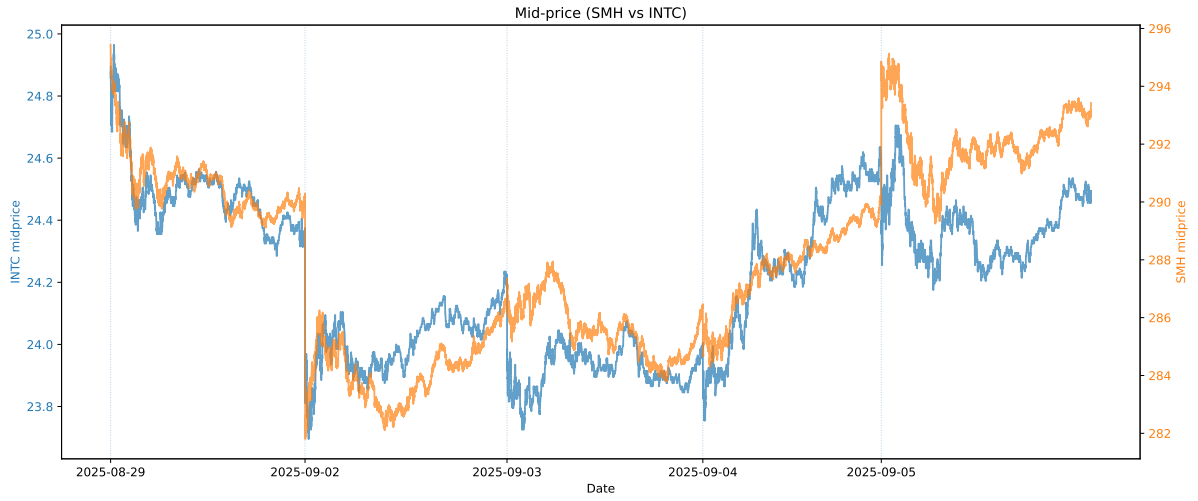


Figure 4.8: Mid-prices for SMH and INTC, prices are intended in US dollars.

We test the two assets for co-integration using Johansen test statistics, the results are reported in Table 4.6. We see that both tests indicate strong evidence of at least one co-integrating vector. Meaning that there exists at least one stationary linear combination denoted with:

$$\tilde{S}_t = \alpha_1 S_t^{\text{smh}} + \alpha_2 S_t^{\text{intc}} \tag{4.16}$$

Table 4.5: Johansen Co-integration Test (5% level)

r	Trace Stat	Crit	Reject	Max-Eig Stat	Crit	Reject
0	27.456	18.399	Yes	19.222	17.148	Yes

Table 4.6: Trace statistic, maximum Eigenvalue statistic and their critical values of the Johansen test for co-integration. The H0 is no co-integrating vector.

Notice that the process $\tilde{S}_t$ is a linear combination of the assets prices. We assume that the dynamics for this portfolio can be modelled as:

$$d\tilde{S}_t = \tilde{\kappa}(\tilde{\theta} - \tilde{S}_t)\, dt + \tilde{\sigma}\, dW_t, \tag{4.17}$$

so that the linear combination of the two time series of mid-prices is stationary. We use as a portfolio $\tilde{S}_t$ where the amount of assets held are fixed at α_1, α_2, given by the co-integrating coefficients. We adjust the level of stocks held in the co-integrating portfolio but keep the ratio fixed at a ratio of α_1, α_2. The book-value of the portfolio at every time-step t is therefore

$$BV_t = I_t \left(\alpha_1 \, S_t^{\text{intc}} + \alpha_2 \, S_t^{\text{smh}} \right) .$$

First, we retrieve the co-integrating coefficients. To this end, we fit a Vector AutoRegressive (VAR) model of the form

$$\Delta \mathbf{S}_t = \vec{A} + \mathbf{B} \Delta \mathbf{S}_{t-1} + \vec{\varepsilon}_t \, .$$

That is a discrete time version of the model in Equation (4.1) where $\mathbf{S}_t = \left[S_t^{\text{smh}}, S_t^{\text{intc}} \right]$ are the mid-prices of INTC and SMH, $\vec{A}$ is a vector of constants, $\mathbf{B}$ is matrix of constants and $\vec{\varepsilon}_t$ is a vector of white noise, $\Delta \mathbf{S}_t$ denote the mid-prices change at each time-step. The estimated parameters are reported in Table 4.7.

Table 4.7: VAR Model Coefficients at 1% significance.

	B		A
	$\Delta S_{t-1,\text{SMH}}$	$\Delta S_{t-1,\text{INTC}}$	
$\Delta S_{t,\text{SMH}}$	$0.999^{(***)}$	0.00001	0.021
$\Delta S_{t,\text{INTC}}$	0.0007	$0.996^{(***)}$	0.003

From these estimates, we may estimate the coefficients in Equation (4.17), and thus obtain the co-integrating coefficients for $\tilde{S}_t$, assuming $\Delta t = 1$.

$$
\begin{aligned}
\kappa &= (\mathbb{I} - \mathbf{B})/\Delta t = \begin{bmatrix} 1.40 \cdot 10^{-04} & -1.52 \cdot 10^{-05} \\ -7.78 \cdot 10^{-04} & 3.05 \cdot 10^{-04} \end{bmatrix}, \\
\tilde{\theta} &= \kappa^{-1}\vec{A}\Delta t = \begin{bmatrix} 287.77 \\ 24.183 \end{bmatrix}, \\
\kappa &= U^{-1}\Lambda U \\
\Lambda &= \begin{bmatrix} 8.67 \cdot 10^{-05} & 0 \\ 0 & 3.59 \cdot 10^{-4} \end{bmatrix}.
\end{aligned}
\tag{4.18}
$$

Having estimated the parameters that describe the dynamics outlined in Equation (4.17) we now find two possible candidates for $\tilde{S}$:

$$
\tilde{S} = U^{-1}\mathbf{S} = \begin{bmatrix} -2.960 & -0.205 \\ 2.856 & -0.804 \end{bmatrix} \mathbf{S}\,. \tag{4.19}
$$

Where $\mathbf{S}$ is as before. The coefficients in the second row of U^{-1} are the co-integrating coefficients that correspond to the highest eigenvalue in the matrix Λ. This means that a portfolio with these coefficients will be the most exposed to mean reversion. Thus, the portfolio we will focus on is

$$
\tilde{S}_t = 2.856 \times S_t^{\text{smh}} - 0.804 \times S_t^{\text{intc}} \tag{4.20}
$$

and use $\tilde{S}_t$ as the asset to be traded, keeping fixed the weights in Equation (4.20).

We divide the time series for $\tilde{S}_t$ into two sets: a training set (Figure 4.9) and a testing set (Figure 4.10). The training set is composed by the observations up to the 4th of September and the testing set are the observations on the 5th of September. In this way, we train the agent on almost a week of data and then test the trading performance over one day, the 5th of September. We normalise the $\tilde{S}_t$ series using min-max normalization over the training set[8] and we then estimate two possible regimes for $\tilde{S}_t$ over the training set, following the procedure outlined in Hamilton [1989], Chapter 22. Following this procedure we find that the two regimes are $\theta_1 = 0.2216$ and $\theta_2 = 0.5658$. Figure 4.9 reports the most probable regime at each time-step.

[8]The reason is twofold, on the one hand with normalised data Hamilton's procedure is more numerically stable, on the other hand we will feed the normalised series to the algorithms that require normalise data.

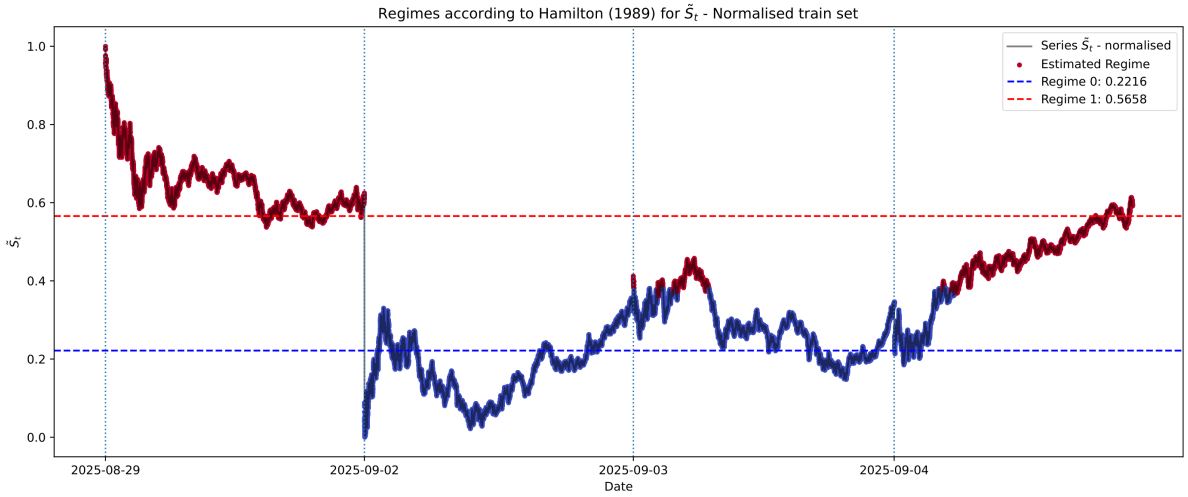


Figure 4.9: Normalised time series for the training phase of the co-integrated portfolio $\tilde{S}_t$ and the most probable regimes at each time step.

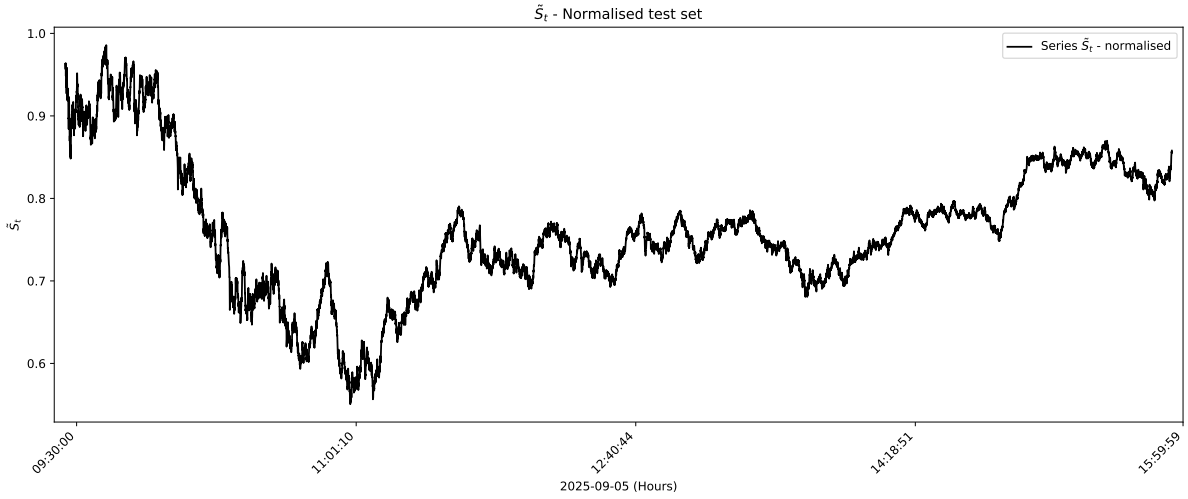


Figure 4.10: Normalised time series for the testing phase of the co-integrated portfolio $\tilde{S}_t$.

4.5.2 Trading experiments

Based on the results on synthetic data we now employ the approaches that performed the best in the case of θ_t, κ_t σ_t modelled as Markov chains, as that was the most complex environment. We thus tackle the pair trading problem with both the *hid-DDPG* approach and the *prob-DDPG* approach. The parameters used for the simulations are reported in Table 4.8 for the DDPG part of the model, while for GRU part of the algorithm we keep the parameters as in the case for θ_t, $\kappa_t \sigma_t$ modelled as Markov chains for both of the approaches. During training the $\{\tilde{S}_u\}_{u=t}^{t-W}$ fed to the GRU and both to Actor and the Critic networks are randomly picked from the training set and have, as in the simulated case above, length W.

The reward for the DDPG phase of the algorithm is the book-value of the portfolio with a

prob-DDPG parameters		
train eps. $10,000$	lr $= 0.001$	batch $= 64$
$\gamma = 0.999$	$I_{\max} = 10$	$\lambda = 0.05$
$\theta = \{0.2216, 0.5658\}$	$W = 100$	layers $= 6$
hidd. nod. $= 64$	train obs. $= 72,700$	test obs. $= 19,789$

Table 4.8: Parameters for the prob-DDPG algorithm.

transaction cost λ, thus similarly to the simulated experiments, the reward is

$$r_t = \Delta BV_t - \lambda |I_t - I_{t-1}| \tag{4.21}$$

where, as before, I_t is the action decided by the DDPG Actor network. We use as benchmark the rolling Z-score trading strategy, which is a strategy where the agent accumulates units of $\tilde{S}_t$ proportional to the negative rolling Z-Score of the portfolio itself. For the rolling Z-score computation we use the same window $W = 100$ as in the GRU algorithm. Thus, we compare the *prob-DDPG* and the *hid-DDPG* and the rolling Z-score strategies as reported in Figure 4.11 and in Figure 4.12.

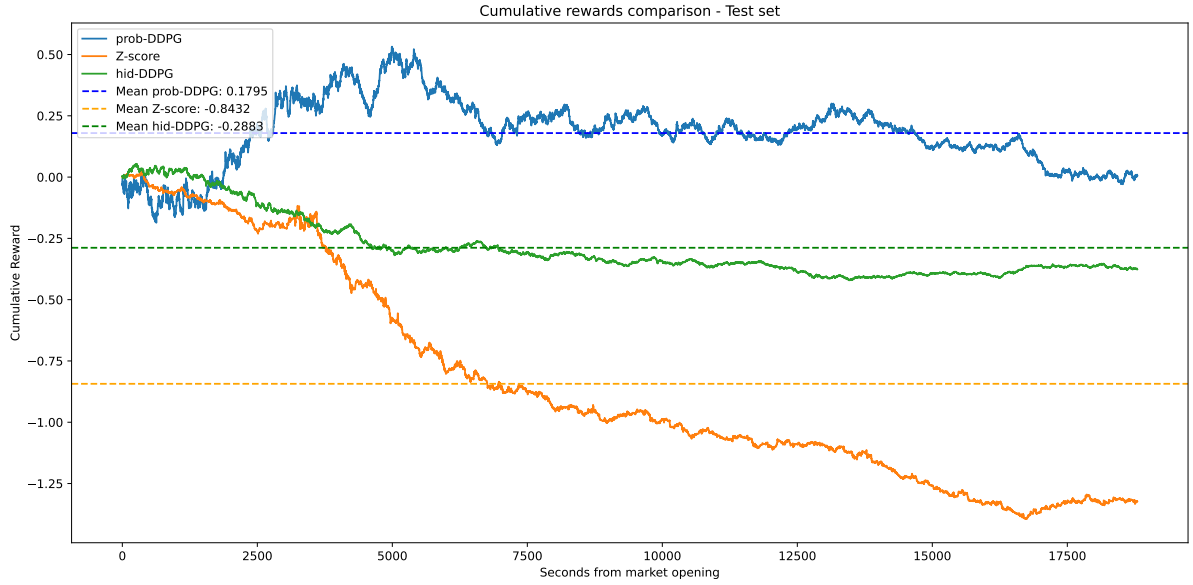


Figure 4.11: Comparison of realised cumulative rewards for the *hid-DDPG*, *prob-DDPG* and rolling Z-score strategy on the testing set.

As in Figure 4.11, the *prob-DDPG* strategy outperforms the others in the test set by achieving the highest average reward, ultimately resulting in the highest cumulative rewards through the selected policy in this straightforward trading experiment. Indeed, Figure 4.12 shows that while average cumulative rewards are positive with the *prob*-DDPG approach, they are negative

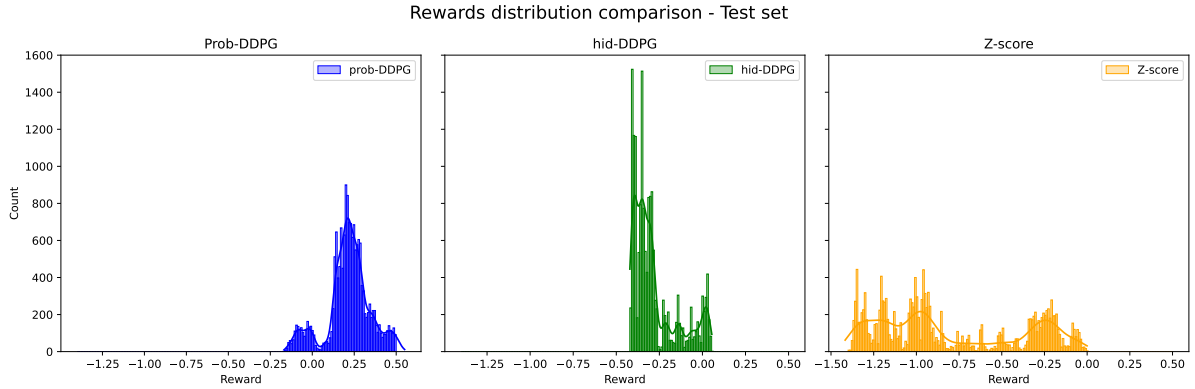


Figure 4.12: Histograms of the realised cumulative rewards for the three methods used.

for the other methods. Although the cumulative rewards for the *hid*-DDPG case are higher than those from the naive Z-score strategy, they remain below those of the *prob*-DDPG method. The average rewards, along with their standard deviations, are detailed in Table 4.9.

Table 4.9: Average cumulative rewards and their standard deviation for the approaches used. Rewards are calculated over one day of trading at a frequency of 1 second.

Cum. Rew.	prob-DDPG	hid-DDPG	Z-score
Average	0.1795	−0.2883	−0.8432
Std. Dev.	0.1329	0.1282	0.4341

These findings are consistent with the outcomes from the numerical experiments discussed earlier. Consequently, leveraging the probability of transitioning into either a lower or higher regime for the portfolio $\tilde{S}_t$ significantly helps the RL algorithm in determining the most suitable trading policy, especially when dealing with a potentially intricate and unknown data-generating process. This two-step approach enhances the interpretability of these algorithms, which are typically known as *black boxes* and often function as data-intensive oracles that can yield inefficient solutions, as observed with the *hid*-DDPG algorithm applied to this issue, unless equipped with background information about the problem at hand. Therefore, we can conclude that the information provided to the agent, and especially the way in which the information is used, does matter when training a RL agent and is paramount for the effective and successful application of model-agnostic algorithms like RL architectures, even with high-frequency data with strong simplifying assumptions.

4.6 Conclusions

In this paper, we have developed three different DDPG algorithms to approximately solve the optimal trading problem when the trading signal is mean-reverting, and the parameters of the data-generating process are modelled as independent Markov chains. Given the sequential nature of the trading signal, we have chosen to use Gated Recurrent Units (GRUs), a specific type of recurrent neural network, to effectively process the time-dependent structure of the signal data used by the agent to trade. The GRU structure and the output of these networks is then used in three different ways to train a RL agent, modelled using the DDPG framework.

The trading signal is modelled as an Ornstein-Uhlenbeck process, which, in the simplest case, reverts to three distinct long-run regimes. The transitions between these regimes are governed by a Markov chain. The agent approximately solves the trading problem by leveraging information about the state of the world, which encompasses the current value of the trading signal, the inventory level, and posterior probability estimates of the most likely regime. Alternatively, the agent may rely on an estimate of the next trading signal value - computed in a previous step - or directly use hidden states extracted from a GRU network trained in conjunction with the DDPG Actor-Critic networks, thus training the GRU network along with the RL algorithm.

We tested the agent's performance in increasingly more complex environments where, along with the mean reversion parameter, both the speed of mean reversion and the volatility of the trading signal were governed by independent Markov chains.

The numerical results show that the agent, when equipped with posterior probability estimates of the mean reversion regimes (which it learns in a first stage), achieves the highest trading rewards. This suggests that providing structured and interpretable information to a RL algorithm significantly enhances its ability to approximate solutions to inter-temporal optimisation problems.

Finally, we find that these conclusions remain valid when the agent faces trading problems with real market data, when dealing with a pair trading problem for two co-integrated assets.

We can conclude that when tackling optimisation problems which do not result in closed-form solutions, the quality of information provided to the RL algorithm is crucial for the quality of the solution. Although all three approaches result in a trading policy that, on average, yields either positive or, in the worst case, close to zero average rewards across all scenarios considered, the approach that consistently works best is to first gather insights into the dynamics of the

data-generating process and only then incorporate this information into the features that the agent uses in the RL algorithm.

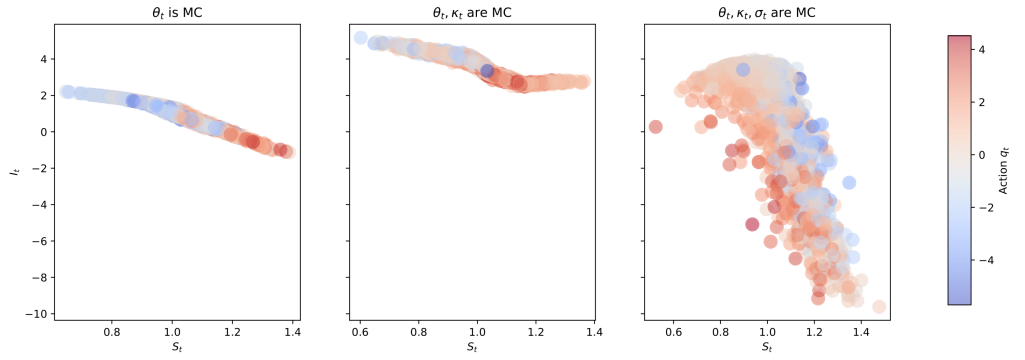
Interestingly, the type of information fed into the algorithm also appears to be a key factor. Specifically, providing estimates of the next signal values does not yield any significant improvement, suggesting that it is more effective to supply general information about the process - as in the *hid-DDPG* approach - while ensuring that this information remains closely related to the actual parameters governing the underlying dynamics, as in the *prob-DDPG* approach. Indeed, the latter proves to be the most effective, delivering the highest rewards when tested on both synthetic and real data.

Future research directions to investigate how, and by how much, the quality and the type of information affects the learning process, the resulting interactions, and the eventual equilibria that might emerge in a similar setup but in a multi-agent type of problem.

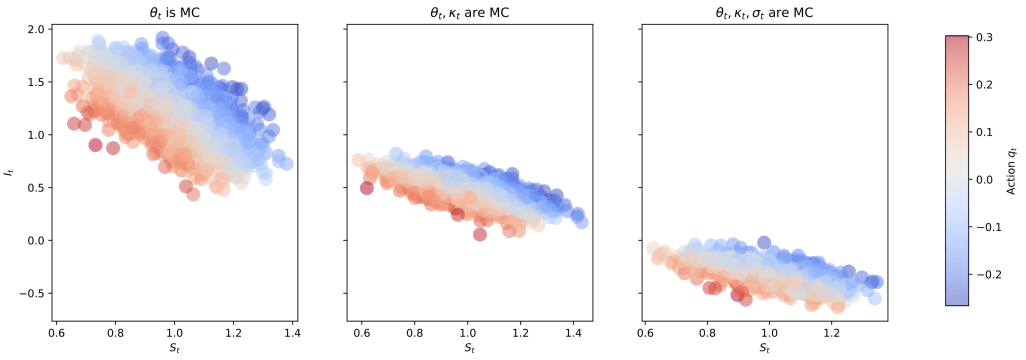
Appendix B

Supplementary materials

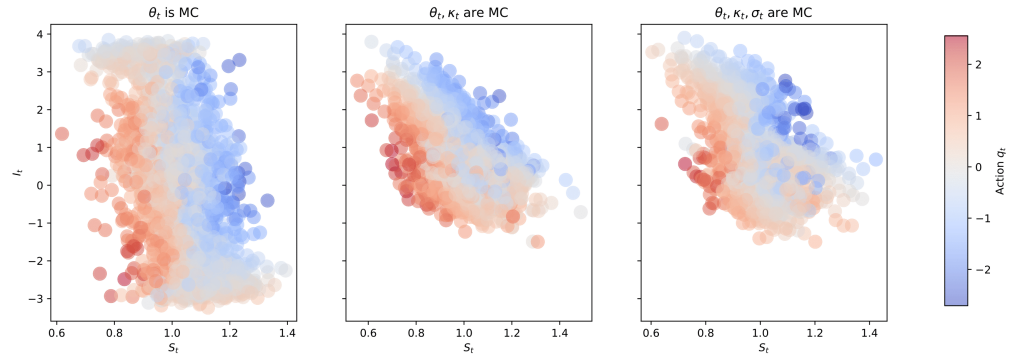
B.1 Figures



(a) hid-DDPG



(b) reg-DDPG



(c) prob-DDPG

Figure B.1: Value of buy and sell actions q_t per level of inventory I and signal S_t for the different approaches used.

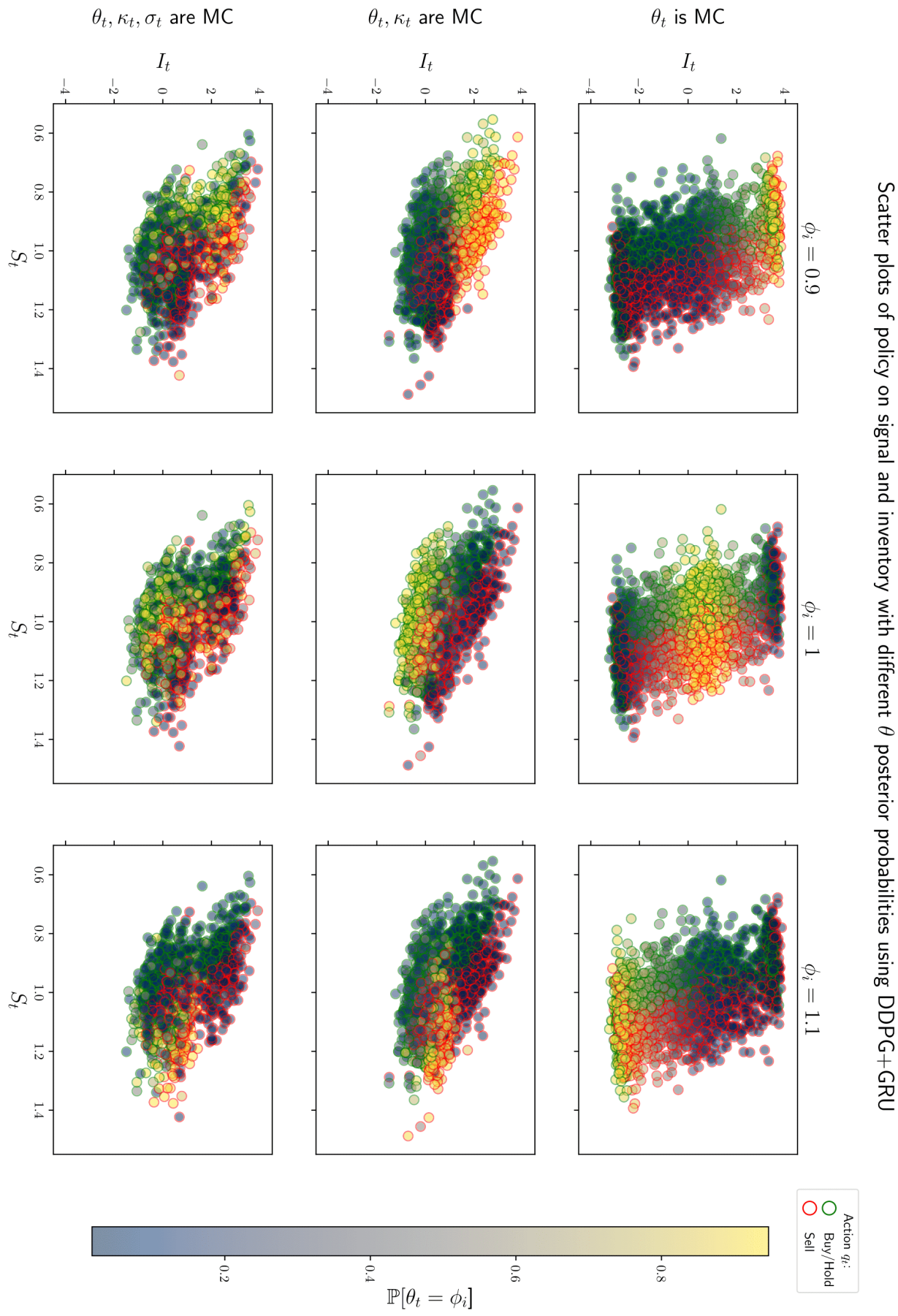


Figure B.2: Policies chosen by the DDPG agent per level of inventory, price signal S_t and posterior probability of mean reversion θ_t, when *prob-DDPG* approach is used.

B.2 Algorithm

Algorithm 6 Deep Deterministic Policy Gradient (DDPG) for optimal trading

Require: W look-back window, ℓ, l and N iteration

Initialise actor network $\pi(\mu_\pi)$ and critic network $Q(\mu_Q)$ with random weights μ_π and μ_Q
Initialise target network and $Q_{\text{tgt}}(s, a|\mu_{Q_{\text{tgt}}})$ with weights $\mu_{Q_{\text{tgt}}} \leftarrow \mu_Q$
for m = 1 , . . . , N **do**
    Obtain batches of length b of $S_{[0,W+2]}$ signal time series and random levels of Inventory I.
    Initialise $\varepsilon = 1$
    procedure GRU
        Train the GRU network if hid-DDPG;
    end procedure
    procedure Update Critic
        for ℓ **do**
            Receive initial observation batch $\mathbf{F}$
            Pass $(\{S_u\}_{u=t}^{t-W}$ to GRU pre-trained and obtain $\{\Phi_k\}, k = 1, 2, 3)$ (if prob-DDPG);
            Pass $(\{S_u\}_{u=t}^{t-W}$ to GRU pre-trained and obtain $\tilde{S}_{t+1}$ (if reg-DDPG);
            Pass $(\{S_u\}_{u=t}^{t-W}$ to GRU to train if (hid-DDPG);
            obtain $\mathbf{G}$;
            Select inventory $I = \pi(\mathbf{G}|\mu_\pi) + \mathcal{N}(0, \varepsilon)$ according to the current policy and noise;
            Execute action I and observe reward r; and next state $\mathbf{G}'$;
            Observe next states $\mathbf{G}'$;
            Set $y^{(i)} = r^{(i)} + \gamma Q_{\text{tgt}}(\mathbf{G}'^{(i)}, \pi(\mathbf{G}'^{(i)}|\mu_\pi)|\mu_{Q_{\text{tgt}}})$;
            $\mathcal{L}_1 = \frac{1}{b}\sum_i^b (Q(\mathbf{G}^{(i)}, I^{(i)}|\mu_Q) - y^{(i)})^2$;
            Set $\mu_Q = \mu_{Q_{\text{tgt}}}$;
        end for
    end procedure
    procedure Update Actor
        for l **do**
            Receive initial observation batch $\mathbf{F}$;
            Pass $(\{S_u\}_{u=t}^{t-W}$ to GRU pre-trained and obtain $\{\Phi_k\}, k = 1, 2, 3)$ (if prob-DDPG);
            Pass $(\{S_u\}_{u=t}^{t-W}$ to GRU pre-trained and obtain $\tilde{S}_{t+1}$ (if reg-DDPG);
            Pass $(\{S_u\}_{u=t}^{t-W}$ to GRU to train if (hid-DDPG);
            Obtain $\mathbf{G}$;
            Get $I = \pi(\mathbf{G}|\mu_\pi)$;
            Minimise $\mathcal{L}_2 = -\frac{1}{b}\sum_{i=1}^b (Q(\mathbf{G}^{(i)}, \pi(\mathbf{G}^{(i)}|\mu_\pi)|\mu_Q)$ through gradient descent over

$$\nabla_{\mu_\pi}\mathcal{L}_2 = \frac{1}{b}\sum_{i=1}^{b}\left[\nabla_a Q(\mathbf{G}^{(i)}, a^{(i)}|\mu_Q)|_{a^{(i)}=\pi(\mathbf{G}^{(i)}|\mu_\pi)}\nabla_{\mu_\pi}(\mathbf{G}^{(i)}|\mu_\pi)\right]$$

        end for
    end procedure
    Decrease ε;
end for

Part II

Execution problem in a multiple agent setting

Chapter 5

Execution game with deep reinforcement learning

> «I want to see you not through the machine» said Kuno. «I want to speak to you not through the wearisome machine».
>
> – E.M. Forster, *The Machine stops* (1909).

The contents of this chapter are available as a preprint in Lillo and Macrì [2026], and are forthcoming in Annals of Operations Research.

Abstract The use of reinforcement learning algorithms in financial trading is becoming increasingly prevalent. However, the autonomous nature of these algorithms can lead to unexpected outcomes that deviate from traditional game-theoretical predictions and may even destabilize markets. In this study, we examine a scenario in which two autonomous agents, modelled with Double Deep Q-Learning, learn to liquidate the same asset optimally in the presence of market impact, using the Almgren-Chriss (2000) framework. Our results show that the strategies learned by the agents deviate significantly from the Nash equilibrium of the corresponding market impact game. Notably, the learned strategies exhibit supra-competitive solution, which might be compatible with a tacit collusive behaviour, closely aligning with the Pareto-optimal solution. We further explore how different levels of market volatility influence the agents' performance and the equilibria they discover, including scenarios where volatility differs between the training and testing phases.

5.1 Introduction

The increasing automation of trading over the past three decades has profoundly transformed financial markets. The availability of large, detailed datasets has facilitated the rise of algorithmic trading—a sophisticated approach to executing orders that leverages the speed and precision of computers over human traders. Using diverse data sources, these automated systems have revolutionized the way trades are conducted. In 2019, it was estimated that approximately 92% of the trading in the foreign exchange (FX) market was driven by algorithms Kissell [2020]. The rapid advancements in Machine Learning (ML) and Artificial Intelligence (AI) have significantly accelerated this trend, leading to an increasing adoption of autonomous trading algorithms and, as argued in Azzutti et al. [2021], this poses problems from a legislative point of view when it comes to deciding whether and how regulators can hold accountable firms for malpractices due to algorithms' misuse. In the landscape of autonomous trading systems, increasing interest, by both practitioners and academics alike, is being shown on those based on Reinforcement Learning (RL), especially deep RL. Deep RL algorithms differ from traditional supervised learning models. Instead of being trained on labelled input/output pairs, RL algorithms explore a vast space of potential strategies, learning from past experiences to identify those that maximise long-term rewards. From a theoretical standpoint, the problem of learning optimal policies under unknown dynamics has been extensively studied in the context of Markov Decision Processes, where adaptive policies with asymptotically optimal regret have been derived, see for instance Burnetas and Katehakis [1997]. This approach allows for the continuous refinement of trading strategies, further enhancing the efficiency and effectiveness of automated trading.

The flexibility of deep RL comes with significant potential costs, particularly due to the opaque, black-box nature of these algorithms. This opacity can lead to unexpected outcomes that may destabilize the system they control—in this case, financial markets—through the actions they take, such as executing trades. The complexity and risk increase further when multiple autonomous agents operate simultaneously within the same market. The lack of transparency in RL algorithms may result in these agents inadvertently learning joint strategies that deviate from the theoretical Nash equilibrium, potentially leading to unintended market manipulation. Among the various risks, the emergence of collusion is particularly notable. Even without explicit instructions to do so, RL agents may develop cooperative strategies that manipulate the market, a phenomenon known as *tacit collusion*. This type of emergent behaviour

is especially concerning because it can arise naturally from the interaction of multiple agents, posing a significant challenge to market stability and fairness.

In this paper, we investigate the equilibria and the emergence of supra-competitive equilibria and their implications in terms of tacit collusive behaviour in a market where two autonomous agents, modelled by deep RL, engage in an optimal execution task. Although tacit collusion in RL settings has been explored in areas such as market making (as reviewed below), much less is known about how autonomous agents behave when trained to optimally trade a large position. We adopt the well-established Almgren-Chriss framework first introduced in Almgren and Chriss [2000], for modelling market impact, focusing on two agents tasked with liquidating large positions of the same asset. The agents are modelled using Double Deep Q-Learning (DDQL) and engage in repeated trading to learn the optimal equilibrium of the associated open-loop game.

For this specific setup, the Nash equilibrium of the game has been explicitly derived in Schied and Zhang [2017], providing a natural benchmark against which we compare our numerically derived equilibria. In addition, we explicitly derive the Pareto-optimal strategy and numerically characterize the Pareto-efficient set of solutions. Our primary goal is to determine whether the two agents, without being explicitly trained to cooperate or compete, can naturally converge to a non Nash equilibrium, when it existence and uniqueness is known.

Our findings reveal that the strategies learned by the RL agents deviate significantly from the Nash equilibrium of the market impact game. Across various levels of market volatility, we observe that the learned strategies exhibit seemingly tacit collusive behaviour, closely aligning with the Pareto-optimal strategy. Given that financial market volatility is time-varying, we also examine the robustness of these strategies when trained and tested under different volatility regimes. Remarkably, we find that the strategies learned in one volatility regime remain supra-competitive, even when adopted in various volatility conditions, underscoring the robustness of the training process.

Literature review. Optimal execution has been extensively studied in the financial literature. Starting from the seminal contributions in Bertsimas and Lo [1998] and in Almgren and Chriss [2000], many authors have contributed to extend the model's consistency with reality. Notable examples in this area are Cartea et al. [2015], Cartea and Jaimungal [2016], Casgrain and Jaimungal [2019] for optimal control perspective, Bouchaud et al. [2003, 2009], Gueant et al.

[2012], Gatheral et al. [2012], Obizhaeva and Wang [2013] for the transient impact extensions and Guéant and Lehalle [2015] for a limit order generalisation of the Almgren and Chriss framework. In its basic setting, the optimal execution problem considers just one agent unwinding or acquiring a quantity q_0 of assets within a certain time window. However, if many other agents are considered to be either selling or buying, thus pursuing their own optimal execution schedules in the same market, the model becomes more complicated since it now requires modelling other agents' behaviour the same market. Using a large dataset of real optimal executions, Bucci et al. [2020] shows that the cost of an execution strongly depends on the presence of other agents liquidating the same asset. The increased complexity when treating the problem in this way allows for more consistency with reality and, at the same time, opens the path for further studies on how the agents interact with each other in such a context.

The optimal execution problem with n-agents operating in the same market has thus been studied under a game theoretic lens in both its closed-loop (in Carmona and Yang [2011], Micheli et al. [2023]) and open-loop (in Brunnermeier and Pedersen [2005], Carlin et al. [2007], Schöneborn and Schied [2009], Schied and Zhang [2017, 2019]) formulations. In more recent works, rather than just optimal execution, liquidity competition is also analysed (as in Drapeau et al. [2019], Neuman and Voß [2023]) along with market making problems (as in Cont et al. [2021], Cont and Xiong [2022]).

In recent times, with the advancements of machine learning techniques, the original optimal execution problem has been extensively studied using RL techniques. Among the many[1], some examples in the literature on RL techniques applied to the optimal execution problem are found in Ning et al. [2021], Schnaubelt [2022], Macrì and Lillo [2024].

Applications of multi-agent RL to financial problems are scarce if compared to those that study the problem in a single agent scenario. Optimal execution in a multi-agent environment is tackled in Karpe et al. [2020], where the authors analyse the optimal execution problem form the standpoint of a many agents environment and an RL agent trading on Limit Order Book. The authors test and develop a multi-agent Deep Q-learning algorithm able to trade on both simulated and real order books data. Their experiments converge to agents who adopt only the so-called Time Weighted Average Price (TWAP) strategies in both cases. On the other hand, still on optimal execution problem's applications of multi-agent RL, the authors in Bao

[1]For a more comprehensive overview of the state of the art on RL methods in finance, please refer to Sun et al. [2023] and Hambly et al. [2023]

and Liu [2019] analyse the interactions of many agents on their respective optimal strategies under the standpoint of cooperative competitive behaviours, adjusting the reward functions in the Deep Deterministic Policy Gradient algorithm used, in order to allow for either of the two phenomena, but still using the basic model introduced in Almgren and Chriss [2000], and modelling the interactions of the agents via full reciprocal disclosure of their rewards and not considering the sum of the strategies to be a relevant feature for the permanent impact in the stock price dynamics.

The existence of collusion and the emergence of collusive behaviours are probably the most interesting phenomena based on agents interaction in a market, as these are circumstances that might naturally arise in markets even if no instruction on how to and whether to collude or not, are given to the modelled agents.

One of the first contributions to collusion theory of autonomous algorithms has been put forth in Waltman and Kaymak [2008]. In this contribution the authors show empirically by means of Q-learning, how two firms in a production game with linear demand function, can learn collusive behaviour even if no direct communication and explicit punishment strategies are present. Hence, in this contribution the authors show how competing producers in a Cournot oligopoly learn to increase prices above Nash equilibrium by reducing production, so that firms learn to collude even if they do not learn to obtain the highest possible joint profit. Thus, the firms reach a supra-competitive equilibrium, comparable to the collusive one even if no punishment mechanism is explicitly present or direct communication between the players is present. This remarkable result has been further analysed and extended in Calvano et al. [2020]. Here, the authors show how using Q-learning, two agents in a duopoly are able to retrieve supra-competitive equilibria without information exchange. The equilibrium is sustained by collusive strategies enforced by eventual punishment phases in the case where an agent deviates from said collusive equilibrium.

Notwithstanding the empirical emergence of collusive behaviour, some concerns have emerged on whether or not the supra-competitive outcomes of such games are indeed collusions or are due to some algorithm-specific features and thus, if these possibly tacit collusive behaviours have to be a major concern for the regulators. This topic has been investigated in Abada and Lambin [2023], where the authors consider battery and electricity markets to empirically prove how such agents would indeed punish *any* deviation from a collusive strategy by induc-

ing shocks to the environment where the agents are playing after they have converged in the training phase. They conjecture that this seeming collusion is due to imperfect exploration of states and parameter initialisation. More recently, Abada et al. [2024] expands this point of view by showing how in a Q-learning environment seemingly supra-competitive equilibria are due to the poor exploration policy of unsophisticated algorithms. They show how for simple Q-learning algorithms there is an auto-induced cooperative trap if states are not thoroughly explored and learning rate decreases 'too fast', consequently implying that cooperative strategies have not been explored enough during training. This is opposite to what the authors find with more sophisticated algorithms where collusive behaviours do not emerge. The conclusion is that the supra-competitive outcomes should be of concern for market authorities if and only if the algorithms employed are not sophisticated. Also Ref. Lambin [2024] investigates when supra-competitive equilibria result necessarily from collusive strategies based on reward and punishment mechanisms to sustain cooperation. Unlike previous studies, their analysis critically suggests that both high prices and these strategic patterns actually emerge from a combination of simultaneous exploration and learning inertia intrinsic to reinforcement learning techniques, without any direct causal connection. They show that such forms of apparent collusion can arise even in memoryless settings and with myopic[2] agents. Based on these findings, they caution against making straightforward analogies with human collusion and suggest addressing elevated algorithmic prices through tailored regulatory measures.

There is no unanimous consensus on whether algorithmic collusion does –in fact– naturally emerge from the interaction of autonomous agents, and on the drivers that lead to supra-competitive equilibria. Especially, because of the definition of collusion[3] that, according to Abada and Lambin [2023] is more suitable to a human to human cooperative behaviour rather than to an algorithmic driven phenomenon. Thus, a part of the literature points out at these phenomena as tacitly collusive equilibria while another stream interprets these results as not necessarily collusive even if the strategies employed by the agents might suggest so. This implies that applying existing regulatory solutions to markets where prices are set through reinforcement learning algorithms may miss the point, and thus interventions by the regulator on such markets might eventually be inadequate. In this fragmented landscape, literature on

[2]Agents with a low discount factor on future rewards, i.e. that value more immediate rewards over future possible ones.

[3]"Collusion is when firms use strategies that embody a reward-punishment scheme which rewards a firm for abiding by the supra-competitive outcome and punishes it for departing from it" as defined in Harrington [2018].

collusion in financial markets, to the best of our knowledge, has just started to develop. In fact, for what concerns financial markets, the problem becomes more involved since the actors in the market do not directly set the price in a 'one sided' way as is the case for production economies that are studied in economic literature. Among the many contributions that study the emergence of collusive behaviours in financial markets, in Xiong and Cont [2022] it is shown how tacit collusion arises between market makers, modelled using deep RL, in a competitive market. In Cartea et al. [2022] the authors show how market making algorithms tacitly collude to extract rents, and this behaviour strictly depends on tick size and coarseness of price grid. Finally, in Cont and Xiong [2022] the authors use a multi-agent deep RL algorithm to model market makers competing in the same market, the authors show how competing market makers learn how to adjust their quotes, giving rise to tacit collusion by setting spread levels strictly above the competitive equilibrium level.

To the best of our knowledge no prior work did consider supra-competitive, possibly tacit collusive, equilibria in optimal execution problems. As discussed in the Conclusions section, the existence of such equilibria might indeed imply a loss of welfare for other market participants. Therefore, studying the behaviour of learning agents opens for rather new perspectives on how collusion eventually arises through learning. Finally, studying supra-competitive equilibria and potential collusion for this class of problems is thus instrumental to understanding, and eventually, easing the regulatory tasks of market authorities when it comes to assess possible informational distortions of assets prices which, in turn, might affect the correct functioning of financial markets, resulting in welfare loss for market participants. This paper aims at tackling these points, while enlarging the analysis on these equilibria to the optimal execution problem.

The paper is organised as follows: Section 2 introduces the theoretical setting of the market impact game, Section 3 introduces the DDQL algorithm for the multi-agent optimal execution problem, Section 4 discusses the results for different parameter settings, in Section 5 we critically discuss whether the observed results are tacit collusion or supra-competitive outcomes and finally Section 6 provides conclusions and outlines further research directions.

5.2 Market impact game setting

The Almgren-Chriss model. The setup of our framework is based on the seminal Almgren-Chriss model first introduced in Almgren and Chriss [2000] for optimal execution. In this setting,

a single agent wants to liquidate an initial inventory of q_0 shares within a time window $[0, T]$, which, in the discrete-time setting, is divided into N equal time increments of length $\tau = T/N$. The main assumption of the model is that the mid-price (or the efficient price) evolves according to a random walk with a drift depending on the traded quantity. Moreover, the price obtained by the agent differs from the mid-price by a quantity that depends on the volume traded in the interval. More formally, let S_t and $\tilde{S}_t$ be the mid-price and the price received by the agent at time t, and let v_t be the number of shares traded by the agent in the same interval; then the dynamics is given by

$$\begin{aligned} S_t &= S_{t-1} - p\left(v_t/\tau\right)\tau + \sigma\tau^{\frac{1}{2}}\xi_t \\ \tilde{S}_t &= S_{t-1} - h\left(v_t/\tau\right) \quad . \end{aligned} \tag{5.1}$$

S_t evolves because of a diffusion part $\xi_t \sim \mathcal{N}(0, 1)$ multiplied by the price volatility σ and a drift part, termed *permanent impact*, $p\left(v_t/\tau\right)$, assumed to be linear and constant: $p\left(v_t/\tau\right) = \kappa v_t/\tau$. The price $\tilde{S}_t$ received by the agent is equal to the mid-price S_t but impacted by a *temporary impact* term also assumed to be linear and constant in time: $h(v_t/\tau) = \alpha v_t/\tau$.

These two impacts are the classic components of market impact. In fact, they represent the adverse movement of the asset price generated by trades. Permanent impact reflects permanent changes in the asset's fundamental price process, while the temporary impact should reflect some temporary market imbalances. The former is cumulatively assumed to impact the asset price at least until the end of the execution window, while the latter vanishes as soon as the trade has been performed.

The aim of the agent is to unwind their initial portfolio maximising the cash generated by their trading over the N time steps. This objective can be rewritten in terms of Implementation Shortfall (IS)[4] that is defined in the single agent case as:

$$IS(\vec{v}) = S_0 q_0 - \sum_{t=1}^{N} \tilde{S}_t v_t \quad , \tag{5.2}$$

where $\vec{v} = (v_1, ..., v_N)$ is the vector that contains the positive quantities sold at each time step, i.e. $v_t \geq 0,\ \forall\ t \in [1, N]$ and S_0 is the mid-price at the beginning of the trading window. The

[4]Notice that the IS could, in principle, take negative values as well. But in the absence of exogenous price drift and linear impact functions, as in our case, its expected value is positive.

optimisation problem faced by the agent can be written as

$$\min_{\vec{v}} \ \mathbb{E}[IS(\vec{v})]+\lambda\mathbb{V}[IS(\vec{v})] \qquad s.t. \ \sum_{t=1}^{N} v_t = q_0 \quad , \tag{5.3}$$

where λ is the risk aversion parameter of the agent, $\mathbb{E}[\cdot]$ and $\mathbb{V}[\cdot]$ refer to the expected value and the variance of the IS functional respectively. Under the linearity assumption of the two impacts, the problem can be easily solved analytically. In the following, we are going to consider risk neutral agents ($\lambda = 0$) and in this case the optimal trading schedule is the TWAP, i.e. $v_t = q_0/N$, where the trading velocity is constant.

Almgren-Chriss market impact game. Now we consider two agents selling an initial inventory q_0 shares within the same time window time window $[0, T]$. The traded quantities by the two agents in the time step t are indicated with $v_t^{(1)}$ and $v_t^{(2)}$ and $V_t = v_t^{(1)} + v_t^{(2)}$ is the total quantity traded in the same interval. The equations for the dynamics in discrete time are

$$\begin{aligned} S_t &= S_{t-1} - p\left(V_t/\tau\right)\tau + \sigma\tau^{\frac{1}{2}}\xi_t \\ \tilde{S}_t^{\,(k)} &= S_{t-1} - h\left(v_t^{(k)}/\tau\right) \qquad k = 1, 2 \quad ; \end{aligned} \tag{5.4}$$

i.e. the mid-price is affected by the total traded volume V_t, while the price received by each agents depends on the quantity they trade. Notice that, according to Schied and Zhang [2017] the temporary impact depends only on the volume traded by the agent singularly and not on the volume traded by the other agent. This corresponds to the case where the asset is relatively liquid. Although it would be interesting to extend the model to the case of a temporary impact that depends on both agents' traded volumes, in the present work we will consider the case in Eq. (5.4). Since more than one agent are optimising their trading volumes and the cash received depends on the trading activity of the other agent, the natural setting to solve this problem is the one of game theory. Since each agent is not directly aware of the selling activity of the other agent and the two agents interact through their impact on the price process, the resulting problem is an open-loop game. The existence and uniqueness of a Nash equilibrium in such a symmetric open-loop game has been studied in Schied and Zhang [2017] in the case of n different players and linear impact functions. First of all, they defined the Nash equilibrium, in continuous time and considering the vector of levels of inventory held at each time-step t, i.e. $\vec{q}^{\,(k)}$, as:

Definition 1 (Nash equilibrium Schied and Zhang [2017])**.** *In an n players game, where* $n \in \mathbb{N}$*, with* $q_0^{(1)}, \ldots, q_0^{(n)} \in \mathbb{R}$ *initial inventory holdings and* $\lambda_1, \ldots, \lambda_n$ *non-negative coefficients of risk aversion. A Nash equilibrium for mean-variance optimisation as in Eq.* (5.3) *is a collection of* $\mathbf{q}^* = \{\vec{q}^{(1)^*}, \ldots, \vec{q}^{(n)^*}\}$ *inventory holdings such that for each* $k \in n$*,* $\vec{q}^{\,(k)} \in \mathcal{A}_{det}$ *the mean variance functional is minimised:*

$$\mathbb{E}[IS(\vec{q}^{(k)}|\mathbf{q}^{(-\mathbf{k})^*})] + \lambda_k \mathbb{V}[IS(\vec{q}^{(k)}|\mathbf{q}^{(-\mathbf{k})^*})]$$

for each agent k,[5] *where* $\mathbf{q}^{(-\mathbf{k})^*}$ *are the strategies of the other players minus the strategy of the k-agent that is being considered.*

The set of deterministic strategies, $\mathcal{A}_{\text{det}}$, in the definition is the set of the strategies that is not adapted to the mid-price level. Thus, it does not depend on the realisation of the noise ξ_t in the dynamics of the price process.

Ref. Schied and Zhang [2017] proves that there exists a unique Nash equilibrium for the mean-variance optimisation problem. The unique Nash equilibrium strategy $q_t^{(k)^*}$, the Nash remaining inventory of agent k at time t, for n players is given as a solution to the second-order system of differential equations:

$$\lambda^{(k)}\sigma^2 q_t^{(k)} - 2\alpha\tau\, \ddot{q}_t^{(k)} = \kappa \sum_{j\neq k} \dot{q}_t^{(k)} + \alpha\tau \sum_{j\neq k} \ddot{q}_t^{(k)} \tag{5.5}$$

with two-point boundary conditions

$$q_0^{(k)} = q_0 \quad \text{and} \quad q_T^{(k)} = 0\ , \quad \forall\ k = 1, \ldots, n \tag{5.6}$$

Focusing on the special case of two players, in Schied and Zhang [2017] it is proved that the selling schedule at the unique Nash equilibrium is:

$$q_t^{(1)^*} = \frac{1}{2}(\Sigma(t) + \Delta(t)) \qquad \text{and} \qquad q_t^{(2)^*} = \frac{1}{2}(\Sigma(t) - \Delta(t))\,, \tag{5.7}$$

[5] Where $\mathcal{A}_{\text{det}}$ is the set of all admissible deterministic trading strategies, for a more formal definition see Schied and Zhang [2017].

with:

$$\Sigma(t) = Qe^{-\frac{\kappa t}{6\alpha}} \frac{\sinh\left(\frac{(N-t)\sqrt{\kappa^2+12\alpha\lambda\sigma^2}}{6\alpha}\right)}{\sinh\left(\frac{N\sqrt{\kappa^2+12\alpha\lambda\sigma^2}}{6\alpha}\right)} \qquad \text{and} \qquad \Delta(t) = \tilde{Q}e^{\frac{\kappa t}{2\alpha}} \frac{\sinh\left(\frac{(N-t)\sqrt{\kappa^2+4\alpha\lambda\sigma^2}}{2\alpha}\right)}{\sinh\left(\frac{N\sqrt{\kappa^2+4\alpha\lambda\sigma^2}}{2\alpha}\right)} \tag{5.8}$$

where $Q = q_0^{(1)} + q_0^{(2)}$ and $\tilde{Q} = q_0^{(1)} - q_0^{(2)}$. It can be noticed that the Nash inventory holdings, and thus the trading rates, now depend also on the permanent impact κ, contrary to what happens in the single agent case studied in Almgren and Chriss [2000]. The price level does not enter directly into the optimal inventory formula, but the permanent impact and the volatility of the asset on the other hand do, proportionally to agents' risk aversion.

Remark 3. *In Schied and Zhang [2017] the authors derive the unique Nash equilibrium considering the continuous time version of Eq.*(5.4)*. We will use the Nash inventory holdings and the selling strategy associated to Eq.*(5.7)-(5.8) *discretised over time. We choose to consider the discretised version of the Nash equilibrium, and to derive results in discrete time as they are functional to the numerical and deep RL experiments, as well as to highlight the differences that might arise when autonomous agents are trained together in a similar but subtly different environment. The Nash equilibrium in the discrete time setting was derived in Section 2 of Cordoni and Lillo [2024b], considering a constant decay kernel in a transient impact market.*[6]

Beyond the Nash equilibrium. In this setting, as shown a *unique* Nash equilibrium exists but, leveraging on reviewed literature, other non-Nash equilibria might as well. Especially, supra-competitive and collusive equilibria have been shown to exists in oligopolistic settings such as the one considered in a market impact game. To this end, we aim at studying the behaviour of two agents who have to solve an optimal execution problem in a multi-agent setting. We study the problem from a *learning* perspective, meaning that we will use deep RL to train the agents to solve the liquidation problem in such setup; we then study the strategies found by the agents, comparing the cost of the learnt strategies with the ones prescribed by both the Nash equilibrium strategy and the Pareto optimal strategy, that will be shown to be the collusive strategy. In order to do so, we focus on the case where two *risk neutral* agents want to liquidate the same initial portfolio made by an amount q_0 of the same asset. In the numerical

[6] In Schied et al. [2017] the authors show that the high-frequency limits of the discrete-time equilibrium costs converge to the expected costs in the continuous-time Nash equilibrium.

setup we let the agents play multiple episode instances of the optimal execution problem. This means that, defining a trading episode to be the complete unwinding of the initial inventory q_0 over time $[0, T]$ in N time steps by both agents, the overall game is made by repeated trades at each $t = 1, \dots, N$ time-steps and by B iterations of trading, each containing a full inventory liquidation episode. Therefore, for each iteration i we will consider two vectors $\vec{v}^{(1)}$ and $\vec{v}^{(2)}$ containing the trading schedule for the two agents, .

The numerical setup is justified by the episodical nature of the game under consideration, as by definition -and differently from other setups- the optimal execution problem happens over a finite time-window. Each vector is associated with its IS cost, according to Eq. 5.2. This cost, along with the strategies' vectors are going to be the main objective of our study. We now define the set of admissible strategies and the average IS cost.

Definition 2. *The set $\mathcal{A}$ of admissible selling strategies is composed by the pair of vectors $\vec{v}^{(1,2)} = (\vec{v}^{(1)}, \vec{v}^{(2)})$, such that for $k = 1, 2$:*

- $\sum_{t=1}^{N} v_t^{(k)} = q_0$,
- $\vec{v}_t^{(k)}$ *is $(\mathcal{F}_t)_{t\geq 0}$ - adapted and progressively measurable,*
- $\sum_{t=1}^{N} (v_t^{(k)})^2 < \infty$.[7]

For any $\vec{v}^{(1,2)} \in \mathcal{A}$ the average Implementation Shortfall is defined as:

$$\bar{IS}\left(\vec{v}^{(1,2)}\right) = \frac{1}{N} \sum_{t=1}^{N} IS\left(\vec{v}_t^{(1,2)}\right) . \tag{5.9}$$

where $IS\left(\vec{v}_t^{(1,2)}\right) = IS\left(\vec{v}_t^{(1)}\right) + IS\left(\vec{v}_t^{(2)}\right)$.

Leveraging on Cont and Xiong [2022] and Cont et al. [2021], we define *collusive* strategies and we show that they are necessarily Pareto-optimal. Notice that, given the iterated nature of the game due to the time and inventory constraints that -by design- constrain the agents, we will consider the following equilibrium properties to hold over several iterations of the game. Although the same properties would hold in a one-shot game, we aim at studying what equilibrium would naturally arise when the agents are simultaneously *learning*, without being told to either cooperate or compete. The theoretical results stated below are functional to the analysis

[7] This condition in the discrete-time setting is always trivially true, but we choose to keep it as it is inherited from the continuous-time formulation of the admissible strategy space.

of the learned equilibria, therefore in the numerical experiments part we depart from a one-shot game setup in favour of an iterated game setup.

Definition 3 (Collusion). *A pair of vectors of selling schedules* $\vec{v}_c^{(1,2)} = (\vec{v}_c^{(1)}, \vec{v}_c^{(2)}) \in \mathcal{A}$ *is defined to be a collusion if* $\forall\ \vec{v}^{(1,2)} \in \mathcal{A}$*:*

$$\bar{IS}(\vec{v}_c^{(1,2)}) \leq \bar{IS}(\vec{v}^{(1,2)})$$

Definition 4 (Pareto Optimum). *A pair of vectors of selling schedules* $\vec{v}_p^{(1,2)} = (\vec{v}_p^{(1)}, \vec{v}_p^{(2)}) \in \mathcal{A}$ *is a Pareto optimal strategy if and only if there does not exist* $\vec{v}^{(1,2)} \in \mathcal{A}$ *such that:*

$$\begin{aligned} &\forall\ i = 1, \ldots, B\,, \quad \bar{IS}(\vec{v}_p^{(1,2)}) \geq \bar{IS}(\vec{v}^{(1,2)}) \\ &\exists\ j = 1, \ldots, B\,, \quad \bar{IS}(\vec{v}_p^{(1,2)}) > \bar{IS}(\vec{v}^{(1,2)}) \end{aligned} \tag{5.10}$$

Proposition 1 (Collusive Pareto optima). *For an optimal execution strategy problem where two agents minimise costs as in Eq.* (5.3) *in a market as in Eq.* (5.4)*, a collusive selling strategy* $\vec{v}_c^{(1,2)}$ *in the sense of Definition 3 is necessarily a Pareto optimum as defined in Definition 4. We call this a Collusive Pareto optimal strategy* $\vec{v}_{cp}^{(1,2)}$*.*

Proof. By contradiction, suppose that $\vec{v}_c^{(1,2)}$ is not Pareto optimal, then there must exist an iteration i and a strategy $\vec{u}^{(1,2)} = (\vec{u}^{(1)}, \vec{u}^{(2)})$ such that:

$$\bar{IS}(\vec{u}^{(1,2)}) \leq \bar{IS}(\vec{v}_c^{(1,2)}) \tag{5.11}$$

But this contradicts the hypothesis that $\vec{v}_c^{(1,2)}$ is collusive and hence $\vec{v}_c^{(1,2)}$ must be Pareto-optimal. □

Having shown that a collusive strategy is in fact the Pareto-optimal strategy, we aim at finding the set of all Pareto-efficient strategies, i.e. those selling strategies that result in IS levels where it is impossible to improve trading costs for one agent without deteriorating the one of the other agent. In other words, we are looking for those strategies where an agent is better off only if the other agent is be worse off. We define the set of Pareto solutions and, within this set and we find the Pareto-optimum as the minimum within the set of solutions.

Definition 5 (Pareto-efficient set of solutions). *For the two-player game, considering risk neutral players that aim to solve the optimal execution problem in a market model defined as in*

Eq. (5.4)*, the Pareto-efficient set of strategies is the set of strategies* $\vec{v}^{(1,2)}$ *such that the IS of one agent cannot be improved without increasing the IS value of the other agent.*

We now provide the conditions that allow to find the set of Pareto-efficient solutions as a minimisation of the Implementation Shortfall functional of both the agents. The minimisation has to be intended in the sense of Pareto dominance component-wise.

Theorem 1 (Pareto-efficient set of solutions)**.** *The Pareto-efficient set of strategies are the solutions to the multi-objective optimisation problem:*

$$\begin{cases} \min_{\vec{v}^{(1,2)}} \vec{f}\left(\vec{v}^{(1,2)}\right) \\ s.t.\ \{\vec{v}^{(1,2)} \in \mathcal{A},\ \vec{g}(\vec{v}^{(1,2)}) = \vec{0}\} \end{cases} \tag{P1}$$

where:

$$\vec{f}(\vec{v}^{(1,2)}) = \left[\mathbb{E}\left(IS(\vec{v}^{(1)}|\vec{v}^{(2)})\right), \mathbb{E}\left(IS(\vec{v}^{(2)}|\vec{v}^{(1)})\right)\right]$$
$$\vec{g}(\vec{v}^{(1,2)}) = \left[\left(\sum_{t=1}^{N} v_t^{(1)} - q_0\right)\ , \left(\sum_{t=1}^{N} v_t^{(2)} - q_0\right)\right]$$

Proof. Problem (P1) is a convex multi-objective optimisation problem with $n = N$ design variables ($\vec{v}^{(1,2)}$), $k = 2$ objective functions, and $m = 2$ constraint functions. Leveraging on Gobbi et al. [2015], in order to find the Pareto-efficient set of solutions, Problem (P1) can be restated in terms of Fritz-Johns conditions. We define the $\mathbf{L} \in \mathbb{R}^{(n+m)\times(k+m)}$ matrix as:

$$\mathbf{L} = \begin{bmatrix} \nabla_{\vec{v}^{(1,2)}} \vec{f}(\vec{v}^{(1,2)}) & \nabla_{\vec{v}^{(1,2)}} \vec{g}(\vec{v}^{(1,2)}) \\ \vec{0} & \vec{g}(\vec{v}^{(1,2)}) \end{bmatrix} \tag{5.12}$$

then, the strategy $\vec{v}_p^{(1,2)}$ is a solution to the problem:

$$\mathbf{L} \cdot \vec{\delta} = \vec{0} \tag{P2}$$

where $\vec{\delta} = (\vec{\omega}, \vec{\lambda}) \in \mathbb{R}^{k+m}$. Moreover, $\vec{v}_p^{(1,2)}$ is a Pareto-efficient solution if $\vec{\delta}$ exists, it is a non-trivial solution if $\vec{\delta}$ exists and $\vec{\delta} \neq \vec{0}$.

Leveraging on Gobbi et al. [2015], a solution $\vec{v}^{(1,2),*}$ to the Problem (P2) is a non-trivial Pareto-efficient solution if:

$$\det(\mathbf{L}(\vec{v}^{(1,2),*})^T \mathbf{L}(\vec{v}^{(1,2),*})) = \vec{0} \tag{5.13}$$

where $\mathbf{L}(\vec{v}^{(1,2),*})$ denotes the matrix $\mathbf{L}$ where the argument of $f(\cdot)$ and $g(\cdot)$ functions is a strategy $\vec{v}^{(1,2),*}$. Thus we say that a necessary condition for generic strategy $(\vec{v}^{(1,2),*})$ to be Pareto-efficient is to satisfy Eq. (5.13). In general, Eq. (5.13) gives the analytical formula that describes the Pareto-optimal set of solutions obtainable from Problem (P2). □

The analytical derivation of the Pareto-efficient set of strategies, i.e. of the Pareto-front, for this kind of problem is quite involved and cumbersome, although obtainable via standard numerical multi-objective techniques. Later in the paper, we will solve this problem numerically to find the set for the problem at hand. It is instead possible to derive the absolute minimum of the Pareto-efficient set of solutions since the problem is a sum of two convex functions and is symmetric. We call these strategies, one per agent, Pareto-optimal strategies in the spirit of Definition 4.

Theorem 2 (Pareto optimal strategy)**.** *For a two-player game with risk-neutral players, the Pareto optimal strategy is the solution of the problem:*

$$\begin{cases} & \arg\min_{\vec{v}^{(1)},\vec{v}^{(2)}} \; F(\vec{v}^{(1,2)}) \\ s.t. & \left(\sum_{t=1}^{N} v_t^{(1)} - q_0\right) = 0 \\ & \left(\sum_{t=1}^{N} v_t^{(2)} - q_0\right) = 0 \end{cases} \tag{5.14}$$

where:

$$F(\vec{v}^{(1,2)}) = \mathbb{E}\left(IS(\vec{v}^{(1)}|\vec{v}^{(2)})\right) + \mathbb{E}\left(IS(\vec{v}^{(2)}|\vec{v}^{(1)})\right) \tag{5.15}$$

is $\forall \; t = 1, \ldots, N$*:*

$$v_{t,p}^{(1)} = \frac{q_0}{N}\,, \qquad\qquad v_{t,p}^{(2)} = \frac{q_0}{N} \tag{5.16}$$

i.e. the TWAP strategy.

Proof. First define $\vec{v}^{(1)} = \arg\min_{\vec{v}^{(1)}} IS(\vec{v}^{(1)}|\vec{v}^{(2)})$, meaning that the strategy for agent 1 is a function of the strategy of agent 2. And thus $\vec{v}^{(1)}$ the strategy that minimises the IS of agent 1 given the strategy of agent 2. The same consideration holds for agent 2, where $\vec{v}^{(2)} = \arg\min_{\vec{v}^{(2)}} IS(\vec{v}^{(2)}|\vec{v}^{(1)})$.

Leveraging on Proposition 1, the Pareto optimal strategies $\vec{v}_p^{(1)}, \vec{v}_p^{(2)}$ of the agents considered

solve:

$$\underset{\vec{v}^{(1)},\vec{v}^{(2)}}{\arg\min}\ F(\vec{v}^{(1,2)}) \tag{P1}$$

where:

$$F(\vec{v}^{(1,2)}) = \mathbb{E}\left(IS(\vec{v}^{(1)}|\vec{v}^{(2)})\right) + \mathbb{E}\left(IS(\vec{v}^{(2)}|\vec{v}^{(1)})\right) \tag{5.17}$$

we then set up a constrained optimisation problem, where the constraint binds the agents to just sell their initial inventory q_0 over the time window considered , where now λ_1, λ_2 are multipliers:

$$\nabla_{\vec{v}^{(1,2)}}\left[F\left(\vec{v}^{(1,2)}\right) + \lambda_1\left(\sum_{t=1}^{N} v_t^{(1)} - q_0\right) + \lambda_2\left(\sum_{t=1}^{N} v_t^{(2)} - q_0\right)\right] = 0 \tag{5.18}$$

this can be decoupled into two distinct problems:

$$\begin{aligned}
\nabla_{\vec{v}^{(1)},\lambda_1}\mathcal{L}_1 &= \nabla_{\vec{v}^{(1)},\lambda_1}\left[\mathbb{E}\left(IS(\vec{v}^{(1)}|\vec{v}^{(2)})\right) + \lambda_1\left(\sum_{t=1}^{N} v_t^{(1)} - q_0\right)\right] = 0 \\
\nabla_{\vec{v}^{(2)},\lambda_2}\mathcal{L}_2 &= \nabla_{\vec{v}^{(2)},\lambda_2}\left[\mathbb{E}\left(IS(\vec{v}^{(2)}|\vec{v}^{(1)})\right) + \lambda_2\left(\sum_{t=1}^{N} v_t^{(2)} - q_0\right)\right] = 0
\end{aligned} \tag{5.19}$$

Considering now just the first problem in the previous equation, we notice that:

$$\mathbb{E}\left(IS(\vec{v}^{(1)}|\vec{v}^{(2)})\right) = -\sum_{t=1}^{N}\kappa v_t^{(1)}\sum_{j=1}^{t}(v_j^{(1)} + v_j^{(2)}) - \sum_{t=1}^{N}\alpha {v_t^{(1)}}^2$$

and thus, for every $t = 1, \ldots, N$:

$$\begin{cases}
\frac{\partial\mathcal{L}_1}{\partial v_t^{(1)}} = -\kappa\left((2q_0 - q_t^{(1)} - q_t^{(2)}) + v_t^{(1)}\right) - 2\alpha v_t^{(1)} + \lambda_1 = 0 \\
\frac{\partial\mathcal{L}_1}{\partial\lambda_1} = \left(\sum_{t=1}^{N} v_t^{(1)} - q_0\right) = 0
\end{cases} \tag{5.20}$$

where $\sum_{j=1}^{t} v_j^{(1)} = (q_0 - q_t)$ and q_t is the level of inventory held at the end of the time-step $t = 1, \ldots, N$, we notice that:

$$v_t^{(1)} = -\frac{\kappa(2q_0 - q_t^{(1)} - q_t^{(2)}) + \lambda_1}{\kappa + 2\alpha}$$

thus:

$$
\begin{aligned}
0 &= \sum_{t=1}^{N} -\frac{1}{\kappa + 2\alpha}\kappa \left((2q_0 - q_t^{(1)} - q_t^{(2)}) + \lambda \right) - q_0 \\
\lambda &= \frac{q_0(\kappa + 2\alpha)}{N} + \kappa(2q_0 - q_t^{(1)} - q_t^{(2)}) \\
v_t^{(1)} &= -\frac{\kappa(2q_0 - q_t^{(1)} - q_t^{(2)})}{\kappa + 2\alpha} + \frac{q_0(\kappa + 2\alpha)}{N(\kappa + 2\alpha)} + \frac{\kappa(2q_0 - q_t^{(1)} - q_t^{(2)})}{\kappa + 2\alpha}
\end{aligned}
$$

Finally, the first and the last terms cancel out and we obtain that the Pareto-optimal strategy $\vec{v}_p^{(1)}$ is:

$$
v_{t,p}^{(1)} = \frac{q_0}{N} \quad , \ \forall \ t = 1, \ldots, N \tag{5.21}
$$

for $v_{t,p}^{(2)}$ same considerations hold since the problem is symmetric.

□

To study whether *collusive Pareto-optima*, or more generally supra-competitive non-Nash equilibria, arise in this setting, we model the two agents by equally and simultaneously training them with Double Deep Q Learning (DDQL), a deep RL algorithm. We then analyse the results under the light of the collusion strategies derived above. The aim is to understand how and which equilibria will eventually be learned, and whether in attaining an equilibrium they tacitly collude in order to further drive the costs of their trading down, sustained by a punishment mechanism.

We choose to implement this algorithm and use it for our analysis for two main reasons. On the one hand, using Double Deep Q-Learning allows us to be more consistent with the extant literature that mainly uses Q-learning, outlining if observing tacit-collusive behaviours is still possible when using deep neural networks and memory of past states. In fact, from the literature it is well known that such an off-policy algorithm works well in the single agent case (see, for instance, Macrì and Lillo [2024], Ning et al. [2021]), making it a safe and sensible ground choice when it comes to extend the framework to more than one agent. On the other hand, and more methodologically, we have chosen Double Deep Q-learning because this algorithm directly maximizes the expected discounted rewards without explicitly parametrizing the policy. Thus, this algorithm choice allows us to analyse the problem from a purely reward driven point of view, as the Q-values of the agents are the quantities that are optimised through the use of neural networks without the need to visit every state. We think that an interesting extension of our work could be to compare different RL algorithms to identify pros and cons of each of

them.

5.3 Double Deep Q Learning for multi-agent impact trading

We model the algorithmic agents by using RL based on Double Deep Q-Learning (DDQL). The setup is similar to the single agent algorithm as in Ning et al. [2021] and in Macrì and Lillo [2024], but now we consider two agents interacting in the same environment. The agents are modelled to sequentially interact at each time step to trade either as first or as second. In principle, the game does not need to be a sequential move game but as in real markets, agents cannot trade simultaneously and for this reason we have chosen this numerical setup for the experiments. Moreover, the choice of the algorithm is motivated by the fact that this algorithm has been proven to work with both real data, as in Ning et al. [2021], and especially with simulated data where the permanent impact and the risk aversion parameter are present as in Macrì and Lillo [2024].

Each agent employs two neural networks, namely the main Q-net (Q_{main}) for action selection and the target Q-net (Q_{tgt}) for state evaluation. The four nets share the same architecture, their weights evolve independently during the training phase and they are updated at exactly the same rate. The only difference between the two agents is the timing at which they act, the time at which their nets are updated and the probability of being exploring or exploiting during training.

In fact, at each time step a coin toss decides which agent acts and undergoes training first within the same time step. To be consistent with the model in Schied and Zhang [2017], the toss does not influence the intra-step dynamics of the game in terms of impact paid by either of the agents. A similar model where the coin toss decides which agent trades first, letting the second pay the impact generated by the first-mover would not change the results obtained both theoretically and computationally, we discuss this in Appendix B.2.

We divide the overall numerical experiment into a training and a testing phase. In the training phase we train both the agents to solve the optimal execution problem. In the testing phase, we employ what learnt in terms of Q_{main} weights, letting the trained agents play an iterated trading game. All the results shown in Section 5.4 are obtained in the testing phase.

5.3.1 Setting of the numerical experiments

We consider two risk neutral agents whose goal is to unwind an initial position of $q_0 = 100$ shares with initial value $S_0 = 10\$$ within a time window $[0, T]$. The window is divided into $N = 10$ time-steps of length $T/N = \tau$. The mid-price S_t evolves as in Eq. (5.4), and the agents sell their whole inventory during the considered time-window. This is called an *iteration* and in order to train the DDQL algorithm we consider a number of C iterations. This is called a *run*. Thus, a run of the game is defined to be a repeated game over both N time-steps and in C trading iterations.

Over the C iterations, each agent learns how to trade via an exploration-exploitation scheme, thus using ϵ-greedy policies. This is obtained by changing the weights of their Q_{main} net in order to individually choose the best policies in terms of rewards, related to the obtained IS. The scheme of the algorithm is thus symmetric, and for each agent it is divided into two 'phases': an *exploration* and an *exploitation* phase managed by the parameter $\epsilon \in (0, 1]$, that is common to both the agents[8], that will decrease geometrically during training and globally initialised as $\epsilon = 1$. Depending on the phase, the way in which the quantities to sell v_t are chosen changes, and it does so for each agent symmetrically. When the agents are exploring, they will randomly select quantities to sell v_t in order to explore different states and rewards in the environment, alternatively, when the agents are exploiting, they will use their Q_{main} net to select v_t.

Action selection and reward function

For each of the N time-steps and C trading iterations, the agents' knowledge of the state of market environment is the tuple $g_t^i = (t, q_t^i, S_{t-1})$ for $i = 1, 2$. Thus, each agent knows the current time-step, her individual remaining inventory, and the permanently impacted mid-price at the previous time step. In our setting $t \in \{1, \dots, N\}$ is the discrete time index, $q_t \in [0, q_0]$ is the agent's remaining inventory at time t, $S_{t-1} \in \mathbb{R}_+$ is the (permanently impacted) mid-price at the previous time step. All three components are normalised to lie in $[-1, 1]$ before being used as input to the neural network.

Clearly, only the first and the last are common knowledge of the two agents, while no information on inventory q_t or past selling actions v_t is shared between the agents.

Then, depending on the current value of ϵ, a draw ζ, different for each agent, from a uniform distribution determines whether the agent performs exploration or exploitation. This

[8] More practically, the agents will explore with probability ϵ and exploit with probability $1 - \epsilon$.

means that with probability ϵ the agent chooses to explore and thus she chooses at time-step t the quantity to sell v_t sampling from a normal distribution with mean $\mu = \frac{q_t}{N-t}$ and standard deviation $\delta = \frac{q_t}{N-t}$. Alternatively, with probability $1-\epsilon$, the agent chooses the optimal Q-action as the one that maximises the Q-values from Q_{main} net, thus exploiting what learnt in the exploration phase. We bind the agents to sell all their inventory within the considered time window, still exploring a large number of states, rewards, and actions.

Once every m actions taken during training iterations by both the agents, ϵ is multiplied by a constant $c < 1$ such that $\epsilon \to 0$ as $m \to \infty$. In this way, for a large number of iterations C, ϵ converges to zero and the algorithm gradually stops exploring and starts to greedily exploit what the agent has learned in terms of weights θ of the Q_{main} net. Notice that the update for ϵ happens at the same rate for both the agents, thus they explore and exploit contemporaneously, while the draw from the uniform distribution ζ is different for every agent. The action decision rule in the training phase unfolds, for each agent $i = 1, 2$, as:

$$\begin{aligned} &\epsilon \in (0,1) \ , \ \zeta_i \sim \mathcal{U}(0,1) \\ &v_t^{(i)} = \begin{cases} \sim \mathcal{N}(\mu = \frac{q_t}{N-t}, \delta = \frac{q_t}{N-t}) & \text{, if } \zeta \leq \epsilon \\ \arg\max_{v' \in [0, q_0]} Q_M(g_t^i, v' | \theta_{\text{main}}) & \text{, else} \end{cases} \end{aligned} \tag{5.22}$$

The actions chosen following this rule lie on the strategy space for each agent, which is the set of functions:

$$\pi : (t, q_t, S_{t-1}) \mapsto v_t,$$

mapping the agent's state to a trading action. These policies, denoted by π, are implicitly encoded by the learnt Q-function, since the agent chooses at each state g_t the action $v_t = \arg\max_v Q(g_t, v; \theta)$. Thus, the strategy space corresponds to the function class that the Q-network can represent and optimise over. Notice that we do not account for the strategy space directly; indeed the Double Deep Q learning algorithm that we developed and used is an *off-policy* algorithm, thus we do not parametrise the policies but we aim at finding those actions that maximise the Q value which is the output of the Deep Neural Nets (the Q networks), one for each agent. Notice also that the *action space* for each agent i at each time step t consists of trading volumes set to be $v_t^{(i)} \in [0, q_0^{(i)}]$ Actions are also normalised during training.

After this, each agent calculates the reward as:

$$r_{t,i} = S_{t-1}v_{t,i} - \alpha v_{t,i}^2 \quad , i = 1, 2 \tag{5.23}$$

Notice that the actions of the other agent indirectly impact on the reward of the agent i through the price S_{t-1}. The price comprises both agents actions at the end of time step t. Individually, each agent has knowledge of their own action at that time step, $v_{t,i}$, along with the mid-price S_{t-1}. Therefore, in the reward dynamics we do not allow any *direct* knowledge to be shared among the agents.

Overall, the rewards for every $t \in [1, N]$ are:

$$\begin{aligned} r_{1,i} &= S_0 v_{1,i} - \alpha v_{1,i}^2 \\ r_{2,i} &= (S_1 + \kappa(v_{1,i} + v_{1,i^-}) + \xi)v_{2,i} - \alpha v_{2,i}^2 \\ &\vdots \\ r_{N,i} &= (S_{N-1} + \kappa(v_{N-1,i} + v_{N-1,i^-}) + \xi)v_{N,i} - \alpha v_{N,i}^2 \end{aligned} \tag{5.24}$$

Where by i^- we denote the the other agent, assuming that we are looking at the reward for agent i, and $\xi \sim \mathcal{N}(0, \sigma)$.

Thus, for each time step t the agent i sees the reward from selling $v_{t,i}$ shares, and stores the state of the environment, $g_{t,i}$, where the sell action was chosen along with the reward and the next state $g_{t+1,i}$ where the environment evolves. At the end of the episode, the reward per episode per agent is $-q_0 S_0 + \sum_{t=1}^{N} S_{t-1}v_{t,i} - \alpha v_{t,i}^2$. Written in this form, the aim of the agent is to cumulatively *maximise* such reward, so that the liquidation value of the inventory occurs as close as possible to the initial value of the portfolio.

Training scheme

Table 5.1: Fixed parameters used in the DDQL algorithm. The parameters not shown in the table change depending on the experiments and are reported accordingly.

DDQL parameters				**Model parameters**	
NN layers	5	C train its.	$5,000$	N intervals	10
Hidden nodes	30	M test its.	$2,500$	S_0 price	10\$
ADAM lr	0.0001	m reset rate	75 acts.	q_0 inventory	100
Batch size b	64	c decay rate	0.995	α t. impact	0.002
L mem. len.	$15,000$	γ discount	1	κ p. impact	0.001

Bearing in mind that the procedure is exactly the same for both agents, we focus now on

the training scheme of one of them, dropping the subscript i. We let the states g_t, actions v_t, rewards r_t and subsequent future states g_{t+1} obtained by selling a quantity v_t in state g_t, to form a *transition* tuple that is stored into a memory of maximum length L, we have two different memories, one for each agent. As soon as the memory contains at least b transitions, the algorithm starts to train the Q-nets. To this end, the algorithm samples random batches of length b from the memory of the individual agent, and for each sampled transition j they individually calculate:

$$y_t^j(\theta_{\text{tgt}}) = \begin{cases} r_t^j & \text{if } t = N; \\ r_t^j + \gamma Q_{\text{tgt}}(g_{t+1}^j, v^*|\theta_{\text{tgt}}) & \text{else} \quad . \end{cases} \tag{5.25}$$

In Eq. (5.25), r_t^j is the reward for the time step considered in the transition j, g_{t+1}^j is the subsequent state reached in $t+1$, known since it is stored in the same transition j, while $v^* = \arg\max_v Q_{\text{main}}(g_t^j, v|\theta_{\text{main}})$. In this context γ is a discount factor. Each agent individually minimises the mean squared error loss between the target $y(\theta_{\text{tgt}})$ and the values obtained via the Q_{main} net. In formulae:

$$L(\theta_{\text{main}}, \theta_{\text{tgt}}) = \frac{1}{b}\sum_{\ell=1}^{b}\left(\left[y_t^\ell(\theta_{\text{tgt}}) - Q_{\text{main}}(g_t^\ell, v_t^\ell|\theta_{\text{main}})\right]\right)^2$$

$$\theta^*_{\text{main}} = \arg\min_{\theta_{\text{main}}} \mathcal{L}(\theta_{\text{main}}, \theta_{\text{tgt}})$$

We then use back propagation and gradient descent in order to update the weights of the Q_{main} net. This procedure is repeated for each agent and for each random batch of transition sampled from the agent's memory. Overall, once both agents have individually performed m actions, we decrease ϵ by a factor $c < 1$ and we set $Q_{\text{tgt}} = Q_{\text{main}}$

Once both agents have been simultaneously trained to optimally execute a quantity q_0 while interacting only through the midprice, we let the agents interact on another number $M < C$ of trading iterations. This is the *testing phase*, and now actions for each agent are selected using just her Q_{main} net. As said above, the results analysed below are those obtained in the testing phase.

The features of the Q-nets for each agent are $(q_{i,t}, t, S_{t-1}, v_{i,t})$ and are normalised in the domain $[-1, 1]$ using the procedure suggested in Ning et al. [2021], whereas normalised midprices $\bar{S}_t$ are obtained via min-max normalisation.

When considering the inputs to the Q-networks, we choose this set of features for two main reasons. The first one is more theoretical. We want to model the interaction between the agents in such a way that there is no direct exchange of information about the strategy followed by either of them. With this choice of features, the only available 'shared' information is the one on the the mid-price level of the traded security, thus no direct communication between the two agents happens. The second reason is more practical. On the one hand we want to be parsimonious with the set of features seen by the agent, and since the environment where the agents trade is a multi-agent Almgren and Chriss model, the inventory and the time step along with the traded quantity, would suffice as features since the problem would be solved with strategies that do not depend on the level of mid-price. However, as shown in Ning et al. [2021], Macrì and Lillo [2024] adding additional features that theoretically should be irrelevant, practically help the algorithm learning the optimal trading strategy faster.

In our setup we use fully connected feed-forward neural networks with 5 layers, each with 30 hidden nodes. The activation functions are leakyReLu, and finally we use ADAM for optimisation. With the exception of the volatility parameter σ which will be specified later, the parameters used in the algorithm are reported in Table 5.1, the training algorithm is reported in Algorithm 7.

Algorithm 7 Training of Double Deep Q-Learning multi-agent impact game

Require:

Set $\epsilon = 1$, b batch size, C train iterations;
Set market dynamics parameters;
For each agent $i = 1, 2$ initialise with random weights Q_{main} and make a copy Q_{tgt} ;
For each agent $i = 1, 2$ initialise the memory with max length L.

for k in C **do**
Set $S_0^k = S_0$;

for t in N **do**
$u \sim \text{Bin}(1, 0.5)$; ▷ Decides the order of execution

if $u = 0$ **then** $i = (1, 2)$ else $i = (2, 1)$
end if ▷ Vector of priority ordering of the agents
for each agent i in the order decided above **do**
$g_{i,t} \leftarrow (q_{i,t}, t, S_t^k)$;
$v_{i,t} \leftarrow \begin{cases} \text{sample } \mathcal{N}(\mu = \frac{q_t}{N-t}, \delta = \frac{q_t}{N-t}) & \text{, with probability } \epsilon \\ \arg\max_{v' \in [0, q_0]} Q_M(g_t, v' | \theta_{\text{main}}) & \text{, with probability } (1 - \epsilon) \end{cases}$;
$r_t \leftarrow S_{t-1}^i v_t - \alpha v_t^2$;

$S_{t-1}^k \to S_t^k$; ▷ Generate S_t^k from S_{t-1}^k

$g_{i,t+1} \leftarrow (q_{i,t+1}, t + 1, S_t^k)$;

Memory for agent $i \leftarrow (g_{i,t}, r_{i,t}, v_{i,t}, g_{i,t+1})$; ▷ Memory storing

if Length of memory $\geq b$ **then**
for j in b **do**
Sample a batch of $(g_{i,t}^j, r_{i,t}^j, v_{i,t}^j, g_{i,t+1}^j)$ from memory;
$v_i^* = \arg\max_v Q_{i,\text{main}}(g_{i,t}^j, v_{i,t}^j | \theta_{i,\text{main}})$;
$y_{i,t}^j(\theta_{i,\text{tgt}}) = \begin{cases} r_{i,t}^j & \text{if } t = N; \\ r_{i,t}^j + \gamma Q_{i,\text{tgt}}(g_{i,t+1}^j, v_i^* | \theta_{i,\text{tgt}}) & \text{else} \end{cases}$
end for

$\theta_{i,\text{main}}^* = \arg\min_{\theta_{i,\text{main}}} \mathcal{L}(\theta_{i,\text{main}}, \theta_{i,\text{tgt}})$ via gradient descent with loss to minimise:

$$L(\theta_{i,\text{main}}, \theta_{i,\text{tgt}}) = \frac{1}{b} \sum_{\ell=1}^{b} \left(\left[y_t^\ell(\theta_{i,\text{tgt}}) - Q_{i,\text{main}}(g_{i,t}^\ell, v_{i,t}^\ell | \theta_{i,\text{main}}) \right] \right)^2$$

if Length of agent i memory $= L$ **then** halve the length of agent i memory
end if
end if
After m iterations decay $\epsilon = \epsilon \times c$;
After m iterations $\theta_{i,\text{tgt}} \leftarrow \theta_{i,\text{main}}$;
end for
end for
end for

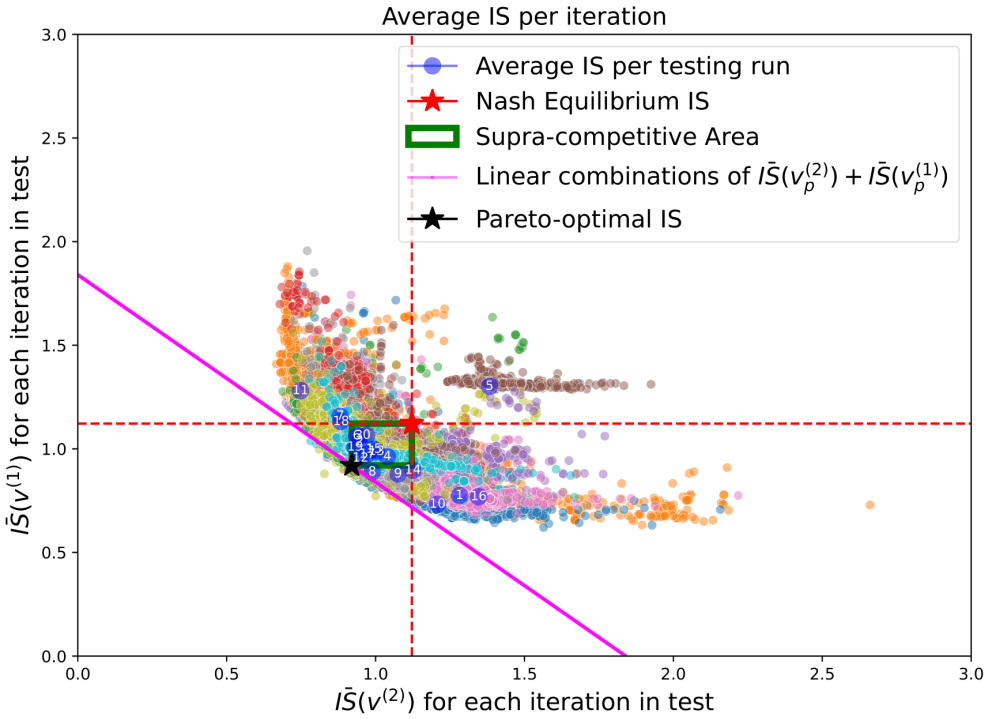


Figure 5.1: Scatter plot of the IS of the two agents for 20 testing runs of 2,500 iterations in the zero noise case ($\sigma = 10^{-9}$).

5.4 Results

In this Section we present the results obtained by using the RL algorithm shown in Algorithm 7 in the game setting outlined in Section 5.2. The experiments aim at studying the existence and the form of the equilibrium strategies and of their costs, learnt by the agents during the experiments, this is done by analysing the policies chosen by the two interacting trading agents. We then compare the equilibria under the learnt dynamics with the Nash equilibrium strategies, the Pareto-efficient set of solutions, and the Pareto optimal strategy. We consider different scenarios, using the market structure as outlined in Eq. (5.4).

It can be easily noticed how the theoretical Nash equilibrium for this game does not depend on the volatility level of the asset. In fact, when the risk aversion parameter $\lambda = 0$, the Nash equilibrium of Eq. (5.7) depends only on the permanent and temporary impact coefficients κ and α, respectively. However, in the numerical determination of the equilibrium solution, volatility σ and the associated diffusion play the role of a disturbance term, since an agent cannot determine whether an observed price change is only due to the impact generated by the trading of the other agent or if it is a random fluctuation due to volatility. In this sense, volatility plays the

role of a *noise-to-signal* ratio here. One can interpret the level of σ as a measure of presence of noise traders that are randomly affecting the price while the two agents do optimal execution. In this sense, we could also consider σ as a measure of market liquidity. To quantify the effect of volatility on the learnt equilibria, we perform three sets of experiments for different levels of the volatility parameter σ, leaving unchanged the impact parameters. More specifically we consider, for both training and testing phase the case where $\sigma = 10^{-9}$ (termed *zero noise* case), $\sigma = 10^{-3}$ (*moderate noise*) and $\sigma = 10^{-2}$ (*large noise*) case. In all three cases, we use the same temporary and permanent impacts $\alpha = 0.002$ and $\kappa = 0.001$.

Other preliminary analyses on the behaviour of the agents when one agent follows a Nash or a Pareto strategy, and the other learns a reaction strategy can be found in Appendix B.1. In what follows we will only concentrate on the simultaneous multi-agent learning problem.

We employ 20 training and testing runs, each run is independent from the others, meaning that the weights found in one run are not used in the others. In this way we aim at independently (run-wise) train the agents to sell their inventory. Each testing run has $M = 2,500$ iterations of $N = 10$ time-steps.

5.4.1 The *zero noise* case

When $\sigma = 10^{-9}$ we model the interaction of both agents in a limiting situation when the market is very illiquid and almost no noise traders enter the price formation process. In this case, the permanent impact is essentially the only driver of the mid-price dynamics in Eq. (5.4), thus price changes are triggered basically only by the selling strategies used by the agents throughout the game iterations.

The results of the experiments are displayed in Figure 5.1, which shows, as a scatter plot, the IS of both agents in each iteration of the testing phase. Each colour represents the results of one of the 20 runs of $2,500$ iterations each. The blue circles show the IS centroids per testing run (the number in the circle identifies the run). For comparison the plot reports as a red star the point corresponding to the Nash equilibrium of Eq. (5.7), which is used to divide the graph into four quadrants, delimited by the red dashed lines. The top-right quadrant contains points which are sub-optimal with respect to the Nash equilibrium for both players, while the points in the top-left and bottom-right quadrant report points where the found equilibrium favours one of the two agents at the expenses of the other. In particular, in these regions one of the agents achieve an IS smaller than the one of the Nash equilibrium, while the other agents

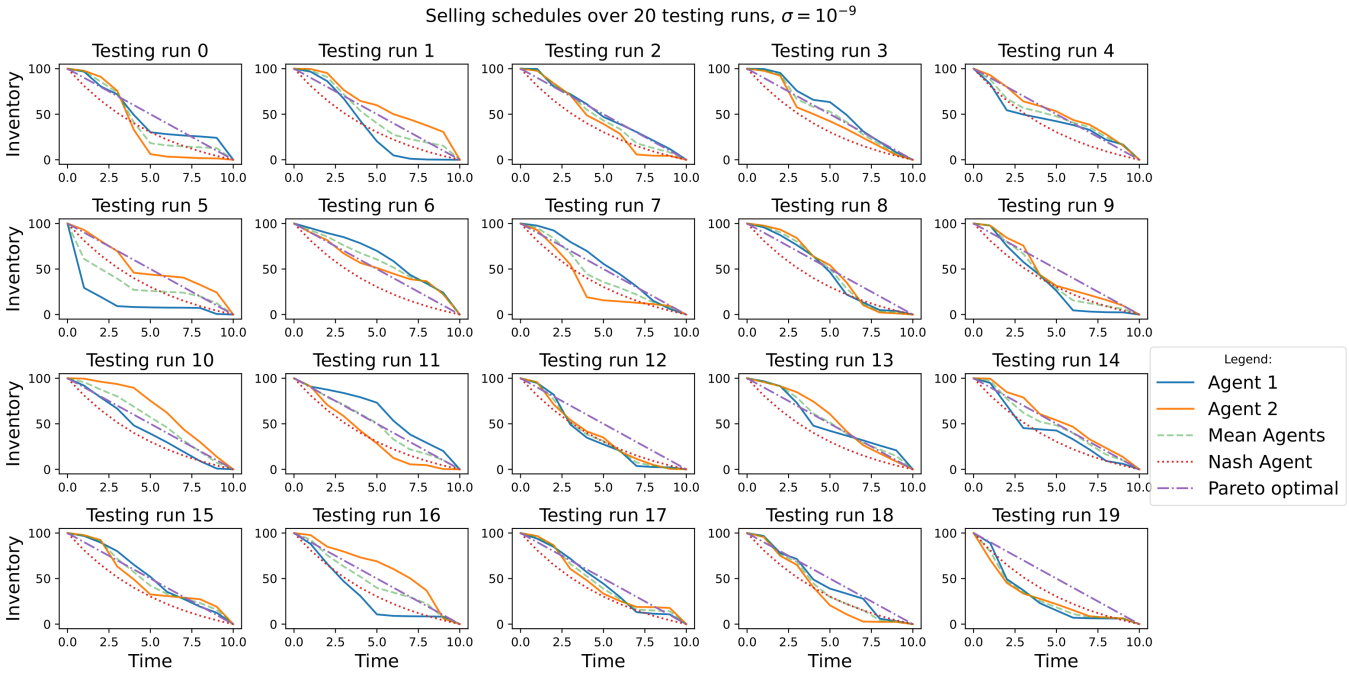


Figure 5.2: Optimal execution strategies for 20 testing runs of 2,500 iterations, using $\sigma = 10^{-9}$.

performs worst. In a sense, one of the agents predates on the other in terms of reward. The bottom-left quadrant is the most interesting, since here *both* agents are able to achieve an IS smaller than the one in the Nash equilibrium and thus contain supra-competitive IS costs that are potential collusive equilibria. For comparison, the black star indicates the IS of the Pareto-optimal strategy found in Theorem 2, while the magenta line is the linear combination of the Pareto-optimal IS. Finally, the green rectangle denotes the area between the Nash and the Pareto-optimal ISs, and we call it the *supra-competitive area*. In fact, in that area we find the ISs for both the agents that lie between the *collusion* defined by the Pareto-optimum and the Nash equilibrium. It is the area where, although the IS costs achieved do not overlap perfectly with the Pareto optimal one, thus the collusion, are still supra-competitive.

Looking at Figure 5.1 we can notice that the ISs per testing run concentrate in the supra-competitive area of the graph (green rectangle) between the Nash and the Pareto-optimal IS. This means that when the noise is minimal, it is easier for the agents to adopt strategies whose cost levels are supra-competitive and potentially consistent with tacit collusive strategies. The agents are able to obtain costs that fall near the collusion IS, i.e. the Pareto-optimum. The majority of the remaining ISs per iteration still lie within the second and fourth quadrant of the graph, meaning that again they are in general able to find strategies that, at each iteration, do not allow one agent to be better without worsening the other.

Considering the strategies found by the agents in Figure 5.2, we notice how the agents trade at different speeds, i.e. their selling policies are consistent with the presence of a slow trader and a fast trader. It can be seen how the policies followed by the agents in almost all the 20 simulations depend inversely on the strategy adopted by the competitor. Moreover, in most of the runs in the supra-competitive region, the *average* strategy of the agents is very similar to the TWAP strategy. This is possible thanks to very low noise, and in this case the agents are able to find more easily the Pareto-optimal strategy, which corresponds to an equilibrium where they both pay the lowest amount possible in terms of IS.

These evidences underline the quite intuitive fact that, when the noise-to-signal ratio is low it is simple for the agents to disentangle their actions from those of other agents, thus making it easier for the agents to learn and possibly infer the strategy of the competitor. The agents find an incentive to deviate from the Nash-equilibrium adopting strategies that allow for lower costs. The majority of these strategies correspond to an average supra-competitive IS, thus with lower cost than in the Nash equilibrium, but still slightly greater than the Pareto-optimum in terms of IS. In general, per iteration neither agent can be better off without increasing the other agent cost level.

Coupling the cost plot in Figure 5.1 with the average selling schedules in Figure 5.2, it can be seen how the centroids that relate to higher costs for one of the agents correspond to a deviation from the TWAP strategy by either of them, such deviation corresponds to another deviation by the other agent, that substantiates in a faster selling rate. There is, on the one hand, evidence of supra-competitive behaviour, on average, when IS costs are considered, which in turn is reflected in the way the agents trade at different speeds. The phenomenon is evident thanks to the very low level of asset's volatility. Thus, hinting to tacit collusive behaviour in this first case, judging by the supra-competitive cost outcomes. In Section 5 we thoroughly discuss this point.

5.4.2 The *moderate noise* case

In this setting, the mid-price dynamics is influenced both by the trading of the agents and by the volatility. The first thing that can be noticed in Figure 5.3 is that the IS per iteration distributes almost only in the second, third and fourth quadrant of the plot. Moreover, the distribution of the points suggests again an inverse relation between the ISs of the two agents, i.e. a lower IS of an agent is typically associated with a worsening of the IS of the other agent.

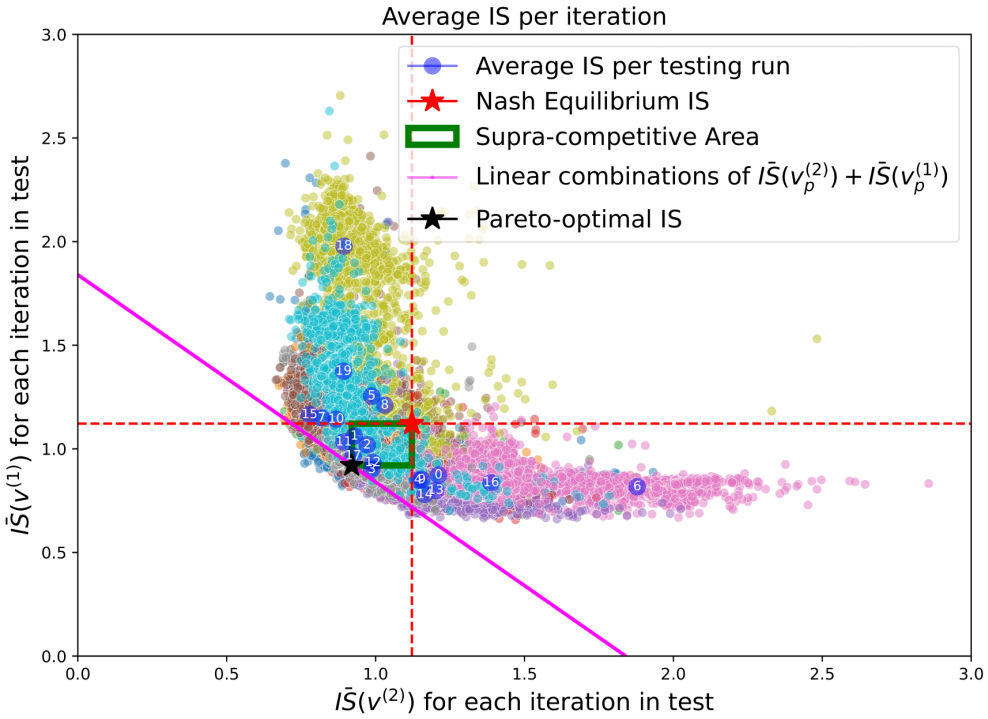


Figure 5.3: Scatter plot of the IS of the two agents in 20 testing runs of $2,500$ iterations in the moderate noise case ($\sigma = 10^{-3}$).

The centroids distribute accordingly. In fact, the centroids that lie in the supra-competitive area of the graph are as numerous as those that lie outside of that area on either the second or fourth quadrant. It can be seen that, outside the green rectangle, the agents tend to behave in a *predatory* way, meaning that one agent has consistently lower costs than the other. This happens tacitly, meaning that no information about the existence or about the strategy followed by the other agent is part of the information available either in the market or in each agent's memory. Notice that predatory strategies are still consistent with the definition of Pareto efficiency but are *not a proper collusion* since they do differ from the Pareto-optimal IS, even if they appear to overlap in Figure 5.3.

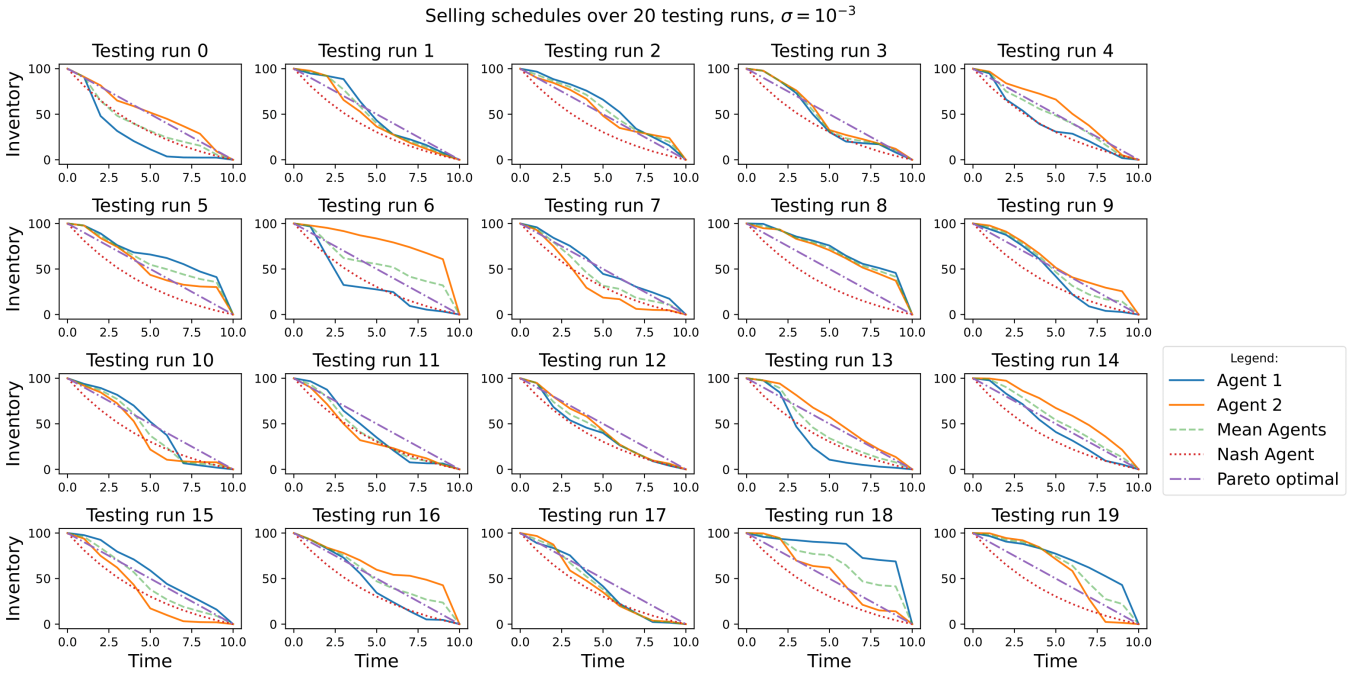


Figure 5.4: Optimal execution strategies for 20 testing runs of $2,500$ iterations, using $\sigma = 10^{-3}$.

Looking at the average strategies implemented by the agents in each testing run (Figure 5.4), it can be noticed how as in the *zero noise* case, roughly speaking there is almost always one agent that trades faster than the other and, again as in the previous case, the one agent that trades slower is the one with a higher IS. The comparison of Figure 5.4 and Figure 5.3 shows that the agent that 'predates' the other is a fast trader and obtains lower costs of execution. This behaviour is more pronounced in this case, and we postulate that this is due to the increased level of noise for this experiment.

We conclude once again that, even if no explicit information about the trading strategies is shared by the agents, during the repeated game they are able to extrapolate information on the policy pursued by the competitor through the changes in the mid-price triggered by their own trading and by the other agent's trading. Thus, it seems plausible that RL-agents modelled in this way are able to extrapolate information on the competitor's policy and tacitly interact through either a plausibly cooperative behaviour that leads to supra-competitive costs or through a predatory behaviour where the costs of one agent are consistently higher than those of the other. Generally speaking, the set of solutions achieved is in line with the definition of the Pareto efficient set of solutions.

5.4.3 The *large noise* case

Finally, we study the interactions between the agents when the volatility of the asset is large. This corresponds to a relatively liquid market, since price changes are severely affected by the volatility level and the price impact triggered by the agents is low in comparison. Looking at Figure 5.5, we notice that in this market setup the higher level of volatility significantly affects the distribution of the IS per iteration. In fact, the points in the scatter plot now distribute obliquely, even if for the most part they still lie in the second, third, and fourth quadrants. Because of the higher volatility level, very low IS values for both agents might be attained per iteration and the structure of the costs per iteration is completely different with respect to to the previous two cases. However, the distribution of the average ISs per testing run, i.e. the positions of the centroids, is still concentrated for the greatest part in the strategies' area in between the Pareto-optimum and the Nash equilibrium costs.

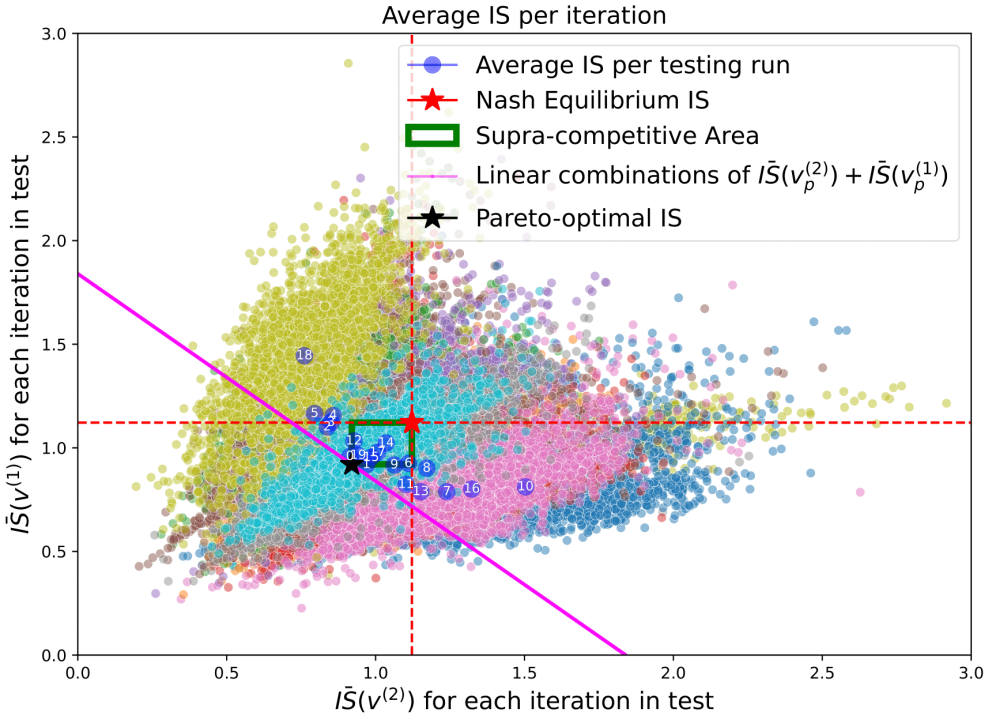


Figure 5.5: Scatter plot of the IS of the two agents for 20 testing runs of $2,500$ iterations in the large noise case ($\sigma = 10^{-2}$).

Figure 5.6 shows the trading strategies for the agents. It can still be noticed how in general if one agent is more aggressive with its trading, the other tends to be not. This behaviour of the agents, resulting in faster and slower traders, still takes place and again the agent that is trading slower pays more in terms of IS. Comparing the centroids' distribution and the selling

schedules in Figure 5.6 and Figure 5.5 reveals how a slower trading speed worsens the cost profile of one agent to the benefit of the other. The average trading strategy of the agents per testing run in the supra-competitive cases basically revolves around a TWAP strategy that is in turn the Pareto-optimum for the problem considered.

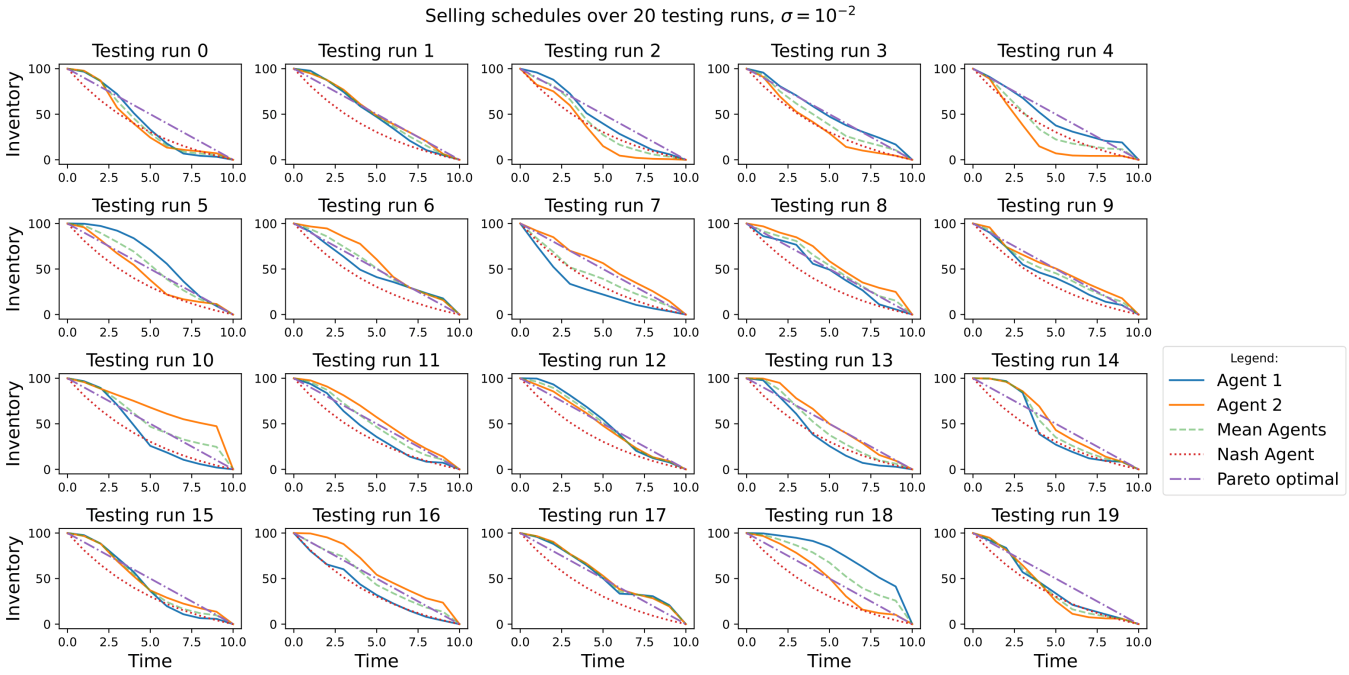


Figure 5.6: Optimal execution strategies for 20 testing runs of $2,500$ iterations, using $\sigma = 10^{-2}$.

5.4.4 Summary of results and comparison with the Pareto front

We have seen above that especially in the presence of significant volatility, the points corresponding to the iterations in a run tend to distribute quite widely in the scatter plot. The centroid summarises the average behaviour in a given run and provides a much more stable indication of the relation between the IS of the two agents. To have a complete overview of the observed behaviour across the three volatility regimes, in Figure 5.7 we show in a scatter plot the position of the centroids of the 20×3 testing runs. It can be seen that the centroids tend to concentrate in the supra-competitive area, irrespective of the volatility level for the experiment. We notice how the agents' costs tend to cluster in this area and to be close to the Pareto-optimal IS. To have a more detailed comparison of the simulation results with the theoretical formulation of the game, we numerically compute the Pareto-efficient set of solutions (or Pareto-front)[9] and represent it on the scatter plot. The Figure shows that, as expected, the

[9]The numerical Pareto front has been obtained using 'Pymoo' package in Python introduced in Blank and Deb [2020].

centroids lie on the top-right of the front. More interestingly, they are mostly found between the front and the Nash equilibrium for all the levels of volatility and, in line with the definition of the Pareto-front, for an agent to get better the other has to be worse off. We further notice that the majority of points for the zero noise case lies very close the Pareto optimum or in the supra-competitive area, thus the lower the volatility the easier it is to converge to a supra-competitive strategy. In the other two cases we notice how, even roughly half of the centroids lie in the supra-competitive area, the rest mostly lie near the numerical Pareto front.

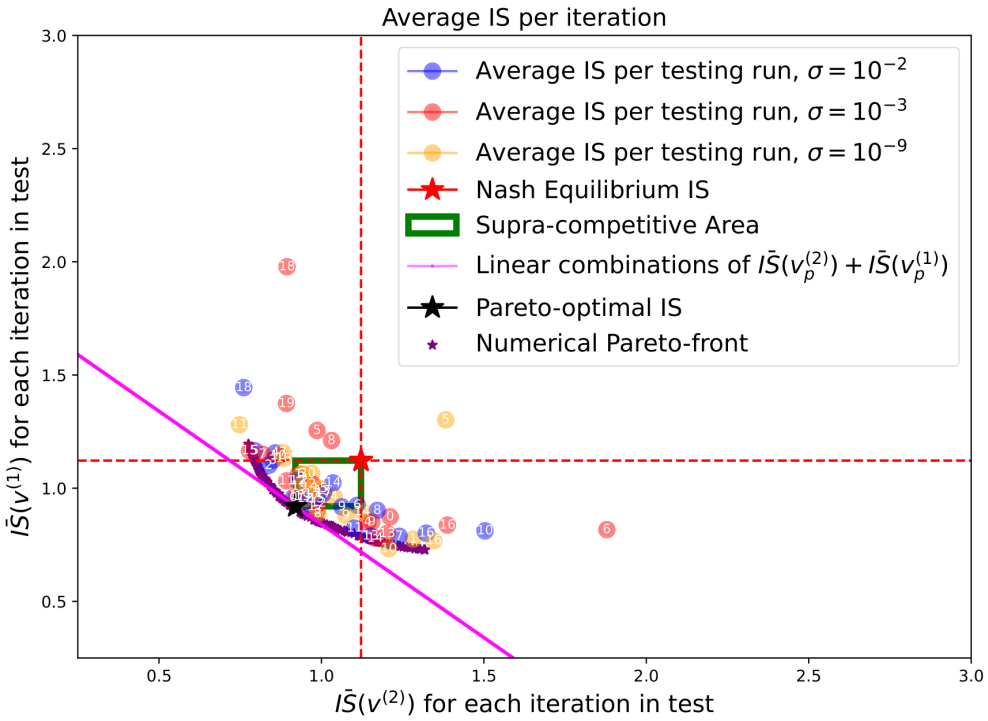


Figure 5.7: Scatter plot of the IS centroids of the two agents in the 20×3 testing runs of $2,500$ iterations each, for all the considered values of σ.

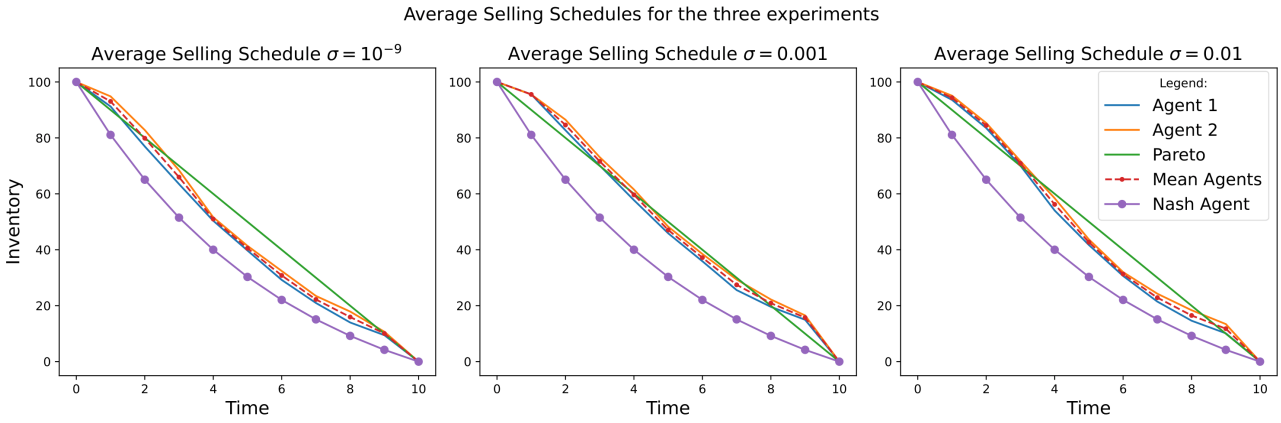


Figure 5.8: Average optimal execution strategy over the 20 testing runs of $2,500$ iterations, for all the values of σ considered.

Figure 5.8 shows that, irrespective of the values considered for the volatility level σ, the average strategy does not differ much from the Pareto-optimum TWAP strategy. In the *zero noise* case, the average strategy of the two agents tends to be slightly more front loaded, i.e. they tend to liquidate more at the beginning of the trading schedule, but still lies between the Nash and the Pareto-optimum. As the σ value increases, and thus in the *moderate noise* and *large noise* cases, we can see how both the agents tend to be less aggressive at the beginning of their execution in order to adopt a larger selling rate towards the end of their execution. This behaviour is stronger the higher the σ, due to the increased uncertainty brought by the asset volatility in addition to the price movements triggered by the trading of both the agents.

5.4.5 Variable volatility and misspecified dynamics

Our simulation results show that volatility plays an important role in determining the IS in an episode of the testing phase, although when averaging across the many iterations of a run, the results are more stable and consistent. In financial market, returns are known to be heteroscedastic, i.e. volatility varies with time. Thus, also from the practitioner's point of view, it is interesting to study how agents, which are trained in an environment with a given level of volatility level, perform in a testing environment where the level of volatility is different. In particular, it is not a priori clear what behaviour among the two agents would naturally arise when the price dynamics is different in the testing and in the training phase. Moreover, it is interesting to study how a change in the environment would impact the overall selling schedule and the corresponding costs. In the following, we study the agents' behaviour in extreme cases, in order to better appreciate their behaviour under time-varying conditions. Specifically, using the same impact parameters as above in both phases, we learn the the weights of the DDQN algorithm in a training setting with $\sigma = 10^{-9}$ and then we employ them in a testing environment where $\sigma = 10^{-2}$. We then repeat the experiment in the opposite case with the volatility parameters switched, i.e. training with $\sigma = 10^{-2}$ and testing with $\sigma = 10^{-9}$. We now run 10 testing runs of $2,500$ iterations each.

Training with *zero noise* and testing with *large noise*

When the agents are trained in an environment where $\sigma = 10^{-9}$ and the weights of this training are used in a testing environment with $\sigma = 10^{-2}$, we find that the distribution of the ISs per iteration is similar to the one that would be obtained in a both testing and training scenario

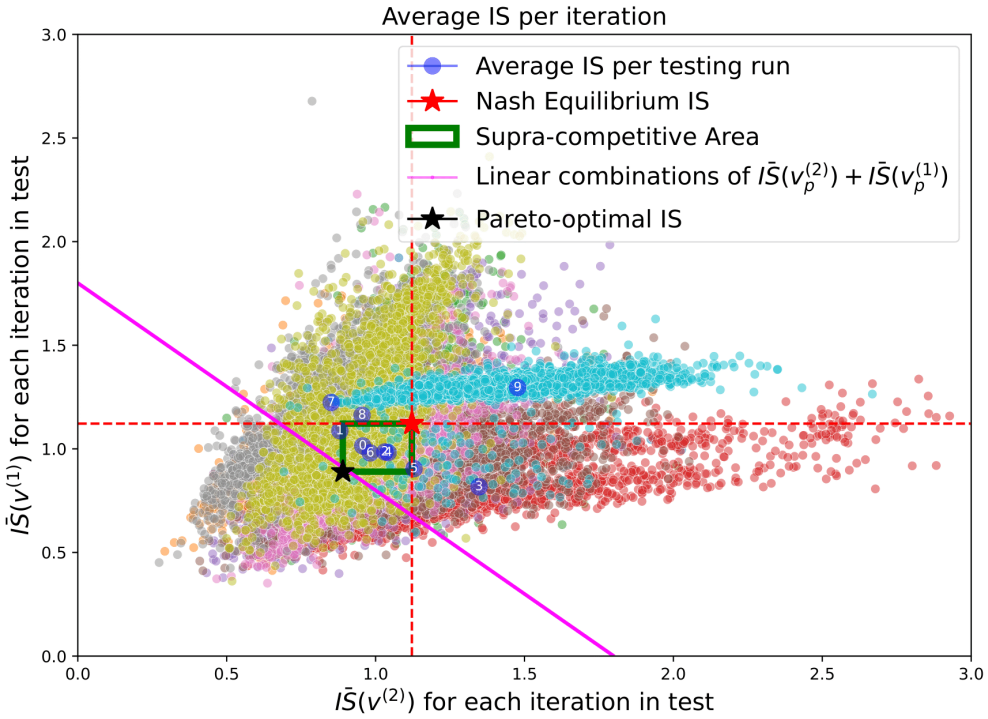


Figure 5.9: IS scatter for 20 testing runs of 2,500 iterations, training $\sigma = 10^{-9}$ and testing $\sigma = 10^{-2}$.

with $\sigma = 10^{-2}$ (see Figure 5.9). The centroids are still mostly distributed in the second, third and fourth quadrant, although some iterations and even a centroid, might end up in the first quadrant as in the $\sigma = 10^{-9}$ case. In general, it can be seen how the centroids mostly lie in the, supra-competitive area of the graph, i.e in the square between the Pareto-optimal IS and the Nash equilibrium, pointing out at the fact that supra-competitive cost outcomes and strategies are still attainable even when the training dynamics are misspecified with respect to the testing ones.

Looking at Figure 5.10, we notice that the selling schedules are mostly intertwined, i.e. rarely the agents trade at the same rate. Traders can still be slow or fast depending on the rate adopted by the other agent. Comparing Figure 5.10 with Figure 5.9 we see that, as in the correctly specified case, the agent that trades faster gets the lower cost, while the slow trader achieves a larger IS.

Training with *large noise* and testing with *zero noise*

Figure 5.11 shows the result of the experiment where agents are trained in an environment where $\sigma = 10^{-2}$ and tested in a market with $\sigma = 10^{-9}$. We observe that the ISs are distributed in a way similar to the case where the agents were both tested and trained in an environment

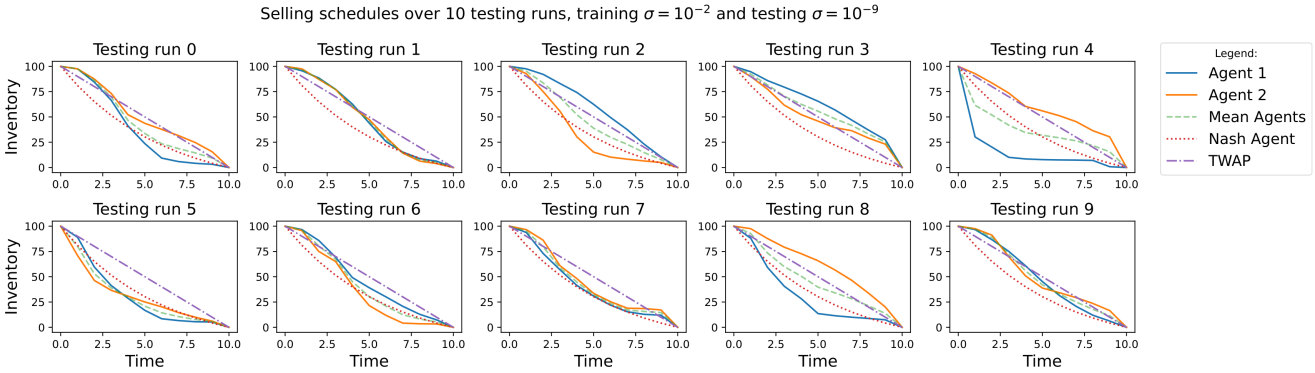


Figure 5.10: Optimal execution strategies for 10 testing runs of 2,500 iterations, training $\sigma = 10^{-9}$ and testing $\sigma = 10^{-2}$.

with $\sigma = 10^{-9}$. Still, the centroids lie in all but the first quadrant, and the majority of them lies in the supra-competitive area, showing how even with misspecified dynamics, supra-competitive strategies naturally arise in this game. The trading schedules adopted show the same kind of fast-slow trading behaviour between the agents, where still the slower trader has to pay higher costs in terms of IS (see Figure 5.12).

Finally, irrespective of the encountered levels of volatility, it seems that what is more important is the market environment experienced during the training phase, thus the selling schedule implemented in a high volatility scenario with DDQN weights coming from a low volatility scenario are remarkably similar to the one found with both training and testing with $\sigma = 10^{-9}$, and vice versa when $\sigma = 10^{-2}$ is used in training and the low volatility is encountered in the test. It might be concluded that once that the agents learn how to adopt a supra-competitive behaviour in one volatility regime, they still behave supra-competitively even if they are dealing with another volatility regime.

5.5 Real tacit collusion or supra-competitive solution?

According to Harrington [2018], «Collusion is when firms use strategies that embody a reward-punishment scheme that rewards a firm for abiding by the supra-competitive outcome and punishes it for departing from it». Are we looking at real collusive behaviour then? Judging by the average trading strategies in Figure 5.8 one can infer that the agents have learnt to adopt a strategy compatible with a supra-competitive equilibrium, very close to the Pareto optimal strategy. Hence, from a 'global' point of view, the strategies suggest a cooperative behaviour by the agents that is directed at obtaining lower costs if compared to the Nash equilibrium

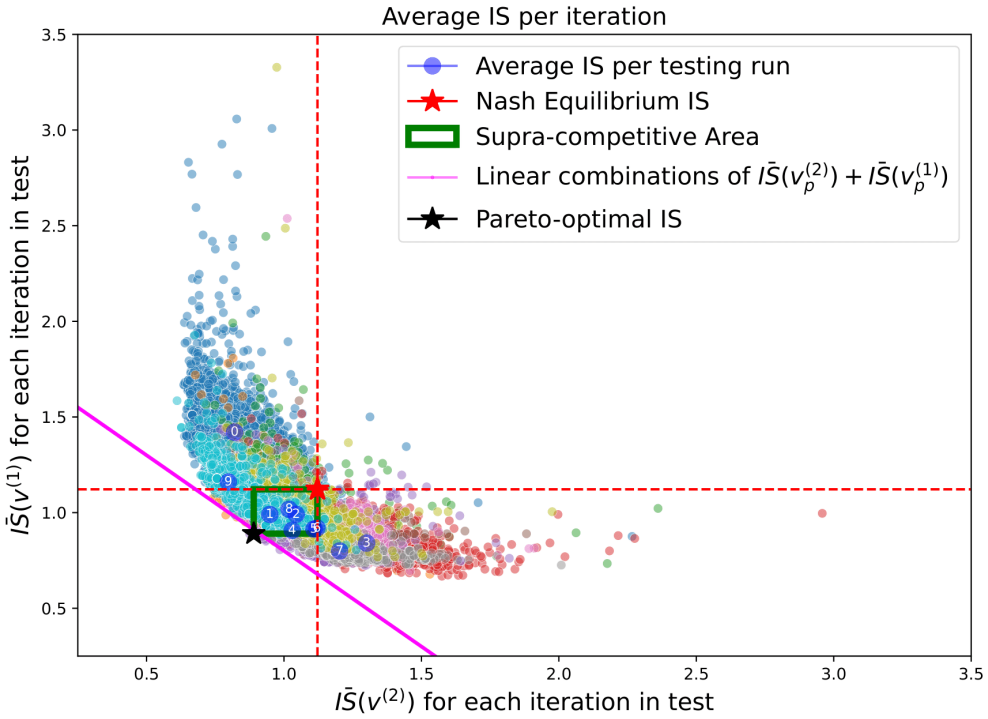


Figure 5.11: IS scatter for 20 testing runs of $2,500$ iterations, training $\sigma = 10^{-2}$ and testing $\sigma = 10^{-9}$.

strategy, as this strategy jointly minimises the costs for *both* the agents collectively. But this alone is not sufficient to talk about collusion, although tacit.

If we consider a more 'local' point of view, run-wise we see how the average of the testing iterations results in centroids that do distribute along the Pareto efficient set of solutions as in Figure 5.7, and this depends on the strategies that, as we saw, show different speeds of trading per agent. In fact, comparing the centroids in Figure 5.3 with the average trading patterns in Figure 5.4, it appears that the points in upper (lower) branch of the Pareto front correspond to cases where agent 1 trades slowly (fast) achieving a higher (lower) cost, while the opposite is true for agent 2. Thus, one can conjecture that the different speeds at which the agents trade in each iteration are a response to that deviation from the Pareto-optimal strategy, as they lie over the efficient frontier that is the set of solutions to the multi-objective optimisation problem. Moreover, symmetry breaks observed in the strategies might be possibly explained by to the ϵ-greedy exploration phase, as one agent explores while the other might exploit according to the parameter ϵ. In particular, agents independently explore and/or exploit the space of admissible strategies, keeping track of rewards associated with the volumes executed at each time step having no knowledge on either the presence or the strategy of the competing agent. This imperfect information while learning along with the explorative phase should be the reason

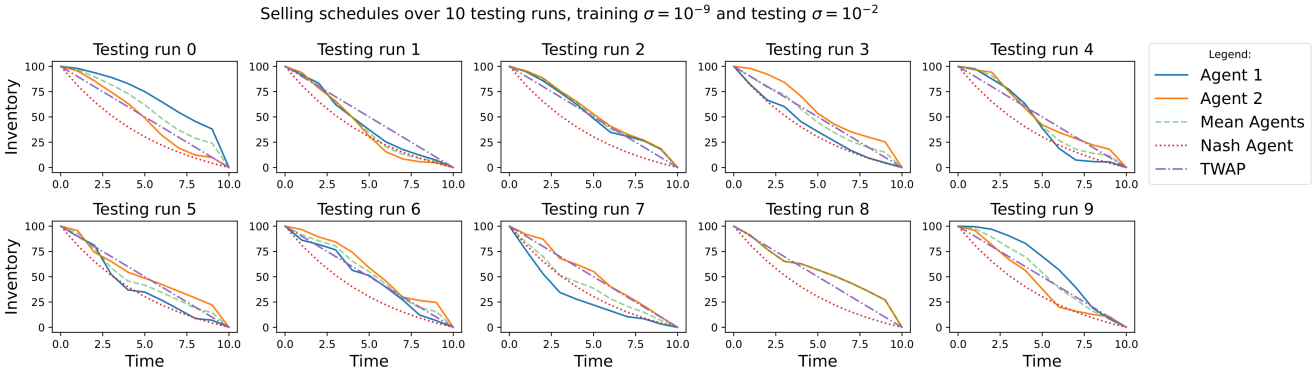


Figure 5.12: Optimal execution strategies for 10 testing runs of $2,500$ iterations, training $\sigma = 10^{-2}$ and testing $\sigma = 10^{-9}$.

of why the symmetry breaks, and fast-slow traders are observed. Although of great interest, a thorough investigation on how the design of the game might impact the learnt policies is left to future research endeavours.

Is it possible to interpret these deviations from the collusive strategy as a form of punishment? In our view, there is no simple answer to this question. On the one hand, considering the peculiar nature of the game itself, which is repeated and iterated because of the learning process, it is not possible to look at short deviations from the equilibrium and an eventual reversion to supra-competitive strategies, in the long run, because of a punishment mechanism. On the other hand, one can interpret the long run as a repetition of what was learned and observed in terms of the average strategy, which is in fact supra-competitive, as seen in Figure 5.8. In summary, we used theoretical results, definitions and theorems on Nash equilibrium leveraging on the contribution by Schied and Zhang [2017], the definition of collusion framed in Cont et al. [2021], Cont and Xiong [2022]. Regarding supra-competitiveness, we throughout the contribution we adopt the definition given in Dou et al. [2024], and relate it to the definition of collusion proposed in Harrington [2018]

However, in order to be conservative we cannot conclude that the observed behaviour is a tacit collusion in the sense of Harrington [2018]. Clearly more analyses are required to answer this question, possibly considering longer iterations allowing to identify evidences of punishment by shocking the action of one of the agent as in Calvano et al. [2020]. We leave this interesting investigation for future research.

As a final note, we highlight that such behaviour could cause welfare reduction for market participants, in terms of a distorted price formation process.

In fact, it is interesting to notice that the observed trading behaviour might have adverse

effects on other market participants' welfare. In the Nash equilibrium the two agents trade more at the beginning of the time interval, the way information is incorporated into the price is relatively fast. On the contrary, when the two agents are colluding they spread more evenly the execution over the time window cooperatively. In this way, private information is incorporated more slowly. Ceteris paribus, this has a negative impact on other market participants by distorting the price formation process, since the mid-price would not accurately describe the market demand and offer mechanism for the stock under execution. This would negatively affect the price formation process for other traders and eventually reduce the welfare of traders who trade based on information available on the market.

In conclusion, even if there no direct evidence of a reward-punishment scheme adopted by the agents, there are possibly negative consequences in terms of welfare at the expense of other market participants, while the agents that are doing optimal execution would benefit by cost reductions.

5.6 Conclusions

In this paper we have studied how supra-competitive strategies arise in an open loop two players' optimal execution game. We first have introduced the concept of collusive Pareto optima, that is, a vector of selling strategies whose IS is not dominated by the ISs achievable by other strategies for each iteration of the game. We furthermore showed how a Pareto-efficient set of solutions for this game exists and can be obtained as a solution to a multi-objective minimisation problem. Finally, we have shown how the Pareto-optimal strategy, that is indeed the collusion strategy in this setup, is the TWAP for risk-neutral agents.

As our main contribution, we have developed a Double Deep Q-Learning algorithm where two agents are trained using RL. The agents were trained and tested over several different scenarios where they learn how to optimally liquidate a position in the presence of the other agent and then, in the testing phase, they deploy their strategy leveraging on what learnt in the previous phase. The different scenarios, large, moderate and low volatility, helped us to shed light on how the trading interactions on the same asset made by two different agents, who are not aware of the other competitor, give rise to supra-competitive and Pareto-efficient strategies, i.e. strategies with a cost lower than a Nash equilibrium but higher than a collusion. This, in turn, is due to agents who keep trading at a different speed, thus adjusting the speed of

their trading based on what the other agent is doing. Agents do not interact directly and are not aware of the other agent's trading activity, thus strategies are learnt from the information coming from the impact on the asset price in a model-agnostic fashion.

Finally, we have studied how the agents interact when the volatility parameter in the training part is misspecified with respect to the one observed by the agents in the testing part. It turns out that the existence of supra-competitive strategies possibly consistent with a tacit collusive behaviour is still present, and thus robust with respect to settings when models parameters are time varying.

There are several possible extensions of our work. An obvious extension is to the setting where more than two agents are present and/or where more assets are liquidated, leading to a multi-asset and multi-agent market impact games, leveraging on the work done in Cordoni and Lillo [2024a]. Second, impact parameters are constant, while the problem becomes more interesting when liquidity is time-varying (see Macrì and Lillo [2024] for RL optimal execution with one agent). Third, we have considered a linear impact model, whereas it is known that impact coefficients are usually non-linear and might follow non-trivial intraday dynamics. The inclusion of non-linear impacts would allow for *dynamic arbitrages* and therefore policies that would result in negative expected costs.

On another note, as in this contribution the agents are modelled to be symmetric, with identical information and risk aversion; it would be interesting to model a break of this symmetry, for instance through heterogeneous risk aversion or asymmetric information. Clearly, this task would require a closed (or semi-closed) form solution to the Nash equilibrium for the two agents, in order to obtain a fair comparison of the results obtained with a well-established analytical solution.

Finally, the Almgren-Chriss model postulates a permanent and fixed impact, while many empirical results point toward its transient nature (see Bouchaud et al. [2009]). Interestingly, in this setting the Nash equilibrium of the market impact game shows price instabilities in some regime of parameters (see Schied and Zhang [2019]), which are similar to market manipulations. It could be interesting to study if different market manipulation practices might arise when agents are trained, as in this paper, with RL techniques accounting -within the Markovian setting of RL techniques- for the *transient nature* of the impact. The answer would certainly also be of great interest to regulators and supervising authorities.

B.1 Further learning experiments

In this appendix we briefly analyse the behaviour of the algorithm outlined in Section 5.3 in two different scenarios. In the first case we analyse the behaviour of two independent agents, we consider the same scenario where both the agents have to solve the optimal execution problem in a market impact game but we let them explore following a different exploration rule. In the second case we change the scenario and train just one agent to solve the optimal execution problem while the other agent consistently plays the Nash equilibrium strategy outlined in Eq. (5.7). Clearly, in this last case we expect the agent to learn a strategy similar to the Nash strategy. As clearly in this case the learning agent has no incentive to deviate given the strategy of the other player, who is -by design- following the allocation suggested in Eq. (5.7).

Simultaneous learning around Nash. For the first set of complementary experiments, we consider agents that start exploring using the Nash equilibrium strategy, but leaving everything unchanged as in Section 5.3. To this end, we have modified the exploration policy in Eq. (5.22), such that for agent $i = 1, 2$ the exploration/exploitation rule is:

$$\begin{aligned} &\epsilon \in (0,1) \ , \ \zeta_i \sim \mathcal{U}(0,1) \\ &v_t^{(i)} = \begin{cases} \sim \mathcal{N}(\mu = v_t^N, \delta = v_t^*) & , \text{if } \zeta_i \leq \epsilon \\ \arg\max_{v' \in [0, q_0]} Q_M(g_t^i, v' | \theta_{\text{main}}) & , \text{else} \end{cases} \end{aligned} \tag{B.26}$$

Where v_t^N is the action for the Nash equilibrium strategy at time step t. We experiment over 10 testing runs of 2500 iterations, for all the values of σ. We report the results in Fig. B.13. It can be noticed how the agents recover strategies similar to those observed in Fig. 5.8 in Section 5.4, hence strategies that -on average- fluctuate over the TWAP strategy and away from the Nash equilibrium strategy that was the initial exploration strategy.

Single learning in a multi-player setup. We consider the case of an agent who is playing the Nash equilibrium solution, while the other is learning according to the rule reported in Eq. (5.22). We find that the latter agent is able to learn a trading strategy that overlaps with the true Nash equilibrium strategy, as reported in Fig. B.14. In this case we expect the agent to find this strategy as the Nash equilibrium strategy is the optimal -best response- strategy in this case.

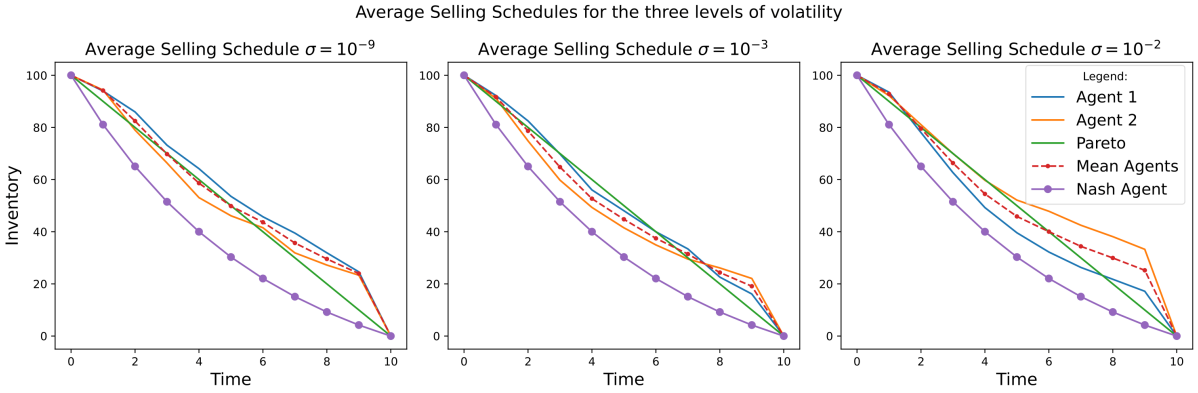


Figure B.13: Average optimal execution strategy over the 10 testing runs of 2500 iterations, for all the values of σ considered using exploration policy centred around Nash actions.

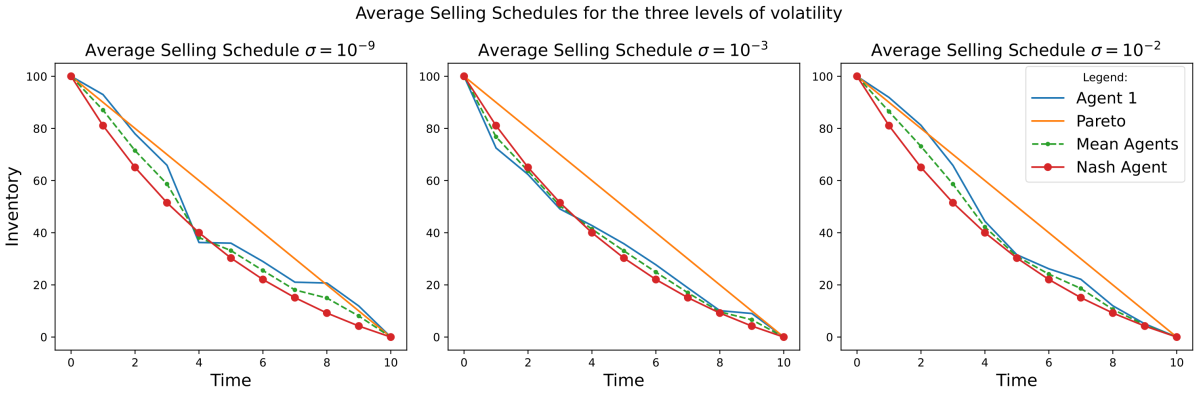


Figure B.14: Average optimal execution strategy over a run of 2500 iterations, for all the values of σ considered. Agent 2 consistently employs the Nash strategy while Agent 1 is being trained.

B.2 Reward and coin toss experiments

In this appendix we briefly analyse the behaviour of the algorithm outlined in Section 5.3 in the case when the reward in Eq. (5.23) is calculated as suggested in Schied and Zhang [2019]. Therefore, leaving all the other features as in Algorithm 7, the reward for each agent depends on the order of precedence of trading calculated according to a coin toss at each time step, the reward for each agent is thus calculated as:

$$\begin{aligned} r_{t,i} &= S_{t-1} v_{t,i} - \alpha v_{t,i}^2 \\ r_{t,i^-} &= \left(S_{t-1} + \kappa v_{t,i}\right) v_{t,i^-} - \alpha v_{t,i^-}^2 \end{aligned} \tag{B.27}$$

Therefore, in a given time-step, the agent trading as second pays the price impact generated by the first one. This ensures the symmetry and guarantees that, as the game unfolds, no

advantage in terms of trading timing is present for either of the two agents. The two agents are still trained as described above. Therefore the independent agents, have to solve the optimal execution problem in a market impact game but one pays the impact of the other at a time-step level rather than over the episode, according to a coin toss.

We experiment over 10 testing runs of 2500 iterations, for all the values of σ. We report the results in Fig. B.15. It can be noticed how the agents recover strategies similar to those observed in Fig. 5.8 in Section 5.4, hence on average learnt policies fluctuate over the TWAP strategy and away from the Nash equilibrium strategy, confirming the fact that the order of trading precedence for the agents over the large number of interactions needed to train, does not affect much the strategy learnt by the agents, therefore hinting at a persistence of supra-competitive equilibria even when the reward is slightly modified as in Eq. (B.27).

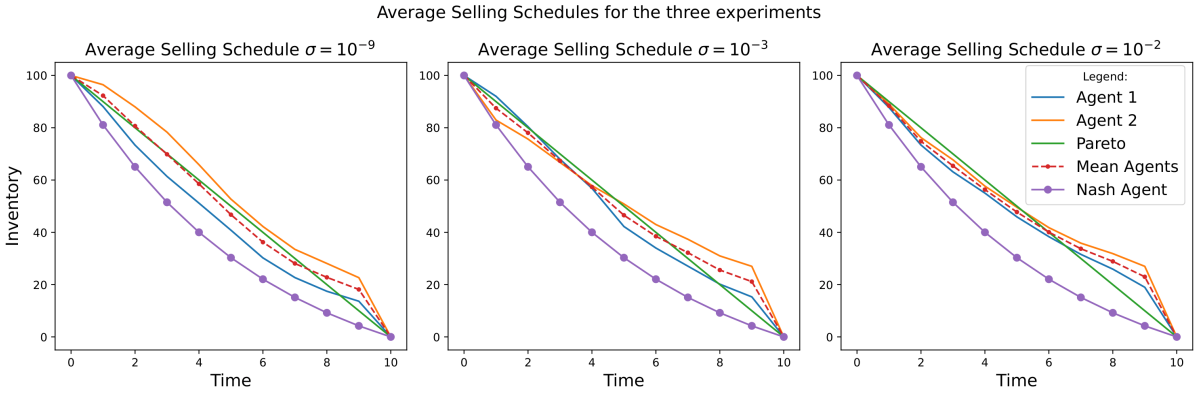


Figure B.15: Average optimal execution strategy over the 10 testing runs of 2500 iterations, for all the values of σ considered using a coin toss dependent reward.

Chapter 6

Conclusions

> "I am glad you are here with me. Here at the end of all things, Sam."
>
> – J.R.R. Tolkien, *The Return of the King* (1955)

In this thesis, we have investigated how reinforcement learning can be employed to address optimal execution and algorithmic trading problems, where market impact, liquidity, and informational structures evolve dynamically and are only partially observable. Across three complementary studies, we have shown how deep RL algorithms, in particular Double Deep Q-Learning (DDQL) and Deep Deterministic Policy Gradient (DDPG) methods, can learn optimal trading strategies under a variety of market conditions, ranging from single-agent execution and trading to multi-agent strategic interaction.

In Chapter 3, we showed how DDQL can effectively learn optimal liquidation strategies when liquidity is time-varying within the Almgren-Chriss linear market impact framework, where the impact is decoupled into both temporary and permanent components. These findings suggest that RL provides a robust and model-free tool when liquidity is inherently latent and evolves over time. Our experiments further showed how incorporating mid-price information, mitigates the sensitivity of the learned strategies to volatility and training length, highlighting the adaptability and the parsimonious nature of RL-based execution frameworks.

In Chapter 4, we have considered a trading problem, we have thus extended the scope of RL applications to trading environments governed by latent dynamics where the agent has to trade according to a signal. We proposed three DDPG-based architectures: *hid-DDPG*, *prob-DDPG*, and *reg-DDPG*. These architectures are built to handle informational content of a signal

following a Ornstein-Uhlenbeck process with regime-switching mean reversion, volatility, and speed parameters. Through extensive numerical tests and an empirical application to equity pair trading, we found how supplying structured and interpretable information about hidden regimes can dramatically improve the performance of the agent in terms of trading returns. In particular, the *prob-DDPG* architecture, featuring posterior regime probabilities estimated through a previously trained GRU network, consistently achieved the highest returns. These results underline that the quality and structure of the information fed to an RL agent are crucial for learning effective inter-temporal trading policies.

In Chapter 5, we explored how multiple agents would interact in a market impact game where the shared underlying market dynamics evolve according to the Almgren-Chriss model. Therefore, using a two-player DDQL setup we showed that agents trained independently do not converge to the Nash equilibrium. Rather, they often learn independently *supra-competitive* strategies that align closely with the Pareto-optimal liquidation costs as outcomes. These behaviours are consistent across varying levels of volatility and even when model parameters differ between training and testing phases. Suggesting robustness of tacit coordination among learning agents. From a broader perspective, such findings highlight both the potential and the risks associated with autonomous financial agents: on the one hand, the agents that are optimally executing meta-orders seem to coordinate on supra-competitive cost outcomes, but on the other their behaviour may slow down the process of correct incorporation of information into prices, therefore adversely affecting overall market the welfare.

Overall, this thesis demonstrates that deep reinforcement learning constitutes a powerful and flexible framework for tackling modern challenges in optimal execution and market microstructure. RL agents can infer optimal strategies in time-varying, latent, and stochastic environments; exploit structural regularities in hidden dynamics; and exhibit emergent collective behaviours when interacting. Future research should investigate how these learning mechanisms evolve in multi-agent and multi-asset settings, under non-linear and transient impact models, and in environments where liquidity and volatility co-evolve.

On another note, the analysis presented in these contributions should be regarded as a first step toward more general financial and game-theoretical environments. In two of the chapters, the Almgren and Chriss model provides the analytical backbone, and the results are therefore closely tied to Markovian price dynamics, implicit in that framework. Extending the setting to non-linear impact specifications would enlarge the set of admissible optimal solutions and

may allow for transaction-triggered price manipulation or other manipulative behaviours, as is well known in the literature (see, for instance, Alfonsi and Schied [2010], Gatheral [2010]). Such extensions would offer a richer and more realistic representation of market microstructure, although at the cost of losing the clear analytical benchmarks available under the linear-impact setting, as Nash equilibria and other theoretical tools would have to be derived.

Moreover, when transient impact dynamics are considered, as in Bouchaud et al. [2003], Obizhaeva and Wang [2013], the Markovian structure of the model no longer holds, making the problem substantially more challenging in a reinforcement learning context, since most RL methods are naturally designed for Markov decision problems. This suggests a promising methodological direction, requiring architectures capable of handling path dependence and richer state representations.

Similarly, the informational structure of impact games deserves further investigation. In the present framework, information is represented through the features supplied to the neural networks modelling the learners. While some benchmark solutions, such as the Pareto-optimal TWAP strategy, do not directly depend on the inventories or actions of the competing agent, alternative information sets may induce qualitatively different learned behaviours. Understanding how heterogeneity in preferences, asymmetric information, and richer impact dynamics jointly affect equilibrium selection and learning outcomes constitutes, in my view, a particularly promising direction for future research.

Beyond methodological developments, understanding how RL-driven agents influence market stability and price formation, along with a deeper understanding on what role the different type of information provided to such algorithms plays remains an open and crucial direction, with implications not only to quantitative finance but to regulation and market design as well.

List of Figures

List of Tables

List of Algorithms

Bibliography

Ibrahim Abada and Xavier Lambin. Artificial intelligence: Can seemingly collusive outcomes be avoided? *Management Science*, 69(9):5042–5065, 2023.

Ibrahim Abada, Xavier Lambin, and Nikolay Tchakarov. Collusion by mistake: Does algorithmic sophistication drive supra-competitive profits? *European Journal of Operational Research*, 2024.

Jacob Abernethy and Satyen Kale. Adaptive market making via online learning. *Advances in Neural Information Processing Systems*, 26, 2013.

A. Alfonsi and A. Schied. Optimal trade execution and absence of price manipulations in limit order book models. *SIAM Journal on Financial Mathematics*, 1(1):490–522, 2010.

Robert Almgren and Neill Chriss. Optimal execution of portfolio transactions. *Journal of Risk*, 3:5–39, 2000.

Robert Almgren, Chee Thum, Emmanuel Hauptmann, and Hong Li. Direct estimation of equity market impact. *Risk*, 18(7):58–62, 2005.

Alessio Azzutti, Wolf-Georg Ringe, and H Siegfried Stiehl. Machine learning, market manipulation, and collusion on capital markets: Why the" black box" matters. *University of Pennsylvania Journal of International Law*, 43(1):79, 2021.

Wenhang Bao and Xiao-yang Liu. Multi-agent deep reinforcement learning for liquidation strategy analysis. *arXiv preprint arXiv:1906.11046*, 2019.

Weston Barger and Matthew Lorig. Optimal liquidation under stochastic price impact. *International Journal of Theoretical and Applied Finance*, 22(02):1850059, 2019.

Dimitris Bertsimas and Andrew W Lo. Optimal control of execution costs. *Journal of financial markets*, 1(1):1–50, 1998.

Julian Blank and Kalyanmoy Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

Jean-Philippe Bouchaud, Yuval Gefen, Marc Potters, and Matthieu Wyart. Fluctuations and response in financial markets: the subtle nature of 'random' price changes. *Quantitative finance*, 4(2):176, 2003.

Jean-Philippe Bouchaud, J Doyne Farmer, and Fabrizio Lillo. How markets slowly digest changes in supply and demand. In *Handbook of financial markets: dynamics and evolution*, pages 57–160. Elsevier, 2009.

Antonio Briola, Jeremy Turiel, Riccardo Marcaccioli, and Tomaso Aste. Deep reinforcement learning for active high frequency trading. *arXiv preprint arXiv:2101.07107*, 2021.

Markus K Brunnermeier and Lasse Heje Pedersen. Predatory trading. *The Journal of Finance*, 60(4):1825–1863, 2005.

Frédéric Bucci, Iacopo Mastromatteo, Zoltán Eisler, Fabrizio Lillo, J-P Bouchaud, and C-A Lehalle. Co-impact: Crowding effects in institutional trading activity. *Quantitative Finance*, 20(2):193–205, 2020.

Apostolos N. Burnetas and Michael N. Katehakis. Optimal adaptive policies for markov decision processes. *Mathematics of Operations Research*, 22(1):222–255, 1997.

Emilio Calvano, Giacomo Calzolari, Vincenzo Denicolo, and Sergio Pastorello. Artificial intelligence, algorithmic pricing, and collusion. *American Economic Review*, 110(10):3267–3297, 2020.

Francesco Campigli, Giacomo Bormetti, and Fabrizio Lillo. Measuring price impact and information content of trades in a time-varying setting. *arXiv e-prints*, page arXiv:2212.12687, 2022.

Bruce Ian Carlin, Miguel Sousa Lobo, and S Viswanathan. Episodic liquidity crises: Cooperative and predatory trading. *The Journal of Finance*, 62(5):2235–2274, 2007.

René Carmona and Joseph Yang. Predatory trading: a game on volatility and liquidity. *Preprint. URL: http://www. princeton. edu/rcarmona/download/fe/PredatoryTradingGameQF. pdf*, 2011.

Alvaro Cartea and Sebastian Jaimungal. Incorporating order-flow into optimal execution. *Mathematics and Financial Economics*, 10:339–364, 2016.

Álvaro Cartea, Sebastian Jaimungal, and Josè Penalva. *Algorithmic and high-frequency trading.* Cambridge University Press, 2015.

Alvaro Cartea, Patrick Chang, and Josè Penalva. Algorithmic collusion in electronic markets: The impact of tick size. *Available at SSRN 4105954*, 2022.

Álvaro Cartea, Sebastian Jaimungal, and Leandro Sánchez-Betancourt. Reinforcement learning for algorithmic trading. *Machine Learning and Data Sciences for Financial Markets: A Guide to Contemporary Practices. Cambridge University Press*, 2023.

Philippe Casgrain and Sebastian Jaimungal. Trading algorithms with learning in latent alpha models. *Mathematical Finance*, 29(3):735–772, 2019.

Philippe Casgrain, Brian Ning, and Sebastian Jaimungal. Deep q-learning for nash equilibria: Nash-dqn. *Applied Mathematical Finance*, 29(1):62–78, 2022.

Ernie Chan. *Algorithmic trading: winning strategies and their rationale.* John Wiley & Sons, 2013.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Rama Cont and Wei Xiong. Dynamics of market making algorithms in dealer markets: Learning and tacit collusion. *Mathematical Finance*, 2022.

Rama Cont, Xin Guo, and Renyuan Xu. Interbank lending with benchmark rates: Pareto optima for a class of singular control games. *Mathematical Finance*, 31(4):1357–1393, 2021.

Francesco Cordoni and Fabrizio Lillo. Instabilities in multi-asset and multi-agent market impact games. *Annals of Operations Research*, 336(1):505–539, 2024a.

Francesco Cordoni and Fabrizio Lillo. Transient impact from the nash equilibrium of a permanent market impact game. *Dynamic Games and Applications*, 14(2):333–361, 2024b.

Kevin Dabérius, Elvin Granat, and Patrik Karlsson. Deep execution-value and policy based reinforcement learning for trading and beating market benchmarks. *Available at SSRN 3374766*, 2019.

Matthew F Dixon, Igor Halperin, and Paul Bilokon. *Machine learning in finance*, volume 1170. Springer, 2020.

Winston Wei Dou, Itay Goldstein, and Yan Ji. Ai-powered trading, algorithmic collusion, and price efficiency. *Jacobs Levy Equity Management Center for Quantitative Financial Research Paper*, 2024.

Samuel Drapeau, Peng Luo, Alexander Schied, and Dewen Xiong. An fbsde approach to market impact games with stochastic parameters. *arXiv preprint arXiv:2001.00622*, 2019.

Jean-Pierre Fouque, Sebastian Jaimungal, and Yuri F Saporito. Optimal trading with signals and stochastic price impact. *SIAM Journal on Financial Mathematics*, 13(3):944–968, 2022.

Nicolae Gârleanu and Lasse Heje Pedersen. Dynamic trading with predictable returns and transaction costs. *The Journal of Finance*, 68(6):2309–2340, 2013.

Jim Gatheral. No-dynamic-arbitrage and market impact. *Quantitative finance*, 10(7):749–759, 2010.

Jim Gatheral, Alexander Schied, and Alla Slynko. Transient linear price impact and fredholm integral equations. *Mathematical Finance: An International Journal of Mathematics, Statistics and Financial Economics*, 22(3):445–474, 2012.

Massimiliano Gobbi, F Levi, Gianpiero Mastinu, and Giorgio Previati. On the analytical derivation of the pareto-optimal set with applications to structural design. *Structural and Multidisciplinary Optimization*, 51:645–657, 2015.

Gueant, Lehalle, and Fernandez-Tapia. Optimal portfolio liquidation with limit orders. *SIAM Journal on Financial Mathematics*, 3(1):740–764, 2012.

Olivier Guéant. *The Financial Mathematics of Market Liquidity: From optimal execution to market making*. CRC Press, 2016.

Olivier Guéant and Charles Albert Lehalle. General intensity shapes in optimal liquidation. *Mathematical Finance*, 25(3):457–495, 2015.

Ben Hambly, Renyuan Xu, and Huining Yang. Recent advances in reinforcement learning in finance. *Mathematical Finance*, 33(3):437–503, 2023.

James D. Hamilton. A new approach to the economic analysis of nonstationary time series and the business cycle. *Econometrica*, 57(2):357–384, 1989. doi: 10.2307/1912559.

Joseph E Harrington. Developing competition law for collusion by autonomous artificial agents. *Journal of Competition Law & Economics*, 14(3):331–363, 2018.

Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

Dieter Hendricks and Diane Wilcox. A reinforcement learning extension to the almgren-chriss framework for optimal trade execution. In *2014 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*, pages 457–464. IEEE, 2014.

Sebastian Jaimungal. Reinforcement learning and stochastic optimisation. *Finance and Stochastics*, 26(1):103–129, 2022.

Thibault Jaisson. Deep differentiable reinforcement learning and optimal trading. *Quantitative Finance*, 22(8):1429–1443, 2022.

Gyeeun Jeong and Ha Young Kim. Improving financial trading decisions using deep q-learning: Predicting the number of shares, action strategies, and transfer learning. *Expert Systems with Applications*, 117:125–138, 2019.

Michaël Karpe, Jin Fang, Zhongyao Ma, and Chen Wang. Multi-agent reinforcement learning in a realistic limit order book market simulation. In *Proceedings of the First ACM International Conference on AI in Finance*, pages 1–7, 2020.

Michael Kearns and Yuriy Nevmyvaka. Machine learning for market microstructure and high frequency trading. *High Frequency Trading: New Realities for Traders, Markets, and Regulators*, 2013.

Robert Kissell. *Algorithmic trading methods: Applications using advanced statistics, optimization, and machine learning techniques.* Academic Press, 2020.

Petter N Kolm and Gordon Ritter. Modern perspectives on reinforcement learning in finance. *Modern Perspectives on Reinforcement Learning in Finance*, 2019.

Petter N Kolm, Jeremy Turiel, and Nicholas Westray. Deep order flow imbalance: Extracting alpha at multiple horizons from the limit order book. *Mathematical Finance*, 33(4):1044–1081, 2023.

Pankaj Kumar. Deep reinforcement learning for market making. *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1892–1894, 2020.

Xavier Lambin. Less than meets the eye: simultaneous experiments as a source of algorithmic seeming collusion. *Available at SSRN 4498926*, 2024.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Fabrizio Lillo and Andrea Macrì. Deviations from the nash equilibrium and emergence of tacit collusion in a two-player optimal execution game with reinforcement learning. *Forthcoming: Annals of Operations Research*, 2026. URL `https://arxiv.org/abs/2408.11773`.

Fabrizio Lillo and Andrea Macrì. Deviations from the nash equilibrium in a two-player optimal execution game with reinforcement learning. *Annals of Operations Research*, pages 1–33, 2026.

Julian Lorenz and Robert Almgren. Mean–variance optimal adaptive execution. *Applied Mathematical Finance*, 18(5):395–422, 2011.

Guiyuan Ma, Chi Chung Siu, Song-Ping Zhu, and Robert J Elliott. Optimal portfolio execution problem with stochastic price impact. *Automatica*, 112:108739, 2020.

Andrea Macrì and Fabrizio Lillo. Reinforcement learning for optimal execution when liquidity is time-varying. *Applied Mathematical Finance*, 31(5):312–342, 2024.

Andrea Macrì, Sebastian Jaimungal, and Fabrizio Lillo. Deep reinforcement learning for optimal trading with partial information. *arXiv preprint arXiv:2511.00190*, 2025. URL `https://arxiv.org/abs/2511.00190`.

Luca Philippe Mertens, Alberto Ciacci, Fabrizio Lillo, and Giulia Livieri. Liquidity fluctuations and the latent dynamics of price impact. *Quantitative Finance*, 22(1):149–169, 2022.

Alessandro Micheli, Johannes Muhle-Karbe, and Eyal Neuman. Closed-loop nash competition for liquidity. *Mathematical Finance*, 33(4):1082–1118, 2023.

Adrian Millea. Deep reinforcement learning for trading—a critical survey. *Data*, 6(11), 2021. ISSN 2306-5729. doi: 10.3390/data6110119. URL `https://www.mdpi.com/2306-5729/6/11/119`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Eyal Neuman and Moritz Voß. Trading with the crowd. *Mathematical Finance*, 33(3):548–617, 2023.

Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd international conference on Machine learning*, pages 673–680, 2006.

Brian Ning, Franco Ho Ting Lin, and Sebastian Jaimungal. Double deep q-learning for optimal execution. *Applied Mathematical Finance*, 28(4):361–380, 2021.

Anna A Obizhaeva and Jiang Wang. Optimal trading strategy and supply/demand dynamics. *Journal of Financial markets*, 16(1):1–32, 2013.

Gianluca Palmari, Fabrizio Lillo, and Zsolt Eisler. Optimal execution in a time varying environment: well posedness and price manipulation. *in preparation*, 2024.

Alexander Schied and Tao Zhang. A state-constrained differential game arising in optimal portfolio liquidation. *Mathematical Finance*, 27(3):779–802, 2017.

Alexander Schied and Tao Zhang. A market impact game under transient price impact. *Mathematics of Operations Research*, 44(1):102–121, 2019.

Alexander Schied, Torsten Schöneborn, and Michael Tehranchi. Optimal basket liquidation for cara investors is deterministic. *Applied Mathematical Finance*, 17(6):471–489, 2010.

Alexander Schied, Elias Strehle, and Tao Zhang. High-frequency limit of nash equilibria in a market impact game with transient price impact. *SIAM Journal on Financial Mathematics*, 8(1):589–634, 2017.

Matthias Schnaubelt. Deep reinforcement learning for the optimal placement of cryptocurrency limit orders. *European Journal of Operational Research*, 296(3):993–1006, 2022.

Torsten Schöneborn and Alexander Schied. Liquidation in the face of adversity: stealth vs. sunshine trading. In *EFA 2008 Athens Meetings Paper*, 2009.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

Justin Sirignano and Rama Cont. Universal features of price formation in financial markets: perspectives from deep learning. In *Machine Learning and AI in Finance*, pages 5–15. Routledge, 2021.

Shuo Sun, Rundong Wang, and Bo An. Reinforcement learning for quantitative trading. *ACM Transactions on Intelligent Systems and Technology*, 14(3):1–29, 2023.

Richard S Sutton and Andrew G Barto. Reinforcement learning: an introduction mit press. *Cambridge, MA*, 22447, 1998.

Avraam Tsantekidis, Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Using deep learning for price prediction by exploiting stationary limit order book features. *Applied Soft Computing*, 93:106401, 2020.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, 30(1), 2016.

Ludo Waltman and Uzay Kaymak. Q-learning agents in a cournot oligopoly model. *Journal of Economic Dynamics and Control*, 32(10):3275–3293, 2008.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge United Kingdom, 1989.

Haoran Wei, Yuanbo Wang, Lidia Mangu, and Keith Decker. Model-based reinforcement learning for predictions and control for limit order books. *arXiv preprint arXiv:1910.03743*, 2019.

Wei Xiong and Rama Cont. Interactions of market making algorithms. *Association for Computing Machinery*, 2022.

Hongyang Yang, Xiao-Yang Liu, Shan Zhong, and Anwar Walid. Deep reinforcement learning for automated stock trading: An ensemble strategy. In *Proceedings of the first ACM international conference on AI in finance*, pages 1–8, 2020.

Pengqian Yu, Joon Sern Lee, Ilya Kulyatin, Zekun Shi, and Sakyasingha Dasgupta. Model-based deep reinforcement learning for dynamic portfolio optimization. *arXiv preprint arXiv:1901.08740*, 2019.

Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deep reinforcement learning for trading. *arXiv preprint arXiv:1911.10107*, 2019a.

Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deeplob: Deep convolutional neural networks for limit order books. *IEEE Transactions on Signal Processing*, 67(11):3001–3012, 2019b.

Jinan Zou, Qingying Zhao, Yang Jiao, Haiyao Cao, Yanxi Liu, Qingsen Yan, Ehsan Abbasnejad, Lingqiao Liu, and Javen Qinfeng Shi. Stock market prediction via deep learning techniques: A survey, 2023.

Acknowledgements