POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Exploiting FX Trading Patterns at Multiple Time-Scales with Hierarchical Reinforcement Learning

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Author: **Luca Zerman**

Student ID: 968946
Advisor: Prof. Marcello Restelli
Co-advisors: Pierre Liotet
Academic Year: 2021-22

Abstract

The Foreign Exchange market is the largest financial market in the world and is therefore an attraction for banks, institutions and individual traders due to its high liquidity and low bid-ask spreads. With the evolution of technology, automated trading is an ever-more suitable tool to trade this market. Nevertheless, the choice of the trading frequency remains an important issue, considering that the variability and complexity of market patterns do not allow the identification of a single optimal one. In this thesis, we model a realistic intraday trading environment to introduce a novel hierarchical reinforcement learning algorithm, called Hierarchical Persistent Fitted Q-Iteration. This algorithm allows several trading frequencies to be considered simultaneously, resulting in an agent that is able to learn, at any instant, which frequency is best, thus exploiting the signal at different time scales while maintaining a high degree of control over its actions. We test the algorithm initially on the Mountain Car environment to better understand how it works empirically and then, after some exploratory analyses, on the EUR/USD market data. Finally, we compare its performance to the one of an algorithm from literature, Persistent Fitted Q-Iteration. The results confirm the crucial importance of having an agent capable of constantly changing frequency, while also identify further areas for development and improvement.

Keywords: FX Trading, Hierarchical Reinforcement Learning, Fitted Q-Iteration

Abstract in lingua italiana

Il *Forex Exchange market* è il più grande mercato finanziario del mondo e per questo è un attrattore di banche, istituzioni e singoli trader per la sua elevata liquidità e i bassi spread bid-ask. Grazie all'evoluzione della tecnologia, il trading automatizzato sembra essere lo strumento più idoneo per operare in questo mercato nella maniera più efficace. Tuttavia rimane cruciale la scelta della frequenza di trading, considerando che la variabilità e complessità delle condizioni di mercato non consentono l'individuazione di un'unica frequenza ottimale. In questa tesi, costruiamo un ambiente realistico di trading infragiornaliero finalizzato all'introduzione di un nuovo algoritmo gerarchico di reinforcement learning, chiamato *Hierarchical Persistent Fitted Q-Iteration*. Questo algoritmo permette di prendere in considerazione diverse frequenze di trading contemporaneamente, dando vita ad un agente che è in grado di imparare, in ogni istante, quale sia la frequenza migliore, sfruttando quindi il segnale a diverse scale temporali pur mantenendo un elevato controllo sulle proprie azioni. Inizialmente testiamo l'algoritmo sull'ambiente Mountain Car e poi, dopo aver effettuato alcune analisi esplorative sui dati, sul mercato EUR/USD, confrontando le sue prestazioni con quelle di *Persistent Fitted Q-Iteration*, un algoritmo presente in letteratura. I nostri risultati confermano l'importanza cruciale di disporre di un agente in grado di cambiare costantemente la frequenza, individuando anche ulteriori aree di sviluppo e di miglioramento.

Parole chiave: FX Trading, Hierarchical Reinforcement Learning, Fitted Q-Iteration

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

1.0.1 Motivations and Goals 2

1.0.2 Contributions 3

1.0.3 Outline of the Thesis 3

2 Preliminaries **5**

2.1 General Framework 5

2.1.1 Markov Decision Process 5

2.1.2 Value Functions 8

2.1.3 Bellman Equations 9

2.1.4 Optimality 9

2.2 Dynamic Programming 13

2.2.1 Policy Evaluation 13

2.2.2 Policy Improvement 14

2.2.3 Policy Iteration 15

2.2.4 Value Iteration 15

2.3 Temporal-Difference learning 16

2.3.1 TD Prediction 16

2.3.2 TD Control 17

2.3.3 Q-learning 17

2.4 Approximation Methods 19

2.4.1 Fitted Q-Iteration 19

2.4.2 Persistent Fitted Q-Iteration 22

2.5 Regressors for Fitting Q . . . 24
2.5.1 Extremely Randomized Trees . . . 24
2.5.2 XGBoost . . . 25

3 Related Works 29
3.1 Reinforcement Learning in Finance . . . 29
3.1.1 Deep Reinforcement Learning . . . 29
3.1.2 Persistence in Reinforcement Learning . . . 30
3.2 Hierarchical Reinforcement Learning . . . 30
3.2.1 General Setting . . . 30
3.2.2 Approaches for Hierarchical Reinforcement Learning . . . 31
3.2.3 Hierarchical Reinforcement Learning in Finance . . . 35

4 Hierarchical Persistent Fitted Q-Iteration (HPFQI) 37
4.1 Motivation . . . 37
4.2 Algorithm . . . 38
4.2.1 Comparison with HQI . . . 42

5 HPFQI Evaluation on Mountain Car Environment 43
5.1 Problem Formulation . . . 43
5.1.1 Environment Description . . . 43
5.1.2 Motivation . . . 43
5.1.3 Mountain Car as an MDP . . . 44
5.1.4 Dataset . . . 45
5.2 Experimental Results . . . 46
5.2.1 Preliminary Considerations . . . 47
5.2.2 Results . . . 47
5.2.3 Analysis of the Results . . . 54
5.2.4 Randomization of the Initial Velocity . . . 60
5.2.5 Conclusions . . . 63

6 Problem Formulation FX 67
6.1 General Overview . . . 67
6.1.1 FX Market . . . 67
6.1.2 EUR/USD Market . . . 67
6.1.3 Reinforcement Learning in FX . . . 68
6.1.4 Market Data . . . 69
6.2 FX Market Trading as an MDP . . . 69

6.2.1 FX as an Episodic Task 69
6.2.2 Actions 70
6.2.3 State 70
6.2.4 Reward 71
6.2.5 Dataset 72
6.3 Data Exploration 73
6.3.1 Stationarity 73
6.3.2 Volatility 77

7 Experimental Results FX **79**
7.1 Model Selection 79
7.1.1 Regressor Parameters 79
7.1.2 Other Parameters 80
7.2 Results 81
7.2.1 PFQI 81
7.2.2 HPFQI 87
7.2.3 HPFQI vs PFQI 90

8 Conclusions and Future Developments **101**
8.1 Future Developments 102

Bibliography **105**

List of Figures **111**

List of Tables **117**

1 | Introduction

The Foreign Exchange Market (Forex or FX) is a decentralized market for the trading of national currencies that operates 24 hours a day and 5 days a week. It is made up of a network of banks, financial institutions and individual traders who buy and sell currencies from all around the world. Currency values are constantly changing due to various factors such as economic and political events: this creates opportunities for traders to profit by buying or selling currencies based on their predictions of how the currency values will change in the future. According to the *Triennial Central Bank Survey of FX and OTC derivatives markets in 2022* conducted by the Bank of International Statements in 2022, the FX market is the largest financial market in the world with a daily volume of \$7.5 trillion. In particular, the exchange rate between the Euro, the currency of the European Union, and the US dollar, the currency of the United States (EUR/USD), is the most traded currency pair, reaching a daily volume of \$1.8 trillion. These huge trading volumes ensure FX and especially EUR/USD traders several fundamental advantages. Firstly, high trading volume is generally an indication of both high liquidity of the security, which provides traders with more opportunities to enter and exit trades at the desired price levels, and low bid-ask spreads, which lower the costs associated to buying and selling currency pairs. Secondly, the renowned economic stability of both the Eurozone and the United States makes the EUR/USD market less risky for traders compared to other currency pairs where the economies or political systems may be more volatile or unstable.

Technology became increasingly important in FX trading in the 1990s with the rise of electronic trading platforms and the development of algorithmic trading strategies. The emergence of the internet and more sophisticated computer systems allowed traders to access real-time market data and execute trades faster and more efficiently than ever before. In addition, the availability of advanced analytics tools and machine learning algorithms has allowed traders to analyse large amounts of data and make more informed trading decisions. In particular, the trading process can be modeled as a Markov Decision Process where, at each time-step, the artificial trading agent first collects information from the market and then decides what orders to execute. Since the dynamics of the FX market are unknown, reinforcement learning becomes an excellent framework to design autonomous trading agents.

Reinforcement learning has been used in trading since the early 2000s, predominantly in algorithmic trading for financial markets. Some of the earliest uses of reinforcement learning in this field include studies about optimizing trading strategies for options, futures and portfolio management. Since then, reinforcement learning has been used in a variety of trading applications, including high frequency trading, arbitrage and asset allocation. However, reinforcement learning in FX trading is an emerging field that has gained popularity in recent years. The fact that reinforcement learning in FX trading is still in its early stages further stimulates us to embark on this path.

In FX, and in trading in general, the choice of operating frequency is crucial. Indeed, even if profitable trading opportunities can potentially be found at many different time frames, learning them is not equally difficult. Although most of the autonomous agents that interact with the FX market today often operate at a high frequency (e.g.: seconds), their behavior is usually hard-coded since, at such a time scale, the main trading opportunities exploit ephemeral arbitrages and a low-latency access to market data [23]. On the other hand, operating at a very low frequency (e.g.: days, months) is unfeasible without including all those exogenous inputs, such as economic data release and financial news, that may affect market prices. Thus, operating at a middle frequency (e.g.: minutes, hours) is the most suitable scenario for those, like us, who only work on market data.

1.0.1. Motivations and Goals

Although we have just ascertained which frequencies would be the best for us to trade on, we are still far from being certain about the choice of a perfect frequency. Actually, we believe that there is no perfect frequency for every situation; that is where our work fits in. Indeed, in our thesis we aim to apply Hierarchical Reinforcement Learning (HRL) techniques to FX trading for the first time, in order to allow the agent to act at different frequencies. HRL was conceived in the 1990s for learning complex behaviors in environments with a high degree of uncertainty and variability. HRL immediately showed great promise but has only recently been used in the trading world.

Our work begins with the implementation of a batch reinforcement learning algorithm called Persistent Fitted Q-Iteration (PFQI) [32], which allows the frequency of the signal to be tuned effectively. Afterwards, PFQI is revisited in what we call Adapted Persistent Fitted Q-Iteration (APFQI), that represents the fundamental element of the novel hierarchical algorithm: Hierarchical Persistent Fitted Q-Iteration (HPFQI). HPFQI allows to consider different frequencies simultaneously, training an agent who is able to understand, at each time-step, which frequency is the most profitable one. We first test HPFQI on Mountain Car, an environment commonly used as a benchmark in reinforcement learning that particularly fits our purpose. Subsequently, after having performed some quantitative exploration on our data, we evaluate the algorithm on EUR/USD market. From a financial point of view, we trade liquid currency pairs only and fix a relatively small trading size. Hence, we can assume the possibility of quickly going long or short, without any market impact and slippage issues. Our results confirm the critical importance of the possibility to constantly change frequency but also highlights some improvements that can become inspiration for future works.

1.0.2. Contributions

With respect to our natural benchmark [38], the main contributions to the thesis are three:

- the introduction of a new hierarchical reinforcement learning algorithm (HPFQI) that allows the agent to encapsulate both lower and higher-frequency information. This aspect is crucial because low frequency actions consent to intercept signal trends, whereas high frequency ones enables a higher control on agent's policy;
- the detailed quantitative inspection of the data used for training and testing our algorithm, with the aim of being more confident in the results obtained;
- the implementation of the XGBoost algorithm as the regression method used to approximate the action-value function, a measure of the quality of an action, at each HPFQI iteration. XGBoost reduces the computational time and increases the accurateness of the estimation.

1.0.3. Outline of the Thesis

This work is structured as follows:

- in Chapter 2 the general reinforcement learning framework is introduced, paying particular attention to the concepts of Markov Decision Process and value function. Subsequently, after having processed Dynamic Programming and Temporal-Difference Learning methodologies, we familiarize with Fitted Q-Iteration and Persistent Fitted Q-Iteration algorithms, concluding with a brief explanation of Extra-Trees and XGBoost regression methods;
- in Chapter 3 we provide an overview of inherent reinforcement learning works, fo-

cusing especially on deep reinforcement learning, hierarchical reinforcement learning and methods exploiting the concept of persistence;

- in Chapter 4, after justifying the way the algorithm was constructed, we propose, via the definition of APFQI, the Hierarchical Persistent Fitted Q-Iteration algorithm;
- in Chapter 5 the Mountain Car environment is introduced to compare HPFQI with PFQI in a simpler setting. After several analyses, we have drawn conclusions that entice us to continue with FX;
- in Chapter 6 a general overview of FX market is provided, analysing why a reinforcement learning algorithm such ours fits in well; thereafter, a modelling of the trading of EUR/USD as an MDP is proposed. Before testing our algorithm, a quantitative exploration analysis on various data is performed, verifying that the framework is sound and datasets satisfy necessary conditions for the application of reinforcement learning algorithms;
- in Chapter 7, after having performed a model selection through a validation step, different parameterizations of HPFQI are trained and tested on EUR/USD market data and compared with PFQI;
- in Chapter 8 conclusions drawn from experimental results and objectives achieved within this thesis are discussed. Finally, future studies and possible developments are illustrated.

2 | Preliminaries

Reinforcement learning is an automatic learning technique that focuses on solving problems where an agent interacts with an environment, learning to make decisions by performing actions and observing the resulting rewards. In this framework, an agent learns through trial and error by taking actions in an environment and observing the consequences. The goal of the agent is to maximize the total reward it receives over time by choosing actions that lead to the most favorable outcomes.

2.1. General Framework

2.1.1. Markov Decision Process

Going into rather more detail, the general reinforcement learning framework is made of an interaction between an *agent*, who makes decisions driven by an goal which we will discuss in more detail later on, and the *environment*, which indicates everything outside the agent. For what concerns the discrete time setting, at each time-step $t = 0, 1, 2 \dots$ the agent receives a particular representation of the environment's *state* $S_t \in \mathcal{S}$, where $\mathcal{S}$ identifies the set of possible states, and based on that it takes an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ represents the set of available actions in state S_t. In the further time-step $t+1$, the agent receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ as a consequence of its action, and thereafter he will be in a new state S_{t+1}. Notably, the reward R_{t+1} depends on S_t and A_t, indeed we can define the reward function as $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Given that, one question could arise: how does the agent decide to set out between states? Before answering this, we must define the *state-transition probability* function $P(s'|s, a)$. $P(s'|s, a)$ describes the probability of arriving in state s' taking action a from the starting state s, hence it characterizes the environment dynamics. Given that, the goal of the agent roughly is to maximize the total reward it receives over the long run. A way to achieve so is to act through a *policy* π_t, which is a mapping from states to probabilities of choosing each possible action. In particular, $\pi_t(a|s)$ is the probability that $A_t = a$ given that $S_t = s$, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.

But how can we define formally the goal the agent tries to achieve? In general, our aim is to maximize the *expected return*, where the *return* G_t is defined as a specific function of the reward sequence. Intuitively, the first idea that we can come up with could be the sum of the future rewards:

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T \, , \tag{2.1}$$

where T stays for a possible final time step. A definition thus posed is consistent only if we consider a situation where there is a natural notion of final step: in these cases, T is called *terminal state* and the subsequence $\{t, t+1, ..., T\}$ is called *episode*.

On the other hand, in many cases the agent/environment interaction does not break naturally into identifiable episodes, but goes on without limit; in these cases we talk about *continuing tasks*. Given that $T = +\infty$, the return formulation (2.1) could be problematic because the return itself becomes infinite.

For this reason we add the concept of *discount* and thereby of *discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \, , \tag{2.2}$$

where γ is the *discount rate* which has to satisfy $0 \leq \gamma \leq 1$.

The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately, which makes sense considering that a reward in the future is less assured than one now. If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence $\{R_k\}$ is bounded. In particular, if $\gamma = 0$ the agent is concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose the action A_t so as to maximize only R_{t+1}; for this reason, such an agent is called *myopic*. As γ approaches 1, instead, the objective takes future rewards into account more strongly: the agent becomes more *farsighted*.

In Section 2.1 we only vaguely defined the concept of *state*. In the reinforcement learning Framework, the state represents the entirety of the information available to the agent in order to make decisions. But what information do we need to know to act in the best way possible in the environment?

In general, if we assume without loss of generality to have a finite number of states and reward values, we can define the dynamics of the environment response at time $t+1$ to the action taken by the agent at time t as the complete probability distribution:

$$\mathbb{P}(R_{t+1} = r, S_{t+1} = s'|S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t) \tag{2.3}$$

$\forall r, s'$ and for all possible values of $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$. In (2.3), we state that the environment dynamics depend on the whole history of events. Fortunately, almost always we can instead make a stricter assumption, known as *Markov property*. If the state signal satisfies this property, then the environment response at $t+1$ depends only on the state and action representations at time t. Indeed, with the Markov property it is assumed that the future is not influenced by past history if the present is known. Therefore, the probability distribution (2.3) can be redefined as:

$$p(s', r|s, a) \coloneqq \mathbb{P}(R_{t+1} = r, S_{t+1} = s'|S_t = s, A_t = a) \tag{2.4}$$

$\forall r, s', s, a$. In other words, the environment satisfies the Markov property if and only if (2.4) is equal to (2.3) $\forall r, s'$ and for all possible values of $S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t$.

A reinforcement learning task that satisfies the Markov property is called *Markov Decision Process* (MDP) or *Finite Markov Decision Process* if $\mathcal{S}$ and $\mathcal{A}$ are finite. As aforementioned, the probability of each possible pair of next state and reward (s', r), given any pair of state and action (s, a), is denoted by the transition probability defined in (2.4). The importance of this function is that it completely characterizes the environment's dynamics. Indeed, it allows to compute all the other environment components, such as the so called *state-transition probabilities*

$$P(s'|s, a) \coloneqq \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a)\,, \tag{2.5}$$

which has been easily obtained by (2.4) integrating out (in a finite way) the reward. Furthermore, we can compute also the expected rewards for state-action pairs,

$$R(s, a) \coloneqq \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a), \tag{2.6}$$

and the expected rewards for state action next-state triples,

$$R(s, a, s') \coloneqq \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_{r \in \mathcal{R}} r p(s', r|s, a)}{p(s'|s, a)}\,. \tag{2.7}$$

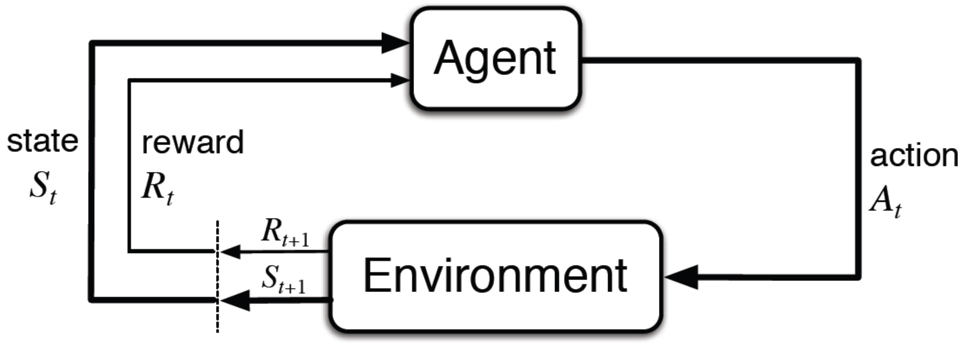


Figure 2.1: The agent-environment interaction in a Markov Decision Process (Source: [44]).

Summing up, a discrete-time MDP is defined as $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the state-transition probability function, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function and γ is the discount factor.

2.1.2. Value Functions

In this section, we define two critical functions for reinforcement learning tasks: the *state-value function* and the *action-value function*. The task of these two functions is to define, in each situation in which the agent may find itself at a certain time, how good that situation is in terms of the expected return. In particular, the state-value function assigns a value for each state, whilst the action-value function assigns a value for each pair state/action taken. Intuitively, the state-value function and the action-value function will depend on the policy chosen by the agent, since the expected total discounted reward will for sure depend on the way the agent behaves. Formally, the state-value function for policy π is defined as:

$$V^{\pi}(s) := \mathbb{E}_{\pi}[G_t | S_t = s] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right] \quad \forall s \in \mathcal{S}, \tag{2.8}$$

where the expression $\mathbb{E}_{\pi}[\cdot]$ stands for the expected value given that the agents follows the policy π. Similarly, the action-value function for policy π can be formalized as

$$Q^{\pi}(s, a) := \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \quad \forall s \in \mathcal{S}\ \forall a \in \mathcal{A}\,. \tag{2.9}$$

2.1.3. Bellman Equations

At this point, firstly we introduce the recursive definitions of the value functions, to later address the concept of optimality in policies and value functions. In order to do so, we begin expressing the return at time t, G_t, in a recursive way. Starting from definition (2.2), it is easy to see that

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{2.10}$$

In light of this result, we are ready to define the state-value function in such a way that it depends on the functions introduced in Section 2.1.1 and on the state-value function itself:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a) \big[r + \gamma \mathbb{E}_\pi \left[G_{t+1}|S_{t+1} = s'\right]\big] \\ &= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a) \big[r + \gamma V^\pi(s')\big] \quad \forall s \in \mathcal{S}\,. \end{aligned} \tag{2.11}$$

This equation is called *Bellman equation for V^π* and it roughly states that the state-value function evaluated in the starting state must be equal to the discounted state-value function evaluated in the expected next state, added to the reward expected along the way. Interestingly, the state-value function V^π represents the unique solution to its Bellman equation [44, Section 3.7].

2.1.4. Optimality

In view of the value functions defined in Section 2.1.2 and the Bellman equation introduced just above, we are ready to understand why V^π and Q^π are relevant in solving a reinforcement learning task. As previously said, the final aim of reinforcement learning is to find a policy that achieves the highest return possible over the long run; if there exists a policy which is better than or at least equal to all the other possible policies, *i.e.* its expected return is greater than or equal to that of all the other policies for all the states, we talk about the *optimal policy*. Given that, we display some relevant value functions' properties in order to lay the groundwork for discussing it:

1. for Finite Markov Decision Processes, value functions define a partial ordering over

policies. Indeed, it holds that a policy π is better than another policy π' if and only if the state-value function for policy π evaluated in state s is greater than or equal to the state-value function for policy π' evaluated in the same state s for all $s \in \mathcal{S}$. More formally,

$$\pi \succeq \pi' \iff V^{\pi}(s) \geq V^{\pi'}(s) \quad \forall s \in \mathcal{S}; \tag{2.12}$$

2. there always exists at least one optimal policy, *i.e.* there is always at least one policy better than or equal to all other policies [44, Section 3.8]. Formally,

$$\exists \pi : \pi \succeq \pi' \quad \forall \pi' \in \Pi\,, \tag{2.13}$$

where Π indicates the space of all the possible policies. All the optimal policies (there may be more than one) are denoted by π^*;

3. all the optimal policies share the same state-value function, called the *optimal state-value function* denoted by V^*, which is defined as:

$$V^*(s) := \max_{\pi} V^{\pi}(s) \quad \forall s \in \mathcal{S}; \tag{2.14}$$

4. all the optimal policies share also the same action-value function, called the *optimal action-value function* denoted by Q^*, which is defined as:

$$Q^*(s,a) := \max_{\pi} Q^{\pi}(s,a) \quad \forall s \in \mathcal{S}\ \forall a \in \mathcal{A}\,. \tag{2.15}$$

More precisely, $Q^*(s,a)$ stands for the expected return for taking the action a when in state s and thereafter following the optimal policy.

Since V^* is a state-value function like any V^{π}, it must satisfy the Bellman equation (2.11) as well:

$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(s,a) \\
&= \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \mathbb{E}_{\pi}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a)\,[r + \gamma V^*(s')]\,,
\end{aligned} \tag{2.16}$$

where the first equation derives from the definition (2.15), the second one from the definition (2.9), the third one from (2.10), the fourth one from (2.14) and (2.8). The last two equations are two forms of the so-called *Bellman optimality equation* for V^*, which expresses the fact that the value of the state under an optimal policy must be equal to the expected return for the best action from that state.

Similarly, we can define the Bellman optimality Equation for Q^*:

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}(s')} Q^*(S_{t+1}, a') | S_t = s, A_t = a\right] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \left[r + \gamma \max_{a' \in \mathcal{A}(s')} Q^*(s', a')\right]. \end{aligned} \tag{2.17}$$

If the dynamics of the environment are known, *i.e.* if the state-transition probability function (2.5) is known, equations (2.16) and (2.17) are solvable (actually they are two systems composed by one equation for each state) and therefore V^* and Q^* can be obtained. Once found V^* and Q^*, the optimal policy is just a stone's throw away: if we have V^*, any policy that is *greedy* with respect to V^* is an optimal policy, where greedy means that the behaviour is determined only by short-term considerations. The power of the optimal state-value function is precisely this: if the agent chooses to act maximizing V^* in a one-step sight, it is actually following the optimal policy in the long term as well, hence V^* allows to see locally and immediately the optimal expected long term return. With Q^* the situation is even simpler: the agent can choose, for any state s, any action that maximizes $Q^*(s,a)$ without even performing a one-step-ahead search. This result hides an even bigger advantage of the optimal action-value function with respect to the optimal state-value one: although the optimal action-value function is a bit harder to represent (it depends on state-action pairs instead of just on states), it consents to choose the optimal action without having to know anything about the environment dynamics. This aspect could seem irrelevant but in real-case scenarios it is often difficult or impossible to represent the environment dynamics (just think about continuous state space $\mathcal{S}$), therefore this benefit proves to be definitely helpful.

Nontheless, unfortunately, these results are rarely useful directly in practice. Indeed, this solution relies on at least three assumptions that are rarely true:

1. the dynamics of the environment are accurately known;
2. computational resources are sufficient to complete the calculation;
3. the states have the Markov property.

For this reason, one is generally not able to implement the exact solution of the kind of tasks in which he is interested because various combinations of these assumptions are usually violated. Therefore, the goal now becomes finding approximate solutions of these equations as close as possible to the exact ones.

2.2. Dynamic Programming

The term *Dynamic Programming* (DP) refers to a collection of algorithms that can be used to find optimal policies whenever the agent has a complete model of the environment's dynamics. Unfortunately, the utility of Dynamic Programming algorithms is limited: firstly, the assumption of a complete knowledge of a perfect MDP model is very rare. Secondly, the computational expense typically is really high, since usually the state space S and action space A are quite big.

Nevertheless, theoretical results that are inapplicable in reality are often fundamental as a benchmark for the methods and algorithms that will be actually used. Dynamic Programming, indeed, helps us to understand how to build more advanced, reliable and usable reinforcement learning algorithms in real-world scenarios. The main idea of DP is to use the value functions introduced in Section 2.1.2 to organize and structure the search for good policies. As can be expected, DP algorithms are obtained turning Bellman equations (Section 2.1.3) into update rules for improving approximations of the desired value functions.

In particular, the procedure of determining the optimal policy is based on the interaction between two processes: *policy evaluation* and *policy improvement*. The former refers to the computation of the value function for a given policy; the latter, instead, refers to the computation of an improved policy given the value function for that policy. These two approaches are usually joint in an iterative fashion to create what is called *policy iteration*.

2.2.1. Policy Evaluation

Policy evaluation represents an alternative method to solve the Bellman equation (2.11) (hence to extract V^π knowing $\pi(a|s)$ $\forall a \in \mathcal{A}(s)$ and $\forall s \in \mathcal{S}$) in an iterative fashion.

The algorithm works as follows. Consider a sequence of approximate value functions $V^0, V^1, V^2, \dots$ where V^0 is chosen arbitrarily. Each approximation successive to V^0 is obtained by using the Bellman equation for V^π (2.11) as an update rule:

$$\begin{aligned} V^{k+1}(s) &:= \mathbb{E}_\pi[R_{t+1} + \gamma V^k(S_{t+1}) | S_t = s] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \big[r + \gamma V^k(s') \big] \quad \forall s \in \mathcal{S}\,. \end{aligned} \tag{2.18}$$

It can be shown that the sequence V^k converges to V^π as $k \to +\infty$ [44, Section 4.1].

2.2.2. Policy Improvement

Now that we have found V^π and thus we have understood how good the policy π is, we could hypothetically think about changing the action taken in a certain state $a \neq \pi(s)$ and after that step following the policy π (*i.e.* considering exactly $Q^\pi(s,a)$), in order to see if a change of action could lead to greater expected return so greater V value.

But how can we be sure that such a change would improve our performances? Let us first state the *policy improvement theorem.*

Theorem 2.1 (From Section 4.2 in [46]). *Let π and π' be any pair of deterministic policies such that*

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \quad \forall s \in \mathcal{S}\,. \tag{2.19}$$

Then,

$$V^{\pi'}(s) \geq V^\pi(s)\,, \tag{2.20}$$

i.e. the policy π' is at least good as π.

Moreover, if the inequality of the hypothesis (2.19) *is strict at any state, then, at that state, also the inequality* (2.20) *is strict.*

Given these results, we can consider of performing changes at all states, selecting at each state the action that appears the best according to $Q^\pi(s,a)$, *i.e.* considering the new greedy policy π' given by

$$\begin{aligned} \pi'(s) &\coloneqq \operatorname*{argmax}_{a\in\mathcal{A}(s)} Q^\pi(s,a) \\ &= \operatorname*{argmax}_{a\in\mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma V^\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \operatorname*{argmax}_{a\in\mathcal{A}(s)} \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s,a)\big[r + \gamma V^\pi(s')\big] \end{aligned} \tag{2.21}$$

By construction, π' satisfies the hypothesis of Theorem 2.1, hence it is at least good as π; the process of generating a new policy which improves the original one in the specific way described above is called *policy improvement.*

If it happens that the new greedy policy π' is exactly as good as the previous policy π, then $V^\pi = V^{\pi'}$ and from (2.21) it follows that

$$
\begin{aligned}
V^{\pi'}(s) &= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma V^{\pi'}(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \big[r + \gamma V^{\pi'}(s') \big] \quad \forall s \in \mathcal{S} \, .
\end{aligned}
\tag{2.22}
$$

But this equation is the same as the Bellman optimality equation (2.16), therefore $V^{\pi} \equiv V^{\pi'} \equiv V^*$ hence both π and π' must be optimal policies.

2.2.3. Policy Iteration

The *policy iteration* approach merges the two former algorithms in the following way. Starting from an arbitrary policy π_0, we perform a policy evaluation step in order to find V^{π_0}. Given that, we can now find a new policy π_1 which is at least good as π_0 thanks to a policy improvement step. Consequently, it is possible to perform again a policy improvement step on the new policy π_1 and thereafter iterate the whole reasoning until two policies with the same state-value function are found, which means that the optimal policy is obtained. This process can be summarized in this way:

$$
\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2 \rightarrow \cdots \rightarrow \pi_{n-1} \rightarrow V^{\pi_{n-1}} \rightarrow \pi_n \rightarrow V^{\pi_n} \equiv V^{\pi_{n-1}} \equiv V^* \, ,
$$

hence $\pi_{n-1} \equiv \pi_n \equiv \pi_*$. Moreover, considering that a finite Markov Decision Process has only a finite number of policies, this process must converge to an optimal policy in a finite number of iterations.

2.2.4. Value Iteration

One issue of policy iteration methodology relies on the policy evaluation part. Indeed, it can take several steps before convergence, hence leading to a high computational cost. Fortunately, there are several methods which simplify the policy evaluation part but leaving unchanged the policy iteration convergence property.

One example is *value iteration*, where the policy evaluation is stopped after just one sweep of this shape:

$$
\begin{aligned}
V^{k+1}(s) &= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma V^{k}(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) \big[r + \gamma V^{k}(s') \big] \quad \forall s \in \mathcal{S} \, .
\end{aligned}
\tag{2.23}
$$

Value iteration is obtained simply by turning the Bellman optimality equation (2.16) into an update rule. Like policy evaluation, this approach formally requires an infinite number of iterations to converge exactly to V^* but usually the process is stopped once the value function changes by only a small amount in a sweep. Furthermore, in order to achieve faster convergence, multiple policy evaluation sweeps are often interposed between each policy improvement step.

2.3. Temporal-Difference learning

Temporal difference (TD) learning refers to a fundamental class of model-free reinforcement learning methods. The crucial difference from Dynamic Programming relies on the fact that TD algorithms can learn optimal policy directly from raw experience without having to know the model of the environment, of its reward and next-state probability distributions. On the other hand, TD methods work as DP ones for what is called bootstrap technique, since value functions estimates are updated starting from other learned estimates without waiting the end of the episodes.

In this section, we firstly present how prediction (Section 2.3.1) and control (Section 2.3.2) are intended in TD framework, also tackling the crucial themes of exploration/exploitation dilemma and difference between on-policy and off-policy methods. Finally, the Q-learning algorithm is introduced (Section 2.3.3).

2.3.1. TD Prediction

The updating rule of a generic TD method can be summarized in this way:

$$\mathrm{VF}^{new} \leftarrow \mathrm{VF}^{old} + \alpha\Big[target - \mathrm{VF}^{old}\Big], \tag{2.24}$$

where VF represents a value function (either state-value or action-value), *target* a better estimate of the chosen value function, VF^{old} the current estimate of the value function, α the learning parameter and VF^{new} the actual value function update. In particular, the term in the square brackets is the so-called *TD error*, which expresses the difference of performance between old and new models. In view of that, the parameter α depicts the confidence placed in the target with respect to the previous model.

The simplest TD method, *TD(0)*, makes the following update:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]. \tag{2.25}$$

This approach is also called *one-step TD* because the target is $R_{t+1} + \gamma V(S_{t+1})$, *i.e.* it waits only one time step before updating the value function estimates.

2.3.2. TD Control

Similarly to what we did with DP approach, we aim to exploit the TD prediction results for the control problem. However, since we are dealing with model-free methods, we stumble into the so-called *exploration/exploitation dilemma*: to maximize the expected return, the agent must prefer actions that it has tried in the past and found to be effective in producing higher reward with respect to others. But, on the other hand, to discover such actions it has to try some new policies different from the first one. Therefore, the agent has to *exploit* what it has already learnt but it also has to continue *exploring* in order to possibly select better actions in the future. If the agent is totally unbalanced on the exploitation side, it remains stuck in the first policy obtained not willing to try something new; if the exploration side prevails, instead, it continues exploring without taking care of previous experience, hence it is not really learning. For this reason, it is always necessary to find a compromise between these two aspects.

This dilemma is approached differently depending on whether one is dealing with *on-policy* or *off-policy* methods: the former ones attempt to evaluate or improve the policy that is used to make decisions, whereas the latter ones consider one policy to generate the data but they evaluate and improve another one. More specifically, on-policy methods learn action values not for the optimal policy but for a near optimal-policy that still explores. Off-policy methods are based on the use of the *target policy*, the one that is learned about, and the *behavior policy*, more exploratory and used to generate behavior.

2.3.3. Q-learning

In this section we present an off-policy TD control algorithm that has been a starting point for our work: *Q-learning*. Its update rule is defined as:

$$Q_{i+1}(S_t, A_t) \leftarrow Q_i(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}(s)} Q_i(S_{t+1}, a) - Q_i(S_t, A_t) \right], \tag{2.26}$$

where i indicates the updating step. As we can see from this formula, independently from the policy chosen for selecting the new action, the update of the action-value function Q is greedy ($R_{t+1} + \max_a Q(S_{t+1}, a)$) and it does not depend on the other policy. In this way, Q directly approximates Q^* independently of the policy being followed; this dramatically simplifies the analyses of the algorithm and enables convergence proofs.

Algorithm 2.1 Q-learning for learning $\pi \approx \pi_*$

```
1:  **Algorithm parameters**: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
2:  **Initialization**: $Q(s, a)\ \forall s \in \mathcal{S}\ \forall a \in \mathcal{A}(s)$ arbitrarily, except that $Q(terminal, \cdot) = 0$

3:  **for** episode in episodes **do**
4:      Initialize $S$
5:      **for** step in episode **do**
6:          Choose $A$ from $S$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
7:          Take action $A$, observe $R$ and $S'$
8:          Update: $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_{a \in \mathcal{A}(s)} Q(s', a) - Q(S, A)\right]$
9:          $S \leftarrow S'$
10:     **until** $S$ is terminal
11: **end for**
```

2.4. Approximation Methods

Heretofore, we analysed models from DP and TD classes, where in both occasions we assumed to deal with a finite MDP, hence with a discrete number of states and actions. In this setting, even though the computational expense could result quite demanding, we can always enumerate all the possible combinations of states and actions ($|\mathcal{S} \times \mathcal{A}|$ is finite), allowing us to manage the exploitation/exploration dilemma and all the other issues in a direct and explicit way.

In almost all the practical scenarios, however, at least one of $\mathcal{S}$ and $\mathcal{A}$ is continuous or really large. This situation makes the aforementioned algorithms unusable, forcing us to find other solutions. One problem that arises when the state space is very large is that almost every state encountered has never been seen before, hence it is not possible to gather information on that exact state from the past experience. Therefore, to make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. One way to achieve so relies on the so-called *function approximation* method. This approach generally exploits supervised learning models which take samples from the evaluation of a desired function (usually the state-value function or the action-value function) in some state-action pairs and try to generalize the function to construct an approximation of it in the entire domain $\mathcal{S} \times \mathcal{A}$. In this way, our value function depends also from a new vector of parameters w which, however, consents us to drastically reduce the dimensionality of the problem ($w \ll |\mathcal{S}|$).

In the next chapter we introduce Fitted Q-Iteration (FQI), a model-free, off-policy and offline reinforcement learning algorithm which represents the starting point of our work. After that, we talk about Persistent Fitted Q-Iteration (PFQI), an algorithm which exploits FQI structure but reiterates the same action for k steps for reasons that we will analyse in detail below.

2.4.1. Fitted Q-Iteration

As we previously explained, reinforcement learning aims to find an optimal control policy from interaction with a system or from observations gathered from a system. In batch mode, this can be achieved by approximating the action-value function based on a set of four-tuples $(s_t, a_t, r_{t+1}, s_{t+1})$, where s_t indicates the system state at time t, a_t the action taken, r_{t+1} the instantaneous reward obtained after taking action a_t and s_{t+1} the system state at time $t+1$, and by determining the control policy from this action-value function.

As we anticipated in Section 2.4, action-value function approximation may be obtained from the limit of a sequence of supervised learning problems. A particularly attractive framework is the one used by Ormoneit and Sen (2002) which applies the idea of fitted value iteration (Gordon, 1999) to kernel based reinforcement learning, and reformulates the action-value function determination problem as a sequence of kernel-based regression problems. This approach is called *Fitted Q-Iteration* (FQI). Fitted Q-Iteration is model-free, off-policy and offline algorithm that allows the agent to learn the optimal policy without directly interacting with the environment. Indeed, it yields an approximation of the action-value function corresponding to an infinite horizon optimal control problem with discounted rewards by iteratively extending the optimization horizon. The dataset required by FQI is in the form:

$$\mathcal{D} = \{(s_t^{(i)}, a_t^{(i)}, r_{t+1}^{(i)}, s_{t+1}^{(i)})\}_{i=1}^{|\mathcal{D}|},$$

meaning that for each t we collect $|\mathcal{D}|$ samples of tuples in which we have the state at time t, the action taken at time t, the reward obtained after that and the state at time $t+1$. Having collected the dataset $\mathcal{D}$, the aim of FQI is to generalize the action-value function over the whole space $\mathcal{S} \times \mathcal{A}$ applying regression techniques over $\mathcal{S}$.

As aforementioned, FQI works in an iterative fashion:

- **first iteration**: the action value function is estimated through a 1-step optimization. In particular, it is approximated as the expected value of the reward at time t, given the state at time t and the action taken at time t ($Q_1(s,a) := \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$), by applying a regression algorithm (we will talk about it in detail later) to a training set whose inputs are the pairs $(s_t^{(i)}, a_t^{(i)})$ and whose outputs are the instantaneous rewards $r_{t+1}^{(i)}$. This step is justified by the fact that, as aforementioned (2.9), if we consider only a one-step sight, $Q(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$, hence we are trying to approximate the action-value function considering the 1-step reward as return;
- ***J*-th iteration**: as can be deducted from this reasoning, an approximation of a Q_J corresponding to an J-step optimization horizon is derived using again a regression algorithm and exploiting the concept of *bootstrapping*. Indeed, the input of the regression algorithm remains the same, whilst the output values of the training set are refreshed by using the value iteration update rule based on the approximate Q_{J-1} function returned at the previous step:

$$y^{(i)} = r_{t+1}^{(i)} + \gamma \max_{a \in \mathcal{A}(s)} Q_{J-1}(s_{t+1}^{(i)}, a)\,, \tag{2.27}$$

where $y^{(i)}$ stands for the output of i-th sample.

This procedure could erroneously encourage us to perform a huge number of iterations. It is true that, iteration after iteration, the optimization horizon expands, leading theoretically to a better approximation of the true action-value function. Indeed, $Q_J = \mathbb{E}\left[\sum_{k=0}^{J-1} R_{t+k+1} | S_t = s, A_t = a\right]$, hence Q_J is closer to the actual Q with respect to Q_M when $M < J$ (the function whose expected value is calculated is closer to the actual return G_t).

Nevertheless, the regression algorithm starts learning with the target at one-step optimization, *i.e.* the instantaneous reward r_{t+1}, introducing inevitably an error. But this is a problem: indeed, at the following step, the target for approximating Q_2 depends on Q_1, encapsulating the information deriving from the previous step but also the error. In this way, the bias error propagates, growing iteration after iteration. This is compounded by another problem: at each iteration other than the first, a maximum operation over the action-value function estimated at the previous step is performed. This operation, if protracted for a long time, introduces an overestimation bias: indeed, the action-value function tends to increase its value even if no real improvement has happened.

Therefore, it is evident that the best number of iteration is not 1 but neither $+\infty$; the right value is to be tuned through a model selection procedure.

Algorithm 2.2 Fitted Q-Iteration (FQI)

```
1: **Input**:
   Batch of Transitions: $\mathcal{D} = \{(s_t^{(i)}, a_t^{(i)}, r_{t+1}^{(i)}, s_{t+1}^{(i)})\}_{i=1}^{|\mathcal{D}|}$
   Number of iterations: $J$
   Action-value function: $Q_0(\cdot,\cdot) = 0$
   Regressor: $Reg$
2: **Output**:
   Greedy policy $\pi$
3: for $j = 1, \ldots, J$ do
4:    for $i = 1, \ldots, |\mathcal{D}|$ do
5:       Set $x^{(i)} = \left(s_t^{(i)}, a_t^{(i)}\right)$
         $y^{(i)} = r_{t+1}^{(i)} + \gamma \max_{a \in \mathcal{A}} Q_{j-1}\left(s_{t+1}^{(i)}, a\right)$
6:       $Q_j = Reg.fit(x, y)$
7:    end for
8: end for
9: $\pi(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} Q_J(s,a) \ \ \forall s \in \mathcal{S}$
```

2.4.2. Persistent Fitted Q-Iteration

The FQI algorithm is quite powerful, nontheless it is not able to provide answers to some questions. Are we sure that the natural control frequency of the problem is the best one? Is there a way to consider any control frequency I want? For instance, if we consider a generic trading environment how can we be sure that the control frequency we choose (one minute, ten minutes, one hour, one day...) is the one that gives the best performances? At first glance, the answer to the previous question could seem to choose the highest frequency possible. In this way there is more freedom in choosing action combinations: indeed, if I work with the lowest time-steps, I can always replicate lower frequency policies. However, this answer is only partially true and it is refuted by Metelli et al. [32].

In order to better explain so, they introduced the concept of *persistence*, with the aim of finding the optimal control frequency in the trade-off created by the desire of having a high control and a low sample complexity. Indeed, when increasing the control frequency, the advantage of individual actions becomes infinitesimal, making them almost indistinguishable for standard value-based RL approaches [47]. As a consequence, the sample complexity increases. Instead, low frequencies allow the environment to evolve longer, making the effect of individual actions more easily detectable.

More technically, *persistence* consists in the repetition of an action for a fixed number of decision steps. In terms of MDP's, if we consider a discrete-time MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ modeled considering the highest possible control frequency, persistence can be seen as an environmental parameter k which can be configured to generate a family of related decision processes $\mathcal{M}_k = \langle \mathcal{S}, \mathcal{A}, P_k, R_k, \gamma^k \rangle$ in which, whenever the agent takes an action, the resulting transition lasts for k steps, with all the one-step rewards collected (with discount) in the new distribution R_k.

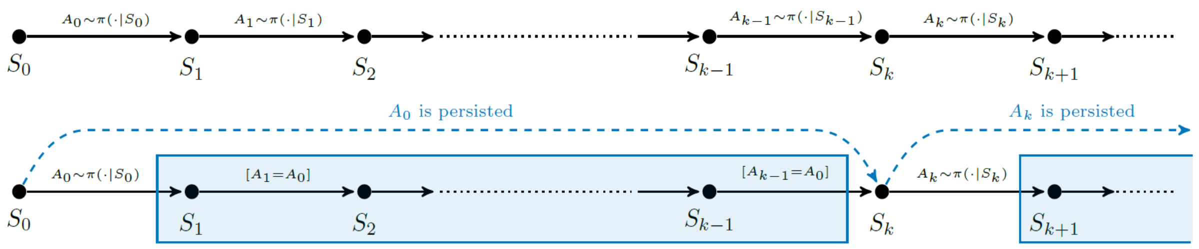

$A_0 \sim \pi(\cdot|S_0)$
$A_1 \sim \pi(\cdot|S_1)$
$A_{k-1} \sim \pi(\cdot|S_{k-1})$
$A_k \sim \pi(\cdot|S_k)$
S_0
S_1
S_2
S_{k-1}
S_k
S_{k+1}
A_0 is persisted
A_k is persisted
$[A_1 = A_0]$
$[A_{k-1} = A_0]$

Figure 2.2: Agent-environment interaction without (top) and with (bottom) action persistence, highlighting duality. The transition generated by the k-persistent MDP $\mathcal{M}_k$ is the cyan dashed arrow, while the actions played by the k-persistent policy are inside the cyan rectangle (Source: [32]).

This new framework can be applied to the aforementioned FQI, creating a new algorithm called *Persistent Fitted Q-Iteration*. This algorithm is an extension of Fitted Q-Iteration with the goal of learning the optimal value function at a given persistence.

Algorithm 2.3 Persistent Fitted Q-Iteration (PFQI)

```
1: Input:
   Persistence: $K$
   Batch of Transitions: $\mathcal{D} = \{(s_t^{(i)}, a_t^{(i)}, \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)}, s_{t+K}^{(i)})\}_{i=1}^{|\mathcal{D}|}$
   Number of iterations: $J$
   Action-value function: $Q_0^{(K)}(\cdot,\cdot) = 0$
   Regressor: $Reg$
2: Output:
   Greedy policy $\pi$
3: for $j = 1, \dots, J$ do
4:     for $i = 1, \dots, |\mathcal{D}|$ do
5:         Set $x^{(i)} = \left(s_t^{(i)}, a_t^{(i)}\right)$
           $y^{(i)} = \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)} + \gamma^K \max_{a \in \mathcal{A}} Q_{j-1}^{(K)}\left(s_{t+K}^{(i)}, a\right)$
6:         $Q_j^{(K)} = Reg.fit(x, y)$
7:     end for
8: end for
9: $\pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}(s)} Q_J^K(s,a) \ \ \forall s \in \mathcal{S}$
```

Among other things, PFQI reduces the overestimation of the action-value function typical of FQI. This happens because, with the new formulation of the MDP, less maximum operations are performed when estimating the action-value function at the same estimation quality.

2.5. Regressors for Fitting Q

2.5.1. Extremely Randomized Trees

In order to talk about the first regression approach used in this work, we take our cue from the paper of Geurts et al. [19] where this approach is introduced for the first time.

Extremely Randomized Trees (better known as *Extra-Trees*) is a tree-based ensemble method for standard batch-mode supervised classification and regression problems. It essentially consists of randomizing strongly both feature and cut-point choice while splitting a tree node. In the extreme case, it builds totally randomized trees whose structures are independent of the output values of the learning sample. Besides accuracy, the main strength of the resulting algorithm is computational efficiency.

In particular, the Extra-Trees algorithm builds an ensemble of unpruned decision or regression trees according to the classical top-down procedure. Its two main differences with other tree-based ensemble methods are that it splits nodes by choosing cut-points fully at random and that it uses the whole learning sample (rather than a bootstrap replica) to grow the trees.

It has three parameters: the number of attributes randomly selected at each node (K), the minimum sample size for splitting a node (`min_split`) and the minimum sample size for a node to be a leaf node (`min_leaf`). This approach is then replicated M times, generating an ensemble model of M trees. These four parameters have different effects: K determines the strength of the attribute selection process, `min_split` and `min_leaf` the strength of averaging output noise, and M the strength of the variance reduction of the ensemble model aggregation. Indeed, the predictions of the trees are aggregated to yield the final prediction by majority vote in classification problems and arithmetic average in regression problems. From the bias-variance point of view, the rationale behind the Extra-Trees method is that the explicit randomization of the cut-point and feature combined with ensemble averaging should be able to reduce variance more strongly than the weaker randomization schemes used by other methods. The usage of the full original learning sample rather than bootstrap replicas is motivated in order to minimize bias. From the computational point of view, the complexity of the tree growing procedure is, assuming balanced trees, on the order of $N \log N$ with respect to learning sample size, like most other tree growing procedures. However, given the simplicity of the node splitting procedure we expect the constant factor to be much smaller than in other ensemble based methods which locally optimize cut-points.

2.5.2. XGBoost

Another algorithm very used for supervised learning problems is *Extreme Gradient Boosting* (*XGBoost*). This algorithm was developed as a research project at the University of Washington by Tianqi Chen and Carlos Guestrin [7]. Since its introduction, this algorithm has not only been credited with winning numerous Kaggle competitions but also for being the driving force under the hood for several cutting-edge industry applications.

Going into technicalities, XGBoost is an efficient open-source implementation of the gradient boosted trees algorithm. In order to describe it in detail, we must first introduce some other approaches:

1. *Decision Trees.* They represent a simple and naive way to organize data and predict classification and regression outcomes from them. They are built starting from

a root node which represents the first information (feature) we deal with in our process. This node is then split based on a criterion, generating two or more nodes. Afterwards, each of these nodes is split as well based on other conditions and so on, until the process is stopped generating some leaf nodes. If we are dealing with classification, each path from the root to a leaf node represents a specific class. Instead, with regression, the reached leaf identifies a predicted value for a specific new case;

2. *Bagging.* Bagging is an ensemble method which consists in averaging multiple models together in order to reduce the variance. Multiple datasets are produced by a single dataset through *bootstrap* approach: a certain number of samples of size B are generated (called bootstrap samples) from an initial dataset of size $N(N > B)$ by randomly drawing with replacement B observations. Therefore, a model for each bootstrap sample is trained. In classification setting the result is obtained by majority voting among the models; in the regression one, instead, the predicted value derives from the average of the predicted values estimated by every model;

3. *Random Forest.* Random forests are just a bagging procedure adopted on decision trees where, additionally, another bagging scheme is used: a modified tree learning algorithm is adopted that selects, at each candidate split in the learning process, a random subset of the features;

4. *Boosting.* The aim of boosting is to reduce the bias. This is achieved by sequentially training weak learners in the following way: some weights are associated to each training sample. After one step of training, the weights of all the samples that the model predicted wrong are increased. Thus, a new training set is obtained and the procedure is repeated until the result is satisfying. Eventually, the final prediction is the weighted prediction of every weak learner;

5. *Gradient Boosting.* Gradient Boosting is a special case of boosting algorithm where errors are minimized by a gradient descent algorithm. Gradient Descent algorithm iteratively optimizes the loss of the model by updating weights. For regression algorithms, we use MSE (Mean Squared Error) loss as an evaluation metric while, for classification problems, logarithmic loss is exploited.

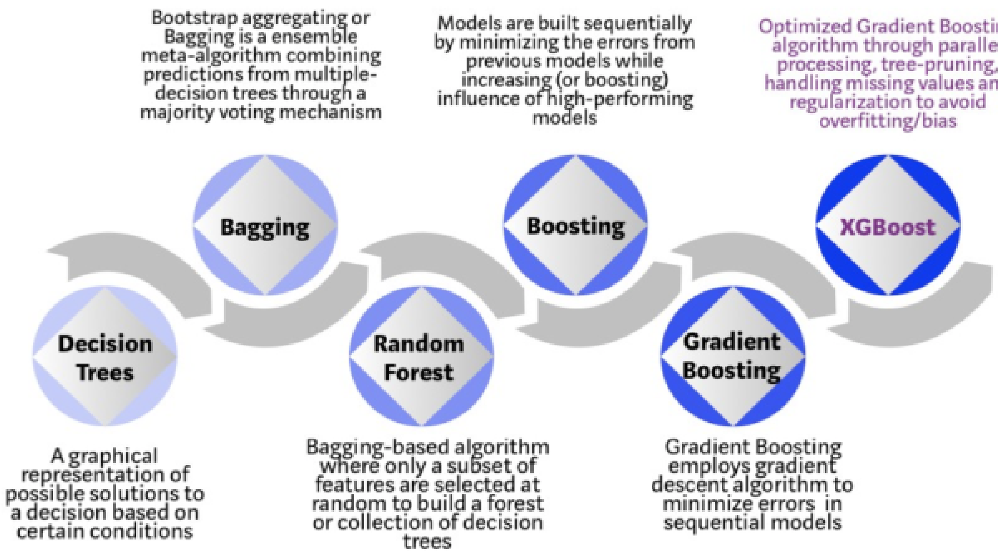

Bootstrap aggregating or Bagging is a ensemble meta-algorithm combining predictions from multiple-decision trees through a majority voting mechanism
Models are built sequentially by minimizing the errors from previous models while increasing (or boosting) influence of high-performing models
Optimized Gradient Boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias
Bagging
Boosting
XGBoost
Decision Trees
Random Forest
Gradient Boosting
A graphical representation of possible solutions to a decision based on certain conditions
Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees
Gradient Boosting employs gradient descent algorithm to minimize errors in sequential models

Figure 2.3: Evolution of XGBoost algorithm from Decision Trees (Source: https://towardsdatascience.com).

XGBoost is a more regularized form of Gradient Boosting: it minimizes a regularized (L1 and L2) objective function that combines a convex loss function (based on the difference between the predicted and target outputs) and a penalty term for model complexity (in other words, the regression tree functions). XGBoost delivers high performance as compared to Gradient Boosting. Its training is very fast and can be parallelized across clusters.

3 | Related Works

In this chapter, we introduce all the works that are in some way related to the new algorithm we are going to present. The chapter is organized in this way: firstly, we introduce reinforcement learning approaches, paying particular attention to deep reinforcement learning and persistence ones. Secondly, after an overview of hierarchical reinforcement learning, we look at how works of this type have been declined in the finance sector and finally we talk about a method that is particularly relevant to us.

3.1. Reinforcement Learning in Finance

Reinforcement learning has been applied to a variety of problems, ranging from simple games [43, 52] to complex real-world tasks such as autonomous robots [10, 21, 27, 33, 57], self-driving cars [8, 9, 28, 36, 54], and control systems [6, 15, 22, 49, 55]. Notably, one of the problems to which reinforcement learning has been applied is finance, and we are delving into it in more details in the following sub-sections.

3.1.1. Deep Reinforcement Learning

Several finance-related reinforcement learning works have flourished in recent years, finance being a suitable environment for such models. Given that enumerating all of them would be almost impossible, we decided to display only a few Deep Reinforcement Learning (DRL) approaches, since they are recent and cutting-edge as well as the hierarchical reinforcement learning ones. In general, the advantages of DRL methods are threefold [2]:

1. DRL allows training an end-to-end agent, which takes available market information as input state and outputs trading actions directly;
2. DRL-based methods optimize overall profit directly and bypass financial prediction as the intermediate task;
3. DRL can directly incorporate task-specific constraints into the training process.

One of them is *DeepScalper* [39], a deep reinforcement learning framework for intraday trading. To efficiently incorporate both micro-level and macro-level market information, an encoder-decoder architecture is proposed to learn robust market embedding. To capture the overall price trend, a novel hindsight reward function is designed with a long-term profit regularizer to provide the agent with the long-term horizon. In particular, to learn a robust intraday market embedding, two encoders are used to represent the market from micro-level and macrolevel respectively. *DeepTrader* [51], instead, is a DRL method to optimize the investment policy. DeepTrader mainly includes two units to handle the problems of cross-asset interrelationship learning and risk-return balancing, respectively. One unit, which is called asset scoring unit, takes individual stock data as input and learns to represent them as "winner scores" for each asset. The winner score indicates how likely a stock is going to rise in the future. To better encode the interrelationships among all stocks, the creators constructed a graph based on different dependence characterizations to capture interrelationships hierarchically, whether long or short, spatial or temporal. The other unit, known as market scoring unit, leverages the market sentiment indicators as input and then embeds financial situations as an indicator to adjust the proportion of long and short funds in every trading period. In the asset scoring unit dilated temporal convolution layer (TCN) ([53]) are used to handle temporal relations in long range sequences. To model short-term spatial properties, the developers adopted an attention mechanism ([17]) to adaptively search the correlations between stocks. Finally, the long-range relationships is modeled by graph convolution networks (GCNs) to better guide the decision making.

3.1.2. Persistence in Reinforcement Learning

The concept of *persistence*, introduced in Section 2.4.2, and in particular of PFQI (Section 2.4.2) has been exploited in forex context by Riva et al. [38]. In that field, persisting an action could be essential because changing allocation too often can cause high transaction costs and, in general, learning trading opportunities at different time-scales in not equally difficult and cannot be done in a unique manner.

3.2. Hierarchical Reinforcement Learning

3.2.1. General Setting

The hierarchical approach to reinforcement learning occurred few years ago with the aim of autonomously decomposing long-horizon decision-making tasks into simpler *subtasks*. In general, the structure is the following: there is a *higher-level policy* which learns to perform the task by choosing optimal subtasks. These subtasks may themselves be reinforcement learning problems with a lower-level policy and so on, until some *primitive actions* are taken. The hierarchy of policies obtained in this way collectively determines the behavior of the agent.

Hierarchical Reinforcement Learning (HRL) is formalized on the basis of the theory of *Semi-Markov Decision Process* (SMDP) [4]. A SMDP is a stochastic control process similar to an MDP (Section 2.1.1) but, unlike MDP, it also involves the concept of time for which an action is executed after it has been chosen. In this way, for any subtask chosen at time t, we can define new transition and reward functions. Notably, the latter represents the expected cumulative reward obtained while following the subtask policy from time t until the termination of that subtask, and, consequently, a new action-value function. More formally, we can define two important components of the hierarchical framework: the *subtask space* Ω_H and the *hierarchical policy* π_H. The former identifies the super-set of all the subtasks used in a hierarchy, while the latter represents the complete state-to-subtask-to-action mapping from the lowest level policy, *i.e.* the policy that selects primitive actions. Thus, the action-value function associated to the HRL agent can be written as:

$$Q_H(s,a) := \mathbb{E}_{a\sim\pi_H|\Omega_H}\left[\sum_{k=0}^{\infty}\gamma^{t+k}R_{t+k+1}|S_t = s, A_t = a\right],$$

where γ is the discount factor, S_t is the state at time t, A_t is the action selected at time t, $r(s_t, a_t)$ is the reward received at time t and, in general, $a \sim \pi_H$ means that a primitive action a is sampled using π_H conditioned on the subtask Ω_H. As we can expect from those definitions, the primary goal of HRL is to find the *optimal hierarchical policy* π_H^* given a task. The secondary one is *subtask discovery*, which refers to automatically finding the optimal subtask space Ω_H^* using the HRL agent's experience data. Subtask discovery is not essential, because a subtask space could be handcrafted using precise domain knowledge. All in all, these two parts are the solution of:

$$\Omega_H^*, \pi_H^* = \underset{\Omega_H}{\operatorname{argmax}}\,\underset{\pi_H|\Omega_H}{\operatorname{argmax}}\, Q_h(s,a) \quad \forall s \in \mathcal{S}\ \forall a \in \mathcal{A}\,.$$

3.2.2. Approaches for Hierarchical Reinforcement Learning

In this section we provide a brief visualization of the main approaches developed for this kind of framework. We can roughly divide all the works by four different procedures:

1. *Single agent, single task, without subtask discovery*;
2. *Single agent, single task, with subtask discovery*;
3. *Multiple agents, single task*;
4. *Single agent, multiple tasks.*

Single agent, single task, without subtask discovery

In this context there are two additional subdivisions: *feudal hierarchy*, where the action space of a higher-level policy is determined by the subgoals linked to various subtasks, and *policy tree*, where the action space of a higher-level policy consists instead of the different lower-level policies of subtasks. The main difference between the two approaches relies on the exploitation of the universal policy: in the first case, we consider a universal policy whose input is the subgoal chosen by the higher-level policy, whereas in the second one each subtask is represented by a separate policy.

Feudal hierarchy has been exploited by Dayan and Hinton [14] in what is called *feudal reinforcement learning*, where there is a higher-level *manager* who communicates a subtask via a subgoal to a lower-level *worker* who, in turn, has to reach that subgoal. Kulkarmi et al. [26] proposed a similar approach called *Two level DQN* in which the manager network is trained through principal task rewards and selects subgoals, whereas the worker one takes the subgoal as input and selects primitive actions in order to reach that subgoal. To address the non-stationarity issue generated by the aforementioned methods, Nachum et al. [34] suggested *HIRO*, where a subgoal re-labelling is exploited: if the agent does not achieve the subgoal after a fixed number of time-steps, then the subgoal is re-labelled in the transition data with another subgoal drawn from a distribution of subgoals that maximize the probability of the observed transitions. Then, the higher-level policy treats the new subgoal as its output in the hindsight, which correlates better with the observed transitions.

Policy tree approaches are led by *Options* framework introduced by Sutton et al. [46] and *MAXQ value function decomposition* [16]. In Options framework the original MDP is extended and generalized with options, predefined by the programmer, into the action space of an agent. In particular, a single-step option is the primitive action itself. Given that the option blends into the original MDP, its transferability is limited but the optimality of the hierarchical policy is guaranteed. In MAXQ approach the original MDP is instead split into smaller sub-MDPs, thus any sub-MDP policy is learned separately from the others decomposing the main action-value function into different action-value functions, one per subtask. Hence, optimality of the hierarchical policy is not guaranteed but the option can be used for other subtasks. Interesting the *Algorithms for Batch Hierarchical Reinforcement Learning* work conducted by Tiancheng Zhao and Mohammad Gowayyed [56]. In this paper three new algorithms are introduced: *Subtask Q-value Iteration*, *Hierarchical Q-value Iteration* and *Fitted Subtask Q-value Iteration*). All of them are strictly related to Fitted Q-Iteration 2.4.1 and one, Hierarchical Q-value iteration, to the hierarchical reinforcement learning framework as well. The general framework derives by *MAXQ* [16] and *Options* [46], where the MDP can be decomposed into a finite set of subtasks $\Omega = \{\Omega_0, \Omega_1, \ldots, \Omega_{n-1}, \Omega_n\}$ with the convention that Ω_0 is the higher-level subtask, *i.e.* it solves the original MDP; each Ω_i is then a Semi-Markov Decision Process [4]. In this way, the hierarchical policy π_H is defined as a set of policies for each subtask ($\pi_H = \{\pi_0, \pi_1, ..., \pi_n\}$). The goal is to learn the recursive optimal hierarchy policy π_H^*, *i.e.* the hierarchical policy such that, for each subtask Ω_i, the corresponding policy π_i is optimal for the SMDP. To do so, they propose the Hierarchical Q-value Iteration, which roughly trains every subtask using Subtask Q-Value Iteration in a bottom-up fashion. The training prerequisite of SQI for a specific subtask is that all of its children have converged to their greedy optimal policies. In order to fulfil this constraint, HQI starts running SQI from subtasks whose children have only primitive actions. After those subtasks converge to their optimal policy, the algorithm continues to other subtasks whose children are either converged or primitive actions.

Single agent, single task, with subtask discovery

Also in the context of subtask discovery we find two lines of thought: one where the subtask space Ω_H is discovered simultaneously to the hierarchical policy learning, and another where these two events are independent (in this case, the subtask discovery usually happens before the hierarchical policy learning).

In the first group we can see both policy tree approaches, such as *Graphical model for unified Option Discovery* [13], *Skill Chaining* [25] and *Option Critic* [3], and feudal hierarchy ones, such as *Feudal Networks* [48]. The second one comprehends different methods like the one proposed by McGovern and Barto [30] which works with bottlenecks on graphs, the *Q-cut* [31] which exploits Max-Flow/Min-Cut Algorithm to deal better with them and *L-cut* [11] where local sub-graphs are used rather than a global graph.

Multiple agents, single task

Starting from the multi-agent reinforcement learning idea where multiple agents learn to coordinate their polices by optimizing a joint task objective, we can exploit the benefits of task decomposition also in this setting, considering a situation where each agent performs some subtasks and different agents learn to coordinate their higher-level policies. We can split this framework in different categories:

1. *Synchronized/Non-Synchronized* termination of agent subtasks;
2. *Centralized/Decentralized with centralized learning/Fully decentralized* agents: in the first situation all the agents use a single central policy. In the second one the agents have their own hierarchical policies but they learn using the same information (state observations and subtasks). The last one identifies the situation where both policies are decentralized and there is no information sharing;
3. *Homogeneous/Heterogeneous* agents: the former ones have similar subtask spaces and hierarchical structures, hence they can independently perform the task without significant inter-dependence among them, whereas the latter ones have different subtask spaces and hierarchical structures, thus they strongly depend from each other in order to achieve the entire task.

These frameworks are exploited in several works, like *Cooperative HRL* [20], *Inter Subtask Empowerment based on Multi-agent Options* [37] and *Feudal Multi-agent Hierarchies* [1].

Single agent, multiple tasks

The general idea of this class of hierarchical reinforcement learning approaches (also called *transfer learning with HRL*) is that some subtasks are shareable across multiple related tasks, meaning that the policy learned for a subtask can be used to accelerate the adaptation of the agent also on other related tasks. One interesting approach is suggested by Konidaris et al [24], where the aim is to create policies robust to transfer to a task domain with a different state space. The invariance is achieved by using two separate state representations: one in the task-space and the other in the agent-space. The agent-space representation contains features that are dependent on the agent and invariant across various tasks. Option policies are learned in the agent-space and they can be transferred to various tasks with different task-spaces as long as the agent-space remains the same across those tasks. The higher-level policy, which selects an Option for execution, is learned using the task-space and adapts according to the active task.

3.2.3. Hierarchical Reinforcement Learning in Finance

In this section we take a look at hierarchical reinforcement learning works strictly related to finance. After a deep analysis, we can state that there is not a huge proliferation of these types of articles, which gives additional value to our cutting-edge approach.

One of these works is proposed by Biedeknapp et al. [5], where the structure is made by a hierarchy in which a *behaviour policy* determines the action to be played given the current state and a *skip policy* determines how long to commit to this behaviour. To learn the behaviour, they exploited the classical Q-learning; to learn the value of a skip n-step Q-learning can be used with the condition that, at each step of the j steps, the action stays the same. This approach is very similar to ours but actually there is a big difference: by introducing a skip-policy, there is no explicit comparison between the different persistences that can be chosen, which there will be in our hierarchy.

Another idea has risen due to COVID19 stock market crash: Suri et al. [41] propose a hierarchical framework for executing trade bids on abrupt real market experiences. The main idea is to fulfill catastrophe minimization by providing the agent with order and bid policies capable of estimating order quantities and executing bid actions respectively. At a given time-step t, the order policy observes state s_t and then follows, with a network, the sub-policy for determining the quantity for the bid, whereas the high-level bid policy utilizes the estimated quantity in conjunction with current states s_t to execute actions as trade bids.

If we move our sight to portfolio management point of view, one hierarchical work is the one drafted by Wang et al. [50]. In this work, the trading process is decomposed into a hierarchy of portfolio management over trade execution and train over the corresponding policies. In particular, the high-level policy gives portfolio weights at a lower frequency to maximize the long term profit and invokes the low-level policy to sell or buy the corresponding shares within a short time window at a higher frequency to minimize the trading cost. This approach and ours are not compatible since here there is a long holding period where the agent holds the pre-selected assets without making any purchase or selling.

Another example of portfolio management application is the article proposed by Gao et al. [18], where they exploit the DQN framework [26] obtaining a novel hierarchical Deep Q-Network, in order to divide the total portfolio value into smaller parts. In particular, each DQN is independent from the others but all of them have identical structure and are responsible for three assets, to reduce the traditional DQN. Furthermore, a controller DQN with a different structure is defined to manage two of the previous networks and lastly, for each controller, there is a controller of higher level to control them.

4 | Hierarchical Persistent Fitted Q-Iteration (HPFQI)

4.1. Motivation

The information seen hitherto helped us to get a general overview of reinforcement learning and state-of-the-art works related to finance, seasoned also by persistent and hierarchical flavours. Since all these different approaches have their own strengths, our aim is to synthesize them in a unique algorithm. More specifically, we aspire to create a machine that is able to perform trading on FX market[1] gathering information from both low and high frequencies. This aspect is really important because low frequency sight consents to intercept signal trends, whereas high frequency one enables a higher control on agent's actions.

For the knowledge we have so far, one way to achieve so would be to train and test models at different frequencies and then to choose the best performing one. For instance, if we think that the most informative frequencies for a certain market are time-steps of 1, 5 and 10 minutes, we could start training and testing three algorithms (e.g: PFQI) at persistences 1, 5 and 10 respectively. After that, we can compare them based on a certain measure (e.g.: profit obtained by the agent in test) and finally select the model with the highest score. However, in this way we are forcing ourselves to choose between various frequencies: going back to the previous example, let us say that we selected PFQI at persistence equal to 5 since it had the highest profit in test. Nonetheless, in this way our agent can change action only after 5 minutes, which makes it unresponsive in situations of high volatility and too reactive in situations where a long trend is predominant. Actually, very often the best way to behave in trading is to constantly switch from long term to short term actions, since market conditions continue to change. With this in mind, our purpose is to create an algorithm whose learning process allows a single agent to encapsulate both low and high-frequencies.

[1]All the details about FX market setting will be covered extensively in Chapter 6.

The most suitable framework for this idea is hierarchical reinforcement learning. As previously said in Section 3.2, its advantage derives from the fact that it generates a single hierarchical policy blending different subpolicies. In this way, at the end of the training process the agent will follow a single policy which, though, encloses several different policies deriving from distinct (but still intertwined) training procedures.

In order to familiarize with the new hierarchical algorithm in the easiest way possible, it came naturally to first introduce an auxiliary algorithm: APFQI. As we will see in Section 4.2, APFQI represents the core of HPFQI, the novel hierarchical model.

4.2. Algorithm

The skeleton of the hierarchical algorithm is inspired by *Algorithms for Batch Hierarchical Reinforcement Learning* work conducted by Tiancheng Zhao and Mohammad Gowayyed [56]. The structure of our algorithm is made up of several elements. The hierarchical policy π_H is obtained exploiting a set of subpolicies $\{\pi_1, \pi_2, \ldots, \pi_{p-1}, \pi_p\}$. The subpolicies, in turn, are retrieved after training some agents on different subtasks $\{\Omega_1, \Omega_2, \ldots, \Omega_{p-1}, \Omega_p\}$. Here, each Ω_i indicates the original subtask but revisited based on a particular persistence $k \in \mathcal{K}$, where $\mathcal{K}$ contains all the persistences taken into account in the whole procedure. Furthermore, p indicates the total number of persistences considered[2]. Specifically, $\mathcal{K}$ is an ordered set: indeed, its elements are organized in decreasing order. Therefore, $\mathcal{K} := \{k_1, k_2, \ldots, k_{p-1}, k_p\}$, where k_1 is the highest persistence, k_2 is the second highest and so on.

Notice that each pair (Ω_i, π_i) represents a classical reinforcement learning setting at a given persistence. In particular, each subpolicy π_i is retrieved performing some iterations of the aforementioned APFQI algorithm on the subtask Ω_i. APFQI is the acronym of *Adapted Persistent Fitted Q-Iteration* (APFQI) due to its similarity with Persistent Fitted Q-Iteration 2.4.2. To clarify, one APFQI model is trained for each persistence considered, for a total of p trainings. Given that each model can be trained with a different number of iterations, we call $\mathcal{J}$ the set of the iterations considered during the whole procedure. Similarly to what we did with persistences, $\mathcal{J} := \{J_1, J_2, \ldots, J_{p-1}, J_p\}$ where J_1 identifies the total number of iterations performed by APFQI at the highest persistence, J_2 the total number of iterations performed by APFQI at the second highest and so on. Notably, unlike $\mathcal{K}$, $\mathcal{J}$ is not necessarily an ordered set: indeed, it may happen that APFQI at the highest persistence will perform less iterations than APFQI at a lower one.

[2]Notably, p is defined such that $|\mathcal{K}| = p$.

In order to create the hierarchical algorithm, all the previous items are placed in such a sequence:

- firstly, the APFQI at the highest persistence k_1 is trained, retrieving the policy π_1 and especially the action-value function $Q_{J_1}^{(k_1)}$ obtained at the last iteration J_1;
- secondly, the APFQI at the second highest persistence k_2 is trained, exploiting the action-value function obtained at the previous step $Q_{J_1}^{(k_1)}$ in such a way that the exiting $Q_{J_2}^{(k_2)}$ brings the best from persistences k_1 and k_2 (we will soon explain what *best* means);
- iteratively, this process is implemented until the APFQI at the lowest persistence $k_p \in \mathcal{K}$ is performed, generating the final action-value function $Q_{J_p}^{(k_p)}$ which is a *summa* of all the previous ones. Eventually, the hierarchical policy is obtained as $\pi_H(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} Q_{J_p}^{(k_p)}(s,a) \quad \forall s \in \mathcal{S}$.

In light of this, how do we get the action-value function at the last iteration of a given persistence to gather the *best* information from all the action-value functions at higher persistences? In the classical Persistent Fitted Q-Iteration, the target update is done in this way:

$$y^{(i)} = \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)} + \gamma^K \max_{a \in \mathcal{A}} Q_{j-1}^{(K)} \left(s_{t+K}^{(i)}, a \right) . \tag{4.1}$$

where K is the given persistence and all the other terms are well described in Algorithm 2.3. In APFQI, instead, the first term of the update remains the same, whilst the second one changes:

$$y^{(i)} = \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)} + \gamma^K \max \left\{ \max_{a \in \mathcal{A}} Q_{j-1}^{(K)} \left(s_{t+K}^{(i)}, a \right), \max_{\substack{a \in \mathcal{A} \\ Q \in \mathcal{Q}}} Q \left(s_{t+K}^{(i)}, a \right) \right\} \tag{4.2}$$

where $\mathcal{Q}$ represent a set of pre-trained action-value functions. We used this notation to leave APFQI as general as possible, but in our case $\mathcal{Q}$ contains specifically the action-value function obtained at the last iteration of all the previous hierarchical steps. With the novel formulation, for each iteration step, each sample can decide to update the target choosing between the action-value function generated by the APFQI algorithm itself (the first term in the curly brackets, *i.e.* the classical PFQI term) and the maximum between the final action-value functions of APFQI trained at higher persistences. In this way, we believe that APFQI has the opportunity to decide whether it is better to act following higher persistence policies or its own. Notably, APFQI at the highest persistence coincides exactly with PFQI at that persistence, because the set of previous action-value functions $\mathcal{Q}$ is empty.

Algorithm 4.1 Adapted Persistent Fitted Q-Iteration (APFQI)

```
1: Input:
   Batch of Transitions: $\mathcal{D} = \{(s_t^{(i)}, a_t^{(i)}, \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)}, s_{t+K}^{(i)})\}_{i=1}^{|\mathcal{D}|}$
   Number of iterations: $J$
   Possibly, a set of action-value functions $\mathcal{Q}$
   Regressor: $Reg$
   Persistence: $K$
2: Output:
   Action-value function $Q$
3: for $j = 1, \dots, J$ do
4:    for $i = 1, \dots, |\mathcal{D}|$ do
5:       Set $x^{(i)} = \left(s_t^{(i)}, a_t^{(i)}\right)$
6:       if $j = 1$ then
7:          $y^{(i)} = \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)} + \gamma^K \max_{\substack{a \in \mathcal{A} \\ Q \in \mathcal{Q}}} Q\left(s_{t+K}^{(i)}, a\right)$
8:       else
9:          $y^{(i)} = \sum_{k=1}^{K} \gamma^{k-1} r_{t+k}^{(i)} + \gamma^K \max \left\{ \max_{a \in \mathcal{A}} Q_{j-1}^{(K)}\left(s_{t+K}^{(i)}, a\right), \max_{\substack{a \in \mathcal{A} \\ Q \in \mathcal{Q}}} Q\left(s_{t+K}^{(i)}, a\right) \right\}$
10:      end if
11:      $Q_j^{(K)} = Reg.fit(x, y)$
12:   end for
13: end for
14: $Q = Q_J^K(s, a) \;\; \forall s \in \mathcal{S}$
```

Algorithm 4.2 Hierarchical Persistent Fitted Q-Iteration (HPFQI)

```
1: Input:
   Set of persistences: $\mathcal{K} = \{k_1, k_2, \dots, k_{p-1}, k_p\}$ where $k_1 > k_2 > \dots > k_{p-1} > k_p$
   Number of iterations: $\mathcal{J} = \{J_1, J_2, \dots, J_{p-1}, J_p\}$
   Batch of Transitions[3]: $\mathcal{D} = \{(s_t^{(i)}, a^{(i)}, r_{t+1}^{(i)}, s_{t+1}^{(i)}, \dots, r_{t+k_1}^{(i)}, s_{t+k_1}^{(i)})\}_{i=1}^{n}$
   Initialize: $\mathcal{Q} = \emptyset$
2: Output:
   Greedy hierarchical policy $\pi_H$
3: for $k \in \mathcal{K}$ do
4:     APFQI($k$, $\mathcal{Q}$)
5:     $\mathcal{Q} \leftarrow \mathcal{Q} \cup Q_{J_k}^{(k)}$
6: end for
7: $\pi_H(s) = \operatorname*{argmax}_{a \in \mathcal{A}(s)} Q_{J_p}^{k_p}(s, a) \;\; \forall s \in \mathcal{S}$
```

The whole algorithm is called *Hierarchical Persistent Fitted Q-Iteration* (HPFQI). As shown in Section 4.2, HPFQI starts training APFQI with the highest persistence and ends with APFQI at the lowest one. The reason for this choice is straightforward: at low frequency it is easy to detect a trend in the signal and not to remain stuck in signal noise but the control power is low. For instance, if the curve changes slope abruptly the agent may not react immediately. On the other hand, at high frequency the actions can be changed very often but it is more likely to learn noise instead of useful information. For instance, the agent may decide to buy because there is an immediate rise in the value of the security when, however, the general trend of the curve is downward. Therefore, the purpose of HPFQI is both to learn the trend easily and eventually to have a policy with a high control power.

Furthermore, this algorithm has a technical advantage over both FQI and PFQI. As PFQI, for the way it is constructed, performs less maximum operations than FQI for the same approximation accuracy; this leads it to be less prone to the overestimation problem. The advantage over PFQI, instead, relies on the MDP structure: as we saw in Section 2.4.2, PFQI exploits a MDP different from the original one. Conversely, if the lowest persistence coincides with the original base frequency, the final outcome of HPFQI is exactly in the form of the original MDP.

[3]Actually, the dataset is not collected precisely like this, since there would be several repetitions. In the algorithm it is defined in such a way only for a matter of clarity.

4.2.1. Comparison with HQI

As aforementioned, to design the structure of HPFQI we took our cue from *Algorithms for Batch Hierarchical Reinforcement Learning* [56]. In particular, as widely described in Section 3.2.2, their main result was about a new hierarchical algorithm called HQI. The main difference between HPFQI and HQI is the following: HQI starts training from primitive actions considering gradually higher subpolicies, whereas HPFQI does it in the opposite way, beginning with persistent actions and concluding with primitive ones. In HQI setting, that flow is chosen because it is assumed that each SQI step reaches the optimal policy. On the contrary, in our setting that hypothesis unfortunately does not hold. Indeed, the concept of persistence has been introduced [38] because in many problems FQI with primitive actions is not able to reach the optimal policy, whereas PFQI learns better. Anyway, having the ability to choose the persistence is definitely an advantage.

So far, we have simply introduced the new algorithm conceptually and justified its choice. Now it is time to see whether the results we expect are actually reflected in real cases or not. As we have already mentioned several times, our aim is to use it within the FX market. However, we decided to test it first in a simpler environment, which will be described in the next section.

5 | HPFQI Evaluation on Mountain Car Environment

5.1. Problem Formulation

5.1.1. Environment Description

Mountain Car is a classic OpenAI gym environment used to test and evaluate Reinforcement Learning algorithms for learning control policies in continuous environments. The problem involves a car positioned between two hills or mountains and the goal of the car is to reach the top of the mountain on the right. The car is subject to gravity, which means that it cannot simply drive up the right mountain, but instead must build up momentum by repeatedly driving up the left mountain and then coasting back down the hill towards the right mountain.

The environment is represented by a two-dimensional continuous state space, where the car's position and velocity are the two state variables. The car's position can range from -1.2 to 0.6, and its velocity can range from -0.07 to 0.07. At each time step, the car can take one of three possible actions: accelerate to the left, accelerate to the right, or do nothing. The goal of the Reinforcement Learning algorithm is to learn a policy such that the car reaches the top of the right mountain as quickly as possible. The reward function is designed to encourage the car to make progress towards the right mountain, and penalize it for taking too long to reach the goal.

The Mountain Car problem has been widely used as a benchmark for evaluating Reinforcement Learning algorithms [12, 35, 40, 42, 45] .

5.1.2. Motivation

Although our main goal is to use the HPFQI algorithm in the FX market, we decided to exploit it first in the Mountain Car setting. We chose to do so for the following reasons: firstly, FX market is a very complex setting, with an inherent high level of stochasticity and unpredictability. Therefore, in such a scenario it may be difficult to come straight to conclusions, both in positive and negative way. On the other hand, Mountain Car is a much less complex environment, having it only two states and especially being almost totally deterministic[1]. Secondly, this OpenAI gym environment fits really well with persistent algorithms: indeed, persistence enables the agent to have a far-sighted view useful both for exploring better the state and action spaces and for accumulating the momentum necessary to reach the top of the mountain. Moreover, this could work even better with our hierarchical approach to persistence: in fact, the agent is not forced to act only with one length of view. In this way, at each time-step the agent can choose how far to look ahead and thus how much to keep the same action depending on the situation.

5.1.3. Mountain Car as an MDP

MDP

The way it was conceived, Mountain Car is easily structurable as an MDP:

- as we have already anticipated, the *state space* $\mathcal{S}$ is made up of two state variables: the car's position $S^0 \in [-1.2, 0.6]$ and the car's velocity $S^1 \in [-0.07, 0.07]$;
- There are three *actions* in $\mathcal{A}$ the agent can choose from: accelerate to the left ($A = -1$), accelerate to the right ($A = 1$), or do nothing ($A = 0$);
- the *reward function* R can assume only two values: -1 if in a given step the agent does not reach the goal, 0 if it succeeds;
- the *state-transition probability function* P is deterministic, meaning that the new state is fully determined by the current state and action;
- the *discount factor* γ can be chosen according to the user specifications; we set it at $\gamma = 0.995$.

Finally, since the Mountain Car problem satisfies the Markov property (the future state depends only on the current state and action, not on the past history), its modelling as an MDP is justified.

[1]As we will see, the transition probability is totally deterministic, whereas the initial state is partially/totally random.

Mountain Car as Episodic Task

Mountain Car has been conceived with an episodic structure. This formulation allows the algorithm to focus on learning policies that lead to successful completion of episodes, rather than trying to maximize the return over an infinite time horizon.

To formulate the Mountain Car problem as an episodic task, we need to define the end of an episode. One way to do this is to set a maximum number of time steps per episode, number that we call l_{ep}. At the start of each episode, the car is placed at a random position within a range of values, *i.e.* $S_0^0 \in [-1.2, 0.6]$, and with zero velocity[2], hence $S_0^1 = 0$. The agent then selects actions to accelerate the car and attempt to reach the goal at the top of the hill. If the car reaches the goal within the specified maximum number of time steps, the episode terminates with a final reward of 0 and a return equal to the discounted sum of the rewards accumulated up to that point. Otherwise, the episode terminates after the maximum number of time steps and the agent receives return of $-\sum_{k=0}^{l_{ep}-1} \gamma^k$.

After an episode ends, the environment is reset to a new starting state and the agent continues learning in a new episode. The goal of the agent is to maximize the return obtained across multiple episodes, *i.e.* to minimize the steps needed to reach the goal.

5.1.4. Dataset

Given that we use the episodic framework, the dataset $\mathcal{D}$ is constructed accordingly. In fact, $\mathcal{D}$ represents the collection of a certain number of episodic datasets. For clarity, we call n_{ep} the number of episodes considered and $\mathcal{D}_{ep}$ the dataset collected in a single trajectory.

Original Construction

The original dataset, *i.e.* considering persistence equal to one, is collected as expected: for each episode, we consider a dataset

$$\mathcal{D}_{ep} = \{(s_t, a_t, r_{t+1}, s_{t+1})\}_{t=0}^{g-1}, \tag{5.1}$$

where s_t represents position and velocity at time t, a_t the action taken at time t, r_{t+1} the reward obtained by taking that action, s_{t+1} the new position and velocity after have taken the action a_t. The parameter g stands for the minimum between the maximum length of

[2]The original definition of the environment considers the initial velocity to be zero; actually, in our experiments we considered the initial velocity both null and random within the range $[-0.07, 0.07]$.

the episode l_{ep} and the time-step, if any, in which the goal is reached.

Construction with persistence

However, when we are working with persistences with values different from one, we need to reshape the dataset. In general, calling k the persistence value, the new dataset is made up as follows:

$$\mathcal{D}_{ep} = \left\{ \left(s_{kt'}, a_{kt'}, \sum_{i=1}^{k} \gamma^{i-1} r_{kt'+i}, s_{k(t'+1)} \right) \right\}_{t'=0}^{g/k-1}, \tag{5.2}$$

where the action $a_{kt'}$ is repeated for k times from time-step kt' to $k(t'+1)-1$. In other words, if we consider an easy reparametrization $t = kt'$, the formulation (5.2) tells us that each sample takes in consideration the position and velocity at time t and the action taken at time t, the position and action that will be detected at time $t+k$ and the discounted cumulative reward retrieved from t to $t+k$; for clarity, we call this last term *persistent reward*. Therefore, the only difference from (5.1) is that the step from t' to $t'+1$ could actually be longer than one original time-step: if it happens, the action a_t is propagated for any time step between t and $t+k$, and the persistent reward is a discounted sum of the original rewards from t to[3] $t+k$. In view of this, we understand that $t = \{0, \ldots, g\}$ is the *original time discretization*, whilst $t' = \{0, \ldots, g/k\}$ represents the *time discretization in persistence framework*, where each step is not made up of only one original time-step but of k ones, and the action a_t is repeated for k times.

In conclusion, it should be noted that if the goal is reached in the middle of a persistence step, *i.e.* in a time-step t between kt' and $k(t'+1)$, the episode ends and the last persistent reward sums $t-kt'$ times, *i.e.* as many times as the number of original time-steps between kt' and the time t when the goal is reached.

5.2. Experimental Results

This chapter will present the results of the new algorithm HPFQI compared with PFQI in the Mountain Car environment. However, before doing so, some necessary notions must be introduced.

[3]It should be noted that the second formulation (5.2) is nothing more than a generalization of the first one (5.1): if in (5.2) we consider $k = 1$, then $t = t'$ and thus (5.1) is retrieved.

5.2.1. Preliminary Considerations

Dataset

In Section 5.1.4, some preliminary theoretical considerations on how to build the initial dataset have already been dealt with. To actually retrieve the initial dataset at a given persistence, we started gathering information from an agent who, beginning from the initial state S_0, follows a random policy until either l_{ep} steps are taken or the goal is reached. This process is repeated for a certain number of times and the union of these results represents the initial dataset $\mathcal{D}$. In our experiments, we always exploited the persistent formulation of the dataset (Section 5.1.4), since working at persistence equal to one with a dataset large enough to have acceptable results proved to be computationally unfeasible. In particular, to generate $\mathcal{D}$ we always considered $l_{ep} = 256$. We thought this value was the best one taking into account both exploration and computational cost: indeed, the agent has enough time to find the goal but without spending too much time in looking for it.

Regressor

Both HPFQI and PFQI need a regressor to give an estimate of the action-value function at each iteration. In the following results, for both algorithms we used the Extra-Trees regressor introduced in Section 2.5.1. We chose it because it had already been used in previous works [29] and because we found out that, in this particular environment, XGBoost generally performs worse.

For each persistence we made a specific tuning of the Extra-Trees hyperparameters, which have been validated on 20 episodes and have been chosen based on their average return. We have considered three hyperparameters:

1. the number of trees (M);
2. the minimum sample size for splitting an internal node (`min_split`);
3. the minimum sample size required by a node to be a leaf node (`min_leaf`).

5.2.2. Results

Persistences and Iterations

Both PFQI and HPFQI, after training, were tested on 20 episodes. As concerns the specific structure of the PFQI and HPFQI models, the persistences chosen are $\mathcal{K} = \{32, 16, 8, 4\}$

and the corresponding iterations are respectively $\mathcal{J} = \{8, 16, 32, 64\}$. We chose them this way in order to be able to compare them better. Indeed, we first must notice that, if the dataset is collected at persistence k, each PFQI and APFQI iteration estimates the target with a precision of k time-steps higher than the previous step. Therefore, for instance, 8 iterations with $k = 32$ look into the future for $32 \cdot 8 = 256$ time-steps. For the way we considered $\mathcal{K}$ and $\mathcal{J}$, each combination, at the end, will estimate the target with a precision up to 256 steps ahead. Notably, the x-axes that we will call *Iteration* in the next plots will represent exactly the product between the actual iteration and the persistence considered.

The further sections are structured as follows: firstly, PFQI results in different shapes are showed, and thereafter HPFQI performances are presented. Afterwords, after comparing the outcomes, some further analysis will be carried out before coming to conclusions.

PFQI

As we previously mentioned in Section 5.2.1, we initially tackled the issue of finding the best hyperparameters for each persistence. Before doing so, for each of them we had to decide the initial dataset length, *i.e.* the number of trajectories performed by a random agent to start from (n_{ep}). This aspect is crucial: indeed, as aforementioned, each Mountain Car episode could be interrupted by reaching either the goal or the allowed threshold of steps. If the former situation occurs, the relative trajectory becomes very important for the agent's later training, because it shows him the right way to reach the goal. On the contrary, the latter situation adds really few information to the future agent.

In light of this, the number of initial episodes has to be carefully chosen so that the PFQI agent has a large enough knowledge base from which to start. However, the choice of n_{ep} cannot be dissociated from the very value of persistence: indeed, we noticed that, the higher the persistence, the higher is the probability that, in a given episode, the random agent reaches the goal. Therefore, n_{ep} has to increase as long as the persistence decreases. Specifically, for the fine-tuning of our hyperparameters, with $k = 32$ and $k = 16$ we considered $n_{ep} = 1000$, whereas for $k = 8$ and $k = 4$ we chose $n_{ep} = 5000$.

We tested all the parameters combinations on 20 episodes each and, from all of them, we retrieved the average return. For each persistence, the combinations with the lowest average return are shown in Table 5.1.

Final Parameters

	k = 32	**k = 16**	**k = 8**	**k = 4**
M	250	200	250	250
`min_split`	5	8	20	30
`min_leaf`	2	2	2	2
n_{ep}	1000	1000	5000	5000
Iterations	8	16	32	64
γ	0.995	0.995	0.995	0.995

Table 5.1: Final parameters for each persistence obtained after Extra-Trees (first batch or rows) and PFQI (second batch of rows) parameters fine-tuning.

Moving on to the actual results, firstly we trained 10 models of PFQI for each persistence with the parameters shown in Table 5.1. The initial dataset of each model has been retrieved following the formulation present in (5.2). Afterwards, each model was tested in 20 episodes and the average return per iteration was obtained. The results are shown in Figure 5.1.

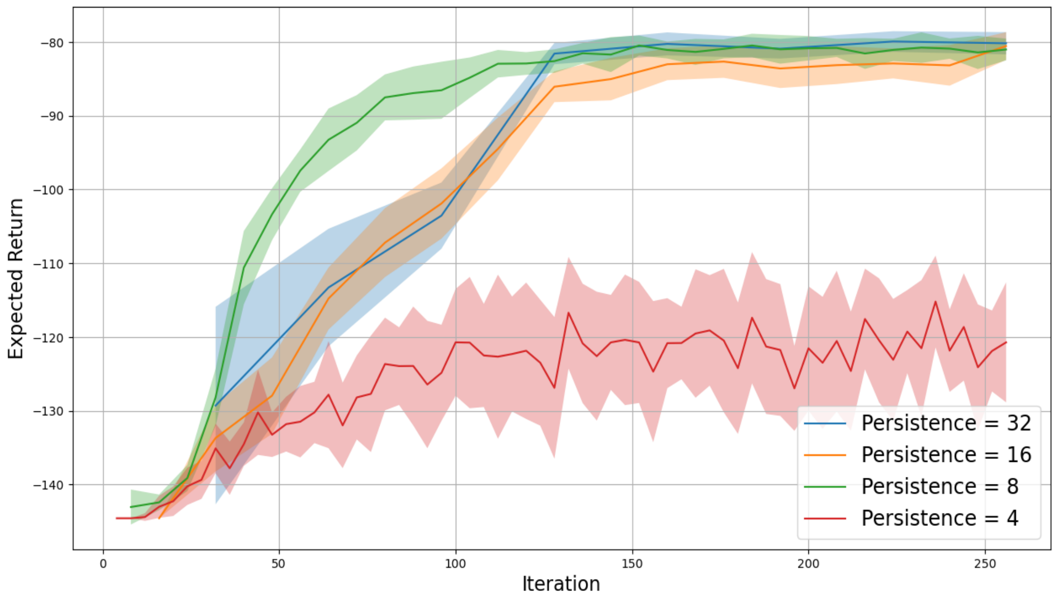


Figure 5.1: PFQI test performance for each persistence. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions and null velocity). For each persistence, ten models with different seeds are used. Therefore, for each persistence, the bold line represents the average expected return among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the PFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2).

Figure 5.1 shows that all the persistences, except for $k = 4$, at their last iteration achieve almost the same expected return. Therefore, for these parameters PFQI already performs well. On the contrary, PFQI with persistence equal to 4 does not perform brilliantly: indeed, it achieves a final expected returns that is just above the worst possible[4]. The main reason for this result lies in the aforementioned problem of the length of the initial dataset. In fact, although for $k = 4$ we considered a high number of episodes, we noticed that with $n_{ep} = 5000$ still there were not enough trajectories reaching the goal. Unfortunately, thinking of raising this value further would make the training computationally unfeasible. In addition, another cause of the negative outcome may rely on the lower value of the persistence itself: indeed, changing frequently the action may cause the agent not to gain enough momentum to easily reach the top of the hill. All in all, this is precisely the issue that HPFQI seeks to address.

[4]The worst possible performance happens when in $l_{ep} = 256$ steps the goal is never reached. In this case, the final return is $\sum_{i=0}^{l_{ep}-1} \gamma^i \cdot R = \sum_{i=0}^{255} 0.995^i \cdot (-1) = -\sum_{i=0}^{255} 0.995^i = \frac{1-0.995^{256}}{1-0.995} \approx -144.57$.

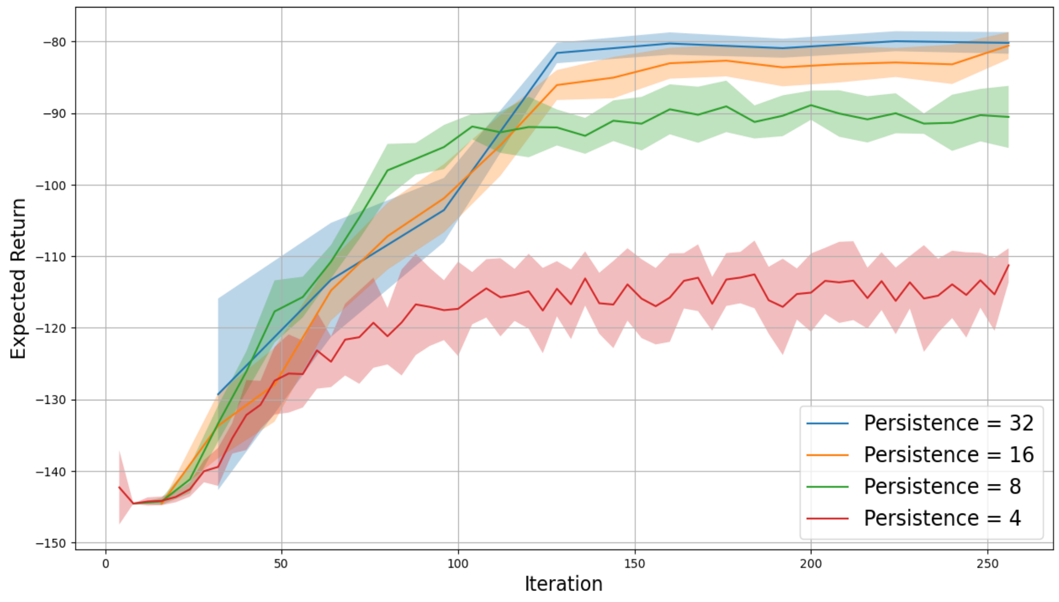


Figure 5.2: PFQI test performance for each persistence with initial datasets of equal length ($n_{eq} = 1000$). Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions and null velocity). For each persistence, ten models with ten different seeds are used. Therefore, for each persistence, the bold line represents the average expected return among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the PFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2).

To confirm the fact that the length of the dataset influences the performance, we trained all models with the same value $n_{eq} = 1000$; the results are shown in Figure 5.2.

As expected, PFQI with the two highest persistences behaves in the same way as before: indeed, in both cases $n_{ep} = 1000$. For what concerns $k = 8$, instead, there is an evident decrease in performance: this result is justified by the presence of fewer trajectories reaching the goal in the initial dataset. Curiously, with persistence equal to 4 it seems that the model works slightly better with a lower dataset dimension. Actually, we believe that this difference could be determined by both the dataset sampling stochasticity and a better adaptation of the parameters to the new dimensionality of the dataset.

HPFQI

For what concerns our novel algorithm, we used the same sequence of persistences and iterations of Section 5.2.2. Thus, HPFQI was trained with $\mathcal{K} = \{32, 16, 8, 4\}$ and, respectively, $\mathcal{J} = \{8, 16, 32, 64\}$. All the other parameters are the same as for PFQI: for

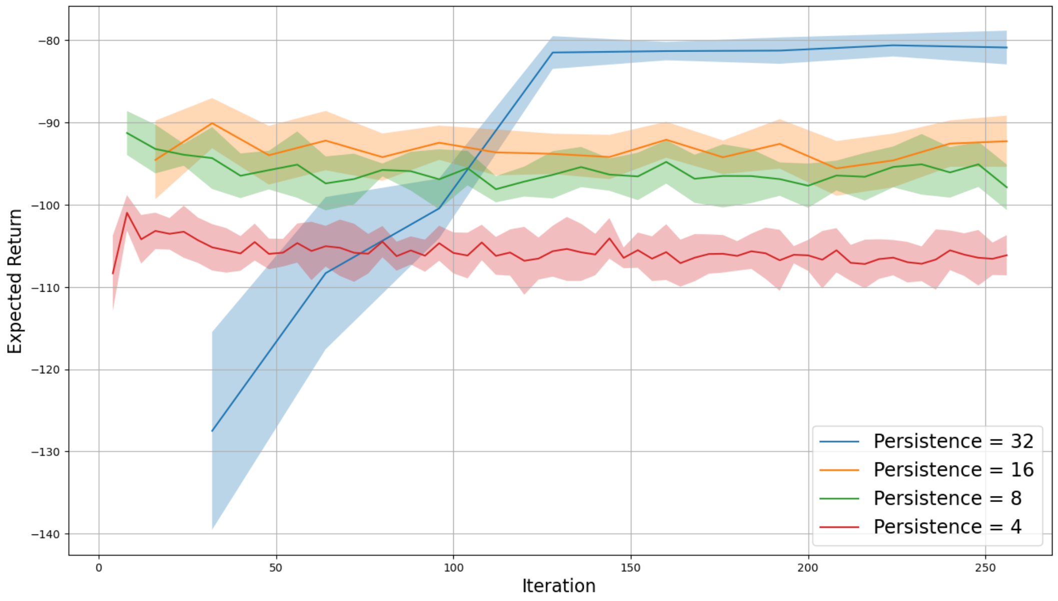
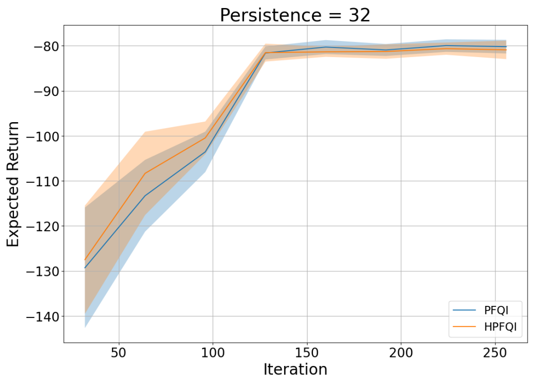


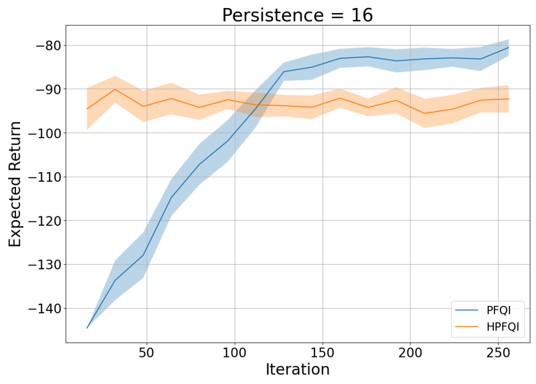
Figure 5.3: HPFQI test performance with $k \in \mathcal{K} = \{32, 16, 8, 4\}$. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions and null velocity). Ten HPFQI models with ten different seeds are used. Therefore, for each persistence, the bold line represents the average expected return of APFQI among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the APFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2).

instance, APFQI at persistence equal to 16 is trained with the same parameters as the parameters in Table 5.1 corresponding to $k = 16$, and so on with all the other persistences. The initial dataset follows the formulation of (5.2) and it is collected at $k = 32$.

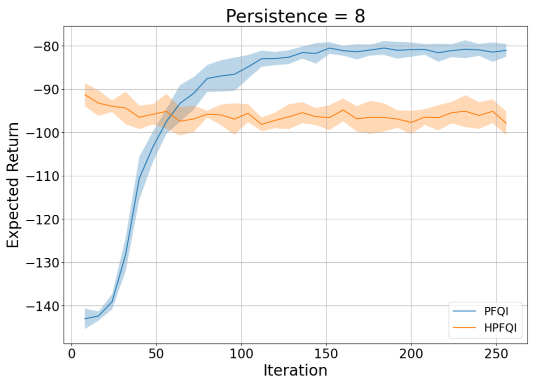
At first glance, Figure 5.3 shows us that, unfortunately, HPFQI performance decreases as the persistences go by. In particular, a significant part of the expected return loss happens exactly in the junction between two consecutive persistences, while the remaining one occurs with increasing iterations. The main reason of the latter probably lies in the so-called *overestimation bias*: in simple terms, it means that, after performing a considerable number of maximum operations, the action-value function estimate is on average higher than the optimal one. As we can see from the figure, this leads to a performance loss. The former, instead, is less immediate, hence it will be analysed better in Section 5.2.3. In order to visualize better the outcome, we compared HPFQI and PFQI for each persistence in Figure 5.4.



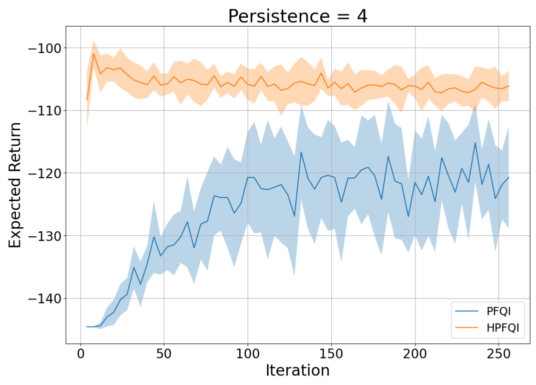
(a) HPFQI (orange) and PFQI (blue) performance at $k = 32$.



(b) HPFQI (orange) and PFQI (blue) performance at $k = 16$.



(c) HPFQI (orange) and PFQI (blue) performance at $k = 8$.



(d) HPFQI (orange) and PFQI (blue) performance at $k = 4$.

Figure 5.4: Comparison between HPFQI and PFQI test performances for each persistence. HPFQI characteristics are the same as in Figure 5.3, whereas PFQI ones come from Figure 5.1.

Figure 5.4a tells us that HPFQI and PFQI at $k = 32$ obtain almost the same final values: this is reasonable, because APFQI at the highest persistence has not any other action-value function to compare with, hence the two algorithms are identical. When we go down to persistences 8 and 4, visible in Figure 5.4b and Figure 5.4c, we notice two main behaviours:

1. in the *initial iterations*, HPFQI reaches a higher expected return than PFQI;
2. in the *final iterations*, PFQI reaches a higher expected return than HPFQI.

The former is reasonable: indeed, PFQI starts learning from scratch at each persistence, while HPFQI immediately exploits the knowledge of the higher persistences. This behaviour highlights that, in HPFQI, information between persistences is passed. Therefore, the theoretical validity of the hierarchical algorithm is confirmed also in real problems. The latter, instead, tells us that if PFQI is given enough time to learn, it will eventually perform better than HPFQI. In Figure 5.4d we see that, instead, at persistence equal to 4 HPFQI always performs better than PFQI. Also there, it is evident that in the hierarchical algorithm actually some information is passed. Furthermore, HPFQI variance with $k = 4$ is lower than PFQI one, hence HPFQI manages to counteract the randomness present at a low persistence. However, the overestimation effect is also evident.

5.2.3. Analysis of the Results

In this section, our purpose is to deeply understand the motivations behind HPFQI results shown in Section 5.2.2.

The first thing that jumps out at you when looking at the Figure 5.3 is that most of HPFQI performance is lost in the handover between two adjacent persistences: indeed, from persistence 32 to 8 there is an average expected return loss of more than 10 and from persistence 8 to 4 almost the same; only from persistence 16 to 8 it keeps approximately the same return. Why does this happen? It is not trivial to give an answer to this question, but let us try to come as close as possible through the following considerations.

First of all, a crucial aspect must be considered: starting the training with persistence equal to 32, the dataset used is in the form (5.2) with, obviously, $k = 32$; for sake of simplicity, we call it $\mathcal{D}_{32}$. Therefore, if we call $\mathcal{D}^*$ the total data collectable at the original time discretization, we notably see that the cardinality of $\mathcal{D}_{32}$ is approximately 1/32 times the cardinality of $\mathcal{D}^*$: indeed, $\mathcal{D}_{32}$ is built precisely taking $\mathcal{D}^*$ and thereafter deleting all the samples that are not multiple of 32. To exemplify, in each episode only the 32^{nd}, the 64^{th} steps and so on until the 256^{th} are considered, while the remaining ones are dropped. Therefore, at the last iteration of APFQI at the highest persistence, the action-value function is estimated based only on $d \in \mathcal{D}_{32}$. However, when it comes to consider the persistence equal to 16, the agent keeps the same action only for 16 steps and then could possibly change it. This change of behaviour entails an increase on the dimension of the state space $\mathcal{S}$: if, with $k = 32$, $\mathcal{S}$ has a certain cardinality, with $k = 16$ it approximately[5] doubles. Thus, the fitting of the action-value function imported from the previous persistence was only performed on half of the current states, while on the others its value is simply estimated by fitting. Therefore, if the fitting of Extra-Trees is not perfect, a large bias is inevitably dragged in.

[5]It cannot be said that the cardinality exactly doubles because of both some possible repetition of states and the fact that the episodes in which the goal is reached are not all equally long.

The problem of bringing the action-value function to the lower persistence is highlighted by Figure 5.5. Indeed, although the action-value function at the subsequent persistence seems to have slightly higher values with respect to the previous one, in all three cases (passage from $k = 32$ to $k = 16$, from $k = 16$ to $k = 8$, from $k = 8$ to $k = 4$) there is a decrease in the proportion of states which see the goal. This decrease is caused mainly by the fact that the states that are added are almost always intermediate states, not necessarily close to the goal. Let us try to justify this better. We consider, without loss of generality, that we are moving from persistence 32 to persistence 16. Furthermore, since in the vast majority of the trajectories the goal is not reached, we study one of this kind. In this setting, the states already visited at the previous persistence are those corresponding to $t \in \{32, 64, 96, \dots, 256\}$. Hence, the new states encountered correspond to $t \in \{16, 48, 80, \dots, 240\}$. Even if we assume that at the last iteration the agent came very close to the target, which is far from obvious, it is clear that most of the new states are far from the target. Therefore, despite the fact that this is caused by the way the problem itself is constructed, the transport of the action-value function is tainted.

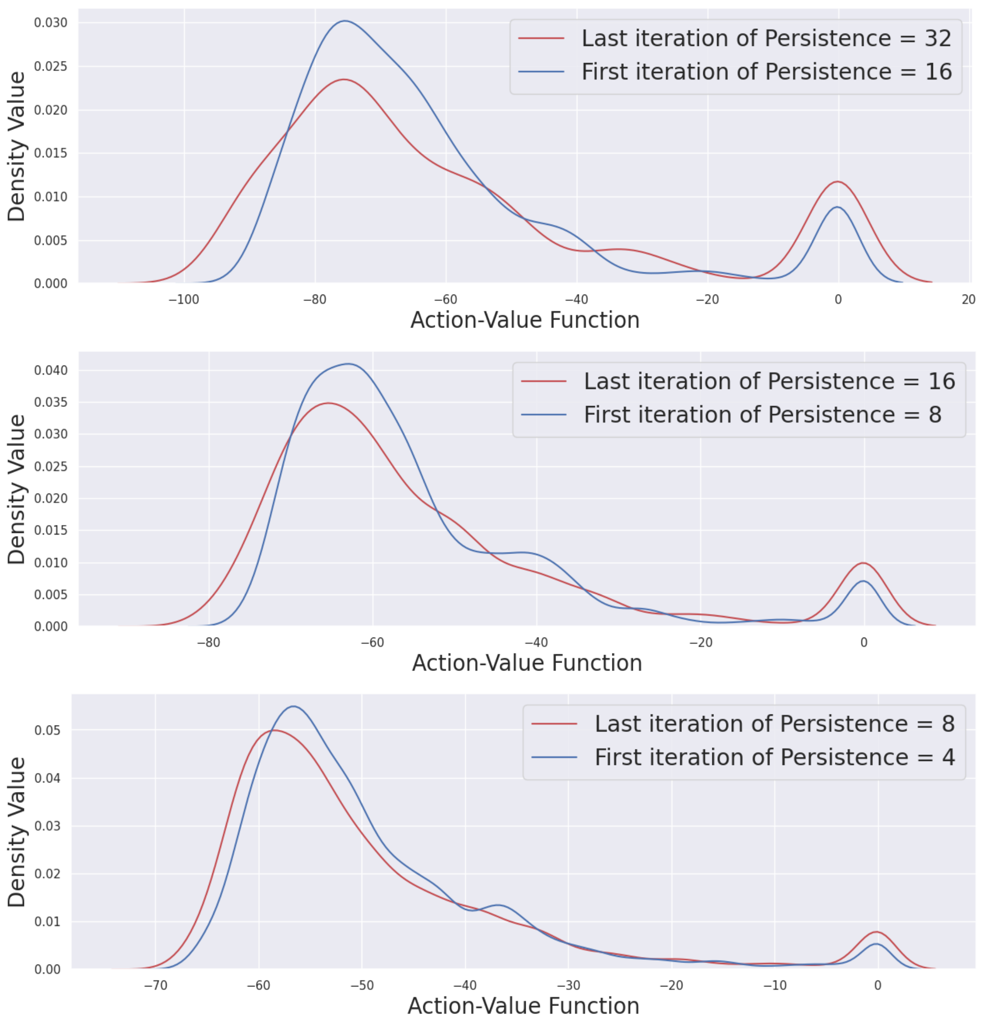
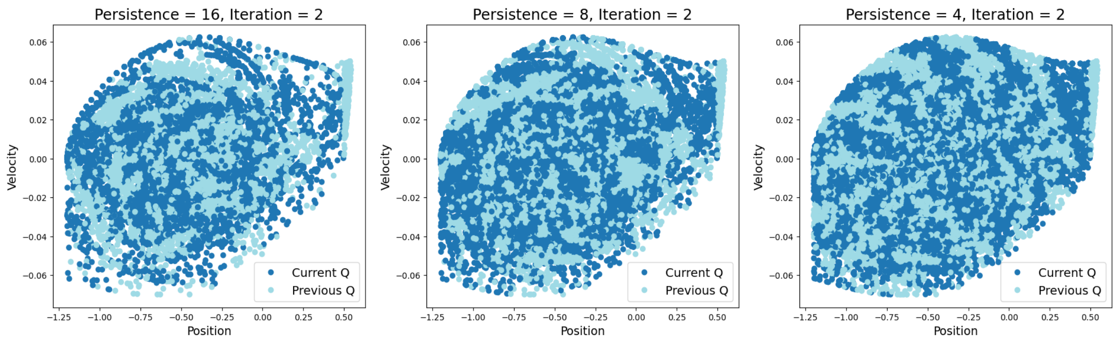


Figure 5.5: Kernel density representation of the action-value function at some iterations of HPFQI.

Furthermore, let us consider, for each persistence lower than 32, HPFQI at the first iteration in which the two action-value functions (the one derived by the previous persistence and the other generated at the current persistence) are compared, *i.e.* at the second one (see Section 4.2). In this cases, it is interesting to notice the way the states divide based on which function is higher between the two. As it is visible in Figure 5.6, this shows a curious pattern: in the two-dimensional representation of all the states the agent encounters, there are very distinct bands of states in which the maximum is reached by the previous action-value function and others in which the opposite happens. In our view, this is caused by both the different cardinality of the state space and the environment itself.



(a) (b) (c)

Figure 5.6: States visited by the agent during HPFQI training at the second iteration of different persistences. The second iteration is chosen because it is the first iteration in which the target compares the action-value function generated by itself with the action-value function obtained at the previous persistence. The different colors highlight whether in the state the maximum has been reached by the previous action-value function (light-blue) or by the current one (blue).

The former issue can be analysed more in detail through Figure 5.7. These graphics show that, on the new states of a given persistence, the action-value functions derived from previous persistences less frequently assume values higher than the action-value function generated at that hierarchical step. In other words, it is more likely that an old state exploits the old action-value functions than a new one. This confirms that the previous action-value functions difficultly adapt to unseen states, causing the aforementioned loss of performance between two consecutive persistences.

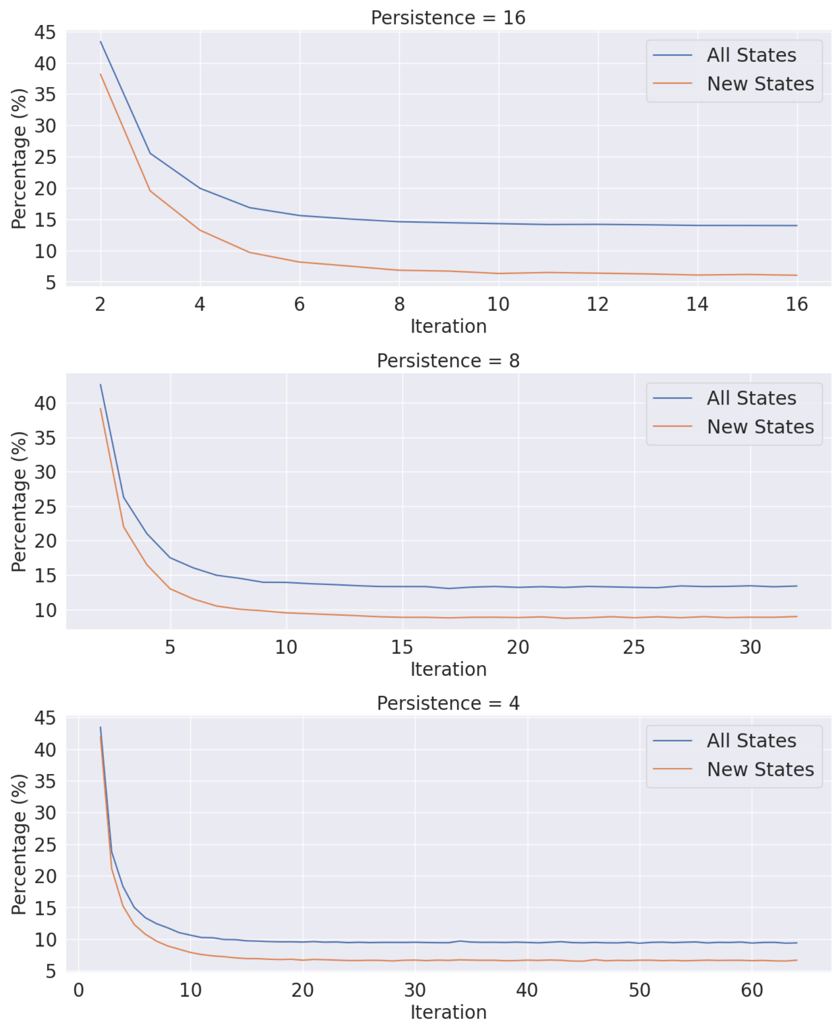


Figure 5.7: Percentage use of the action-value function of previous persistences in HPFQI. The percentage over all the states is displayed in blue, whereas the percentage over the new states only is represented in orange.

Figure 5.7 shows also another aspect: as the number of iterations increases, the percentage of states choosing the action-value function of previous persistences decreases. This could seem a negative behaviour, but instead it proves that the algorithm is working: indeed, as iterations increase, the action-value function generated by APFQI itself (*i.e.* not the one imported from previous persistences) is increasingly influenced by the action-value function of higher persistences. This happens for the following reason: at the second iteration of a certain persistence, *i.e.* at the first in which the two action-value functions are compared, in some states the maximum is reached by the one deriving from the previous persistences, while in the remaining states the maximum is reached by the current one. At the following iteration, the same process happens. However, this time the action-value function generated at the current persistence is no more *pure*, meaning that it has been fitted exploiting also the states that at the previous iteration chose the action-value function deriving from the higher persistences. Therefore, at a high iteration, the new action-value function actually contains information that is also derived from the action-value functions of other persistences.

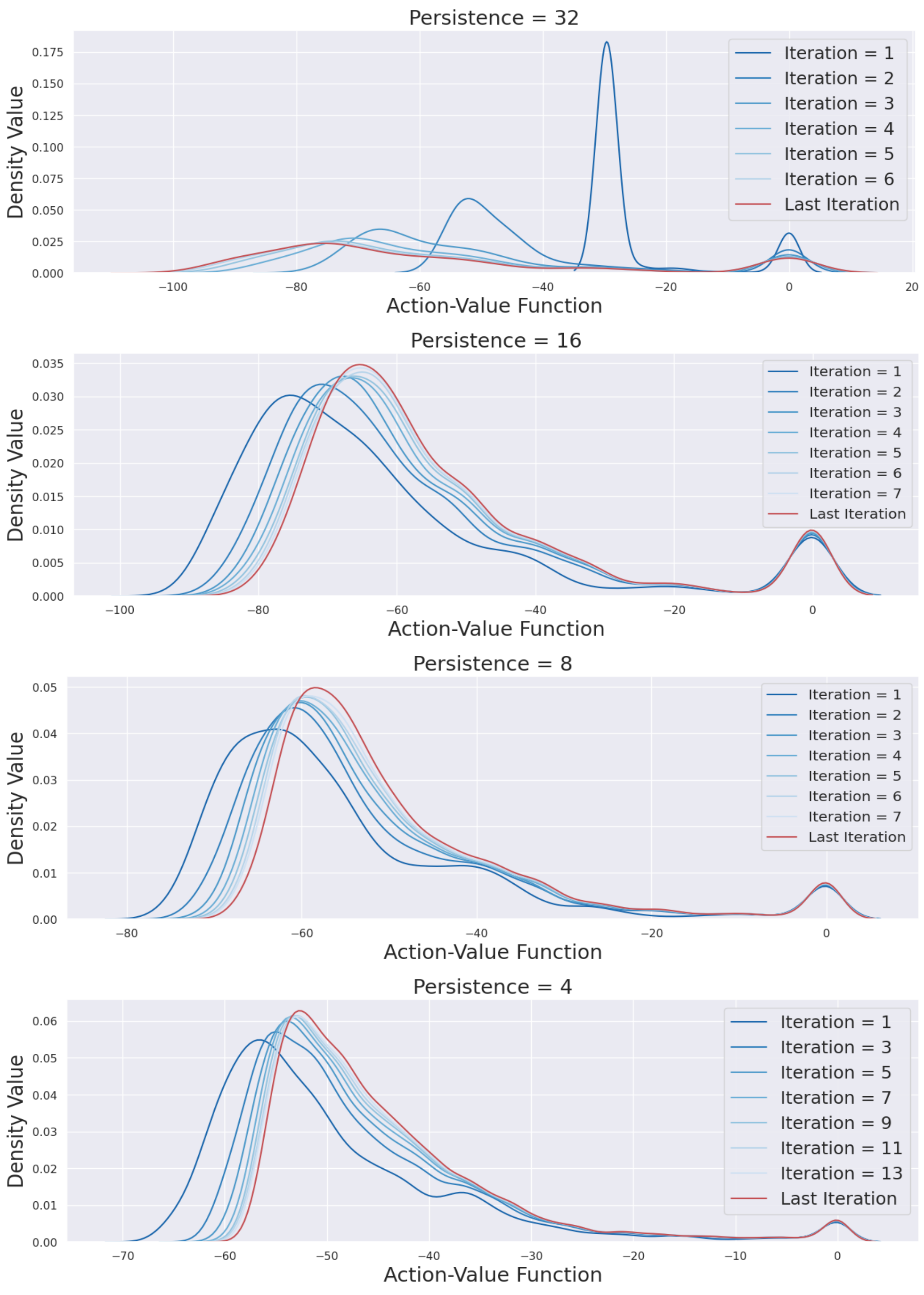


Figure 5.8: Kernel density representation of the action-value function of HPFQI.

This behaviour along the iterations of a persistence is shown in more detail in Figure 5.8.

In fact, in all the persistences in which the hierarchical structure is exploited, the action-value function increases its values and, especially, it slightly increases the number of states with a value close or equal to 0. While the latter shows a behaviour not to be dismissed, the former, instead, probably highlights a slight manifestation to the age-old problem of overestimation typical of FQI-derived methods.

Figure 5.9 is somehow a *summa* of several previous conclusions. Indeed, a number of things can be inferred: as persistence decreases, the presence of values of the action-value function close to 0 decreases, *i.e.* the top of the hill is less reached. However, as persistence decreases, the values assumed by the action-value function on average both increase and accumulate in a central area.

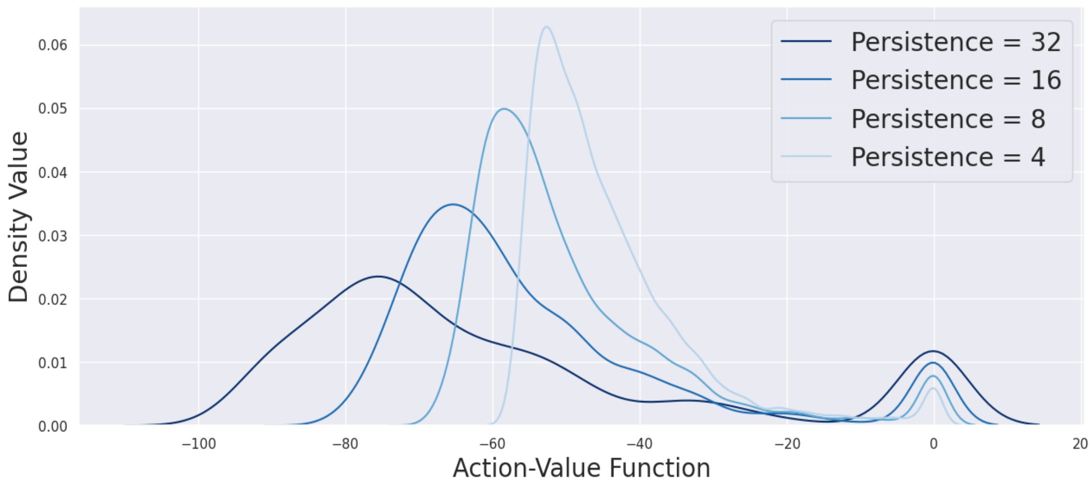


Figure 5.9: Kernel density representation of the action-value function at the last iteration of each persistence of HPFQI.

5.2.4. Randomization of the Initial Velocity

In order to try to remedy the problems encountered in the previous section, we did the same experiments of Section 5.2.2 and Section 5.2.2 but with a little difference: in the classical Mountain Car setting, as aforementioned in Section 5.1.3, only the initial position of the initial dataset is random, whereas the initial velocity is always set equal to 0. Therefore, the initial state of all the collected trajectories belongs to an hyperplane $\mathcal{S}_{init} \subset \mathcal{S}$. In this section, we consider both position and velocity random, uniformly sampled in their respective domains (Section 5.1.3). Therefore, the initial position belongs to the whole state space $\mathcal{S}_{init} \equiv \mathcal{S}$. In this way, being the trajectories in the dataset more diverse, the algorithms should explore more.

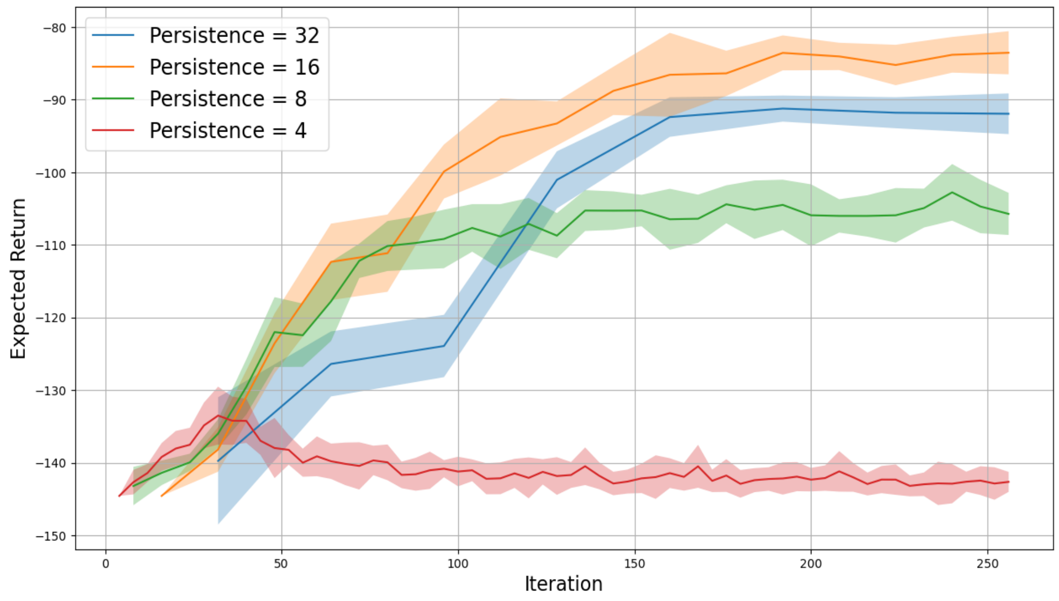


Figure 5.10: PFQI test performance for each persistence with initial datasets of equal length ($n_{eq} = 1000$) and random both initial position and velocity. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes. For each persistence, ten models with different seeds are used. Therefore, for each persistence, the bold line represents the average expected return among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the PFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2).

As it is displayed in Figure 5.10, the learning patterns are less regular than in the previous case. Indeed, PFQI with $k = 16$ performs even better than with $k = 32$. In addition, with persistence equal to 4, the algorithm starts learning something that forgets at higher iteration steps. Probably, this happens because with a much more random dataset the errors occurring at the first iterations are so high that they invalidate completely the learning process.

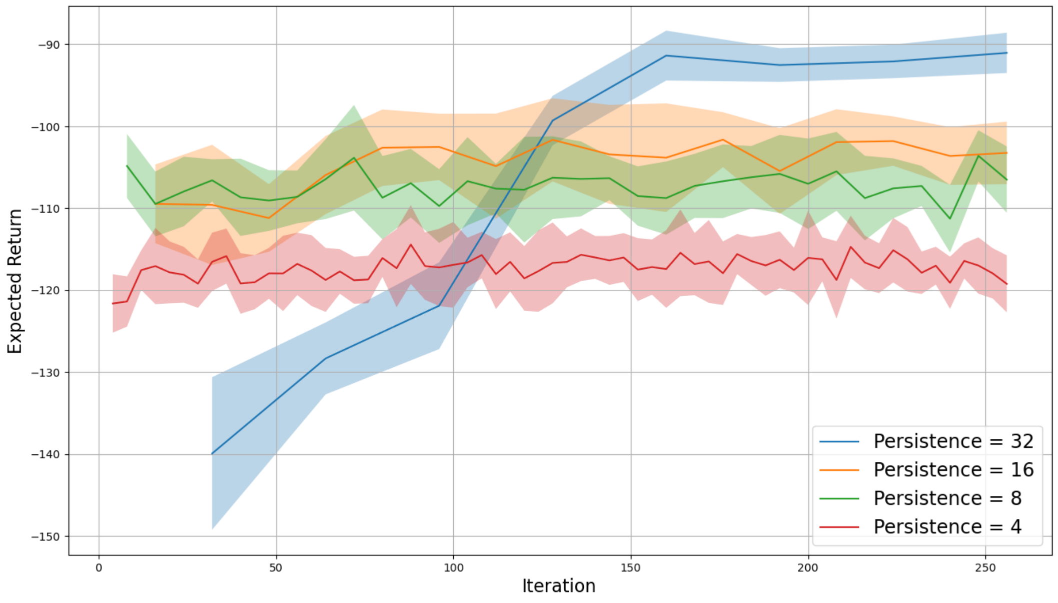
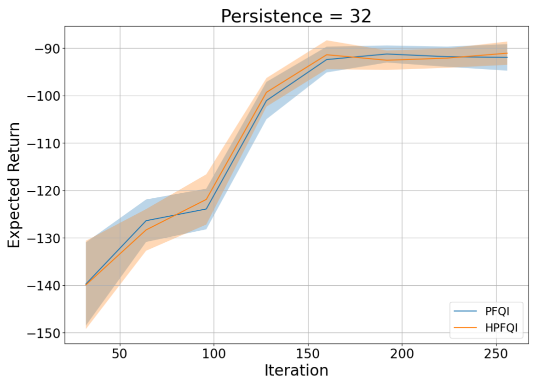
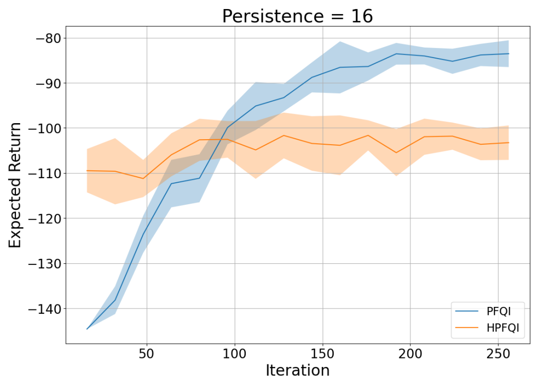


Figure 5.11: HPFQI test performance with $k \in \mathcal{K} = \{32, 16, 8, 4\}$. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions). Ten HPFQI models with ten different seeds are used. Therefore, for each persistence, the bold line represents the average expected return of APFQI among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the APFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2).

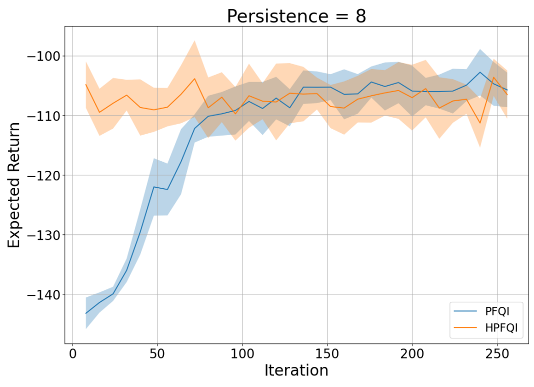
Instead, HPFQI seems to be more stable: indeed, Figure 5.11 is nothing but a downward translation of Figure 5.3. This shift derives only by the fact that, this time, APFQI at the highest persistence performs worse than with non random initial velocity; notably, this worsening equally affects all other persistences. Thus, beyond that the overall outcome structure is unaltered. Therefore, we note that HPFQI shifts the behaviour of higher persistences to lower ones, eliminating the noise of the latters.



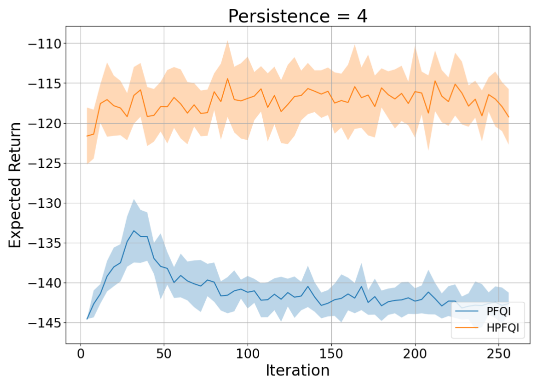
(a) HPFQI (orange) and PFQI (blue) performance at $k = 32$.



(b) HPFQI (orange) and PFQI (blue) performance at $k = 16$.



(c) HPFQI (orange) and PFQI (blue) performance at $k = 8$.



(d) HPFQI (orange) and PFQI (blue) performance at $k = 4$.

Figure 5.12: Comparison between HPFQI and PFQI test performances for each persistence with random both initial position and velocity. HPFQI characteristics are the same as in Figure 5.11, whereas PFQI ones come from Figure 5.10.

All the previous considerations are confirmed by Figure 5.12, where we notice that the patterns are quite similar to Figure 5.4. The only main difference relies on Figure 5.12c, where PFQI eventually reaches HPFQI performances but does not exceed them.

5.2.5. Conclusions

All in all, the general behaviour of the HPFQI algorithm in the classical Mountain Car environment, as can be inferred by Figure 5.9 and all the other graphics displayed in the previous sections, is the following:

1. between one persistence and the next, there is a sudden loss of performance, probably caused mainly by the sudden increase in states not seen by the previous action-

value function;

2. as persistence decreases, the presence of values of the action-value function close to 0 decreases, *i.e.* the top of the hill is less reached. This is probably due to both a proportional decrease in the states close to the goal and the difficult adaptation of the action-value function to a great amount of new states;

3. more in general, as persistence decreases, the values assumed by the action-value function on average both increase and accumulate in a central area: this probably happens due to both the overestimation of the action-value function along the iterations and an increase in the number states in the middle of the trajectories of the higher persistences;

4. HPFQI shows that the information gathered by the action-value function at the highest persistences is passed to and exploited by the lowest ones, as highlighted by the better performance in the first iterations with respect to PFQI (Figure 5.4 and Figure 5.12);

5. due to its hierarchical structure, HPFQI is more robust against typical high-frequency noise than PFQI (Figure 5.4 and Figure 5.12).

The Section 5.2.4, instead, on one hand showed us the stability of the hierarchical algorithm compared to the higher variability of PFQI. On the other hand, as we could not see any improvement in the performance of HPFQI after the change made in the initial state, we cannot conclude that the problem of poor performance is simply due to the change in the number of states.

Although these results were not too positive in the whole, we are convinced that the majority of them is caused by this specific environment and not by the algorithm itself. Indeed, in Mountain Car, the main problem was the lack of exploration of the new states during the persistence-step. Fortunately, in the context of the FX market such problem will not arise. The latter environment, in fact, is way more stochastic, making it much more unlikely that the action-value function focuses too much on certain states, thus failing to generalise to other new ones. In addition, the construction of the dataset itself will be done differently. Firstly, in Mountain Car we had to exploit the formulation (5.2) that forced us to lose all states present between steps of the agent at a certain persistence. This does not happen in FX because, even if we work with a persistence equal to k, we can gather nine different trajectories for each original time-step. In this way, the agent will explore the state space much more and more independently than before. Secondly, the reward function will represent the actual profit/loss at a given step, helping the agent to understand in more detail how best to behave. This does not happen in Mountain Car, where the presence of a unique goal flattens the information extractable by it. Despite this, we believe that a more in-depth analysis of HPFQI in the Mountain Car environment is of considerable interest and worthwhile for future works.

6 | Problem Formulation FX

6.1. General Overview

6.1.1. FX Market

Forex, or Foreign Exchange (FX), is the largest financial market in the world where currencies are traded. It is a decentralized market that operates 24 hours a day, 5 days a week, and it is made up of a network of banks, financial institutions, and individual traders who buy and sell currencies from all around the world. The FX market allows businesses and individuals to exchange one currency for another for various purposes, such as conducting international trade or for investment purposes. Currency values are constantly changing due to various factors such as economic and political events, and this creates opportunities for traders to profit by buying or selling currencies based on their predictions of how the currency values will change in the future. While FX trading offers potential for high profits, it also involves a high level of risk and requires a solid understanding of the market and trading strategies.

6.1.2. EUR/USD Market

The EUR/USD FX market is one of the most actively traded currency pairs in the world. It represents the exchange rate between the Euro, the currency of the European Union, and the US dollar, the currency of the United States. The EUR/USD market offers traders opportunities for both short-term and long-term trading strategies. However, as with all FX trading, there is also a high level of risk involved in trading the EUR/USD pair.

There are several reasons why the EUR/USD FX market is more tradable than other currency pairs:

- *high liquidity*: the EUR/USD market is one of the most liquid markets in the world, meaning there are a large number of buyers and sellers trading the currency pair at any given time. This high liquidity provides traders with more opportunities to enter and exit trades at the desired price levels;

- *low spreads*: the bid-ask spread, which is the difference between the price at which buyers are willing to buy and the price at which sellers are willing to sell, is typically lower for the EUR/USD pair compared to other currency pairs. This means that traders can buy and sell the currency pair at a lower cost, which can help increase profits;
- *high volatility*: the EUR/USD market is known for its high volatility, meaning there can be significant price movements in a short period of time. This volatility can provide traders with opportunities to make large profits in a short amount of time;
- *economic stability*: the Eurozone and the United States are both major economic powers and have stable political systems, making them less prone to sudden and unexpected economic or political events. This stability can make the EUR/USD market less risky for traders compared to other currency pairs where the economies or political systems may be more volatile or unstable;
- *availability of information*: the EUR/USD market is widely covered in the financial news media, and there is a wealth of information available about the factors that affect the currency pair. This information can help traders make more informed trading decisions.

6.1.3. Reinforcement Learning in FX

In general, the FX market can be a good choice for Reinforcement Learning algorithms for several reasons:

- *large amounts of data*: the FX market generates a large amount of trading data, which can be used to train, validate and test Reinforcement Learning algorithms. This data can help the algorithm learn patterns and make more accurate predictions about future price movements;
- *high liquidity*: as previously said, the FX market is one of the most liquid markets in the world. This high liquidity provides a good environment for Reinforcement Learning algorithms to operate and trade without significant impact on the market prices;
- *high volatility*: the FX market is known for its high volatility, meaning there can be significant price movements in a short period of time. This volatility can provide Reinforcement Learning algorithms with more opportunities to learn and adjust

their strategies;

- *24/5 market availability*: the FX market is open 24 hours a day, 5 days a week, providing ample opportunity for Reinforcement Learning algorithms to operate and learn;
- *low transaction costs*: the transaction costs associated with FX trading are generally lower than other financial markets, such as equities or futures. This can make it more cost-effective for Reinforcement Learning algorithms to trade in the FX market.

6.1.4. Market Data

The original dataset is made of market observations collected every minute from Monday to Friday for the EUR/USD currency pair. In addition to the corresponding *date* and *time*, each observation includes the quoted exchange rate, or *mid price*, and the value of the *bid-ask spread*, or simply spread, defined as the difference between the bid price and the ask price. In general, the bid price refers to the highest price a buyer will pay for a security, whereas the ask price refers to the lowest price a seller will accept for a security. Notably, the mid price is not a price itself. Indeed, it is just a quotation of the intrinsic value of the security, being it the average between the bid price and the ask price. From a financial perspective, the spread is an indicator of the liquidity of the currency pair; as aforementioned, the smaller the spread, the more liquid the market is.

6.2. FX Market Trading as an MDP

6.2.1. FX as an Episodic Task

The EUR/USD FX market trading could be represented as a unique MDP, where the training set would be a unique episode from start to finish. However, we decided to divide it in daily episodes for the following motivations:

- *cost-effectiveness*: closing all the positions before the end of the trading day is usually more convenient from a financial point of view. In fact, not only it is less risky, but also cheaper since, in addition to the usual transaction fees, most of the FX brokers charges investors for overnight fees if they hold a position overnight;
- *undiscounted setting*: from a reinforcement learning point of view, considering short episodes allows us to simplify working in the undiscounted setting, *i.e.* with $\gamma = 1$;
- *parallelization*: working on different episodes allows us to parallelize the training of

the algorithm, reducing sensibly the computational costs.

One small change we feel we must make concerns the number of actual hours considered within each individual episode: in order to train our algorithm, we considered only the band from 8:00 CET to 18:00 CET. We decided so because in that period of time the market is liquid enough and matches European working hours, making our approach more consistent and robust.

6.2.2. Actions

The definition of the actions the agent can take is straightforward: at each minute of the episode (or at each multiple of it, if we consider a persistence greater than one), the agent can choose between being *Short*, *Long* or *Flat*. If the agent, at time t, decides to have a *Short* position, *i.e.* it sells, it means that it hopes that in $t+1$ the value of the exchange rate decreases. On the contrary, if the agent in t is *Long*, *i.e.* it buys, it makes a profit when the value of the exchange rate increases. Finally, being *Flat* in t means that between t and $t+1$ no position is held by the agent. For notational convenience, if the agent sells the action is marked as -1, if it is flat the action is marked as 0 and if it buys the action is marked as 1.

Note that in our model we assume that the agent can only deal with a fixed quantity of asset, thus the position must always be -1, 0 or 1 and not something in between. In addition, in order to make the setting more realistic, at the end of each episode the agent is forced to close its position, *i.e.* to be *Flat*.

Therefore, the action space $\mathcal{A}$ is defined as:

$$\mathcal{A}(s) = \begin{cases} \{0\} & \text{if } s \in \mathcal{S}^{term} \\ \{-1, 0, 1\} & \text{otherwise} \end{cases} \tag{6.1}$$

where S^{term} represents the set of terminal states.

6.2.3. State

First of all, it must be noted that, in our setting, the basic time step t is the minute. Therefore, $t+1$ stands for the minute after t, $t+2$ for the minute after $t+1$ and so on. After having performed some preliminary analyses, we decided to include in the state the following features (for clarity, we consider the state evaluated at a certain time t):

- the current *minute* of the day (*i.e.* t);

- the current *day of the week*. The days are associated, in increasing order, to a natural number: for example, Monday is associated to 1 and Friday to 5;
- the last 60 *normalized exchange rate differences* between consecutive minutes, defined as:
$$d_t^k = \frac{p_{t-k+1} - p_{t-k}}{p_{t-k}} \quad \text{for } k = 1, 2, \dots, 60\,, \tag{6.2}$$
where p_t stands for the exchange rate at time t;
- the current *spread* σ_t;
- the current *portfolio position* with respect to the currency pair x_t.

Notice that in (6.2) we consider the difference between two consecutive exchange rates both for stationary issues and for ensuring a greater generalization during the training phase. In addition, we also normalize for p_{t-k} for similar reasons. The current portfolio position x_t, instead, simply represents the action taken by the agent at the previous step, hence $x_t = A_{t-1}$.

6.2.4. Reward

The reward function we are going to use is defined as follows:

$$R_{t+1} = A_t(p_{t+1} - p_t) - c_t|A_t - x_t|\,, \tag{6.3}$$

where
$$c_t = f + \frac{1}{2}\sigma_t\,. \tag{6.4}$$

In the previous equations, A_t stands for the action taken by the agent at minute t, p_t is the mid-price at minute t, x_t represents the portfolio position held at time t, c_t is cost associated to the allocation change, σ_t is the spread at minute t and, lastly, f stands for the fixed transaction fee.

The definition (6.3) is composed of two parts: the first one represents the pure profit/loss in which the agent incurs taking action A_t, whereas the second one indicates the costs associated to that transaction. In particular, as can be deducted by definition (6.4), also the cost function is made up of two terms: half of the current spread, which varies based on t, and a fixed fee by the fact of making a transaction itself. the former assumes that value because, actually, trading orders are never executed at the mid price: if an investor wants to buy any financial asset, he has to pay the corresponding ask price; on the other hand, if he wants to take a short position, he has to sell the asset at the bid price. Since the mid price is defined as the midpoint between the bid and ask prices, then the difference between the quoted price and the order price is exactly half the value of spread. To sum up, the definition (6.3) means that the reward obtained by the agent depends on the instantaneous profit/loss from which, only if the position is changed, is subtracted a certain value. For simplicity, in our work we set $f = 0$.

6.2.5. Dataset

At this stage we are ready to construct the dataset $\mathcal{D}$ structure that will be exploited by our algorithm. When $k = 1$, the dataset definition of a single episode is the same as (5.1), *i.e.*:

$$\mathcal{D}_{ep} = \{(s_t, a_t, r_{t+1}, s_{t+1})\}_{t=0}^{g-1}\,, \tag{6.5}$$

where g identifies the number of minutes for each episode, which in our case is 600.

When $k > 1$, instead, the dataset formulation is a bit different:

$$\mathcal{D}_{ep}{}^{1} = \left\{\left(s_t, a_t, \sum_{i=1}^{k} r_{t+i}, s_{t+k}\right)\right\}_{t=0}^{g-k-1}. \tag{6.6}$$

Unlike Mountain Car setting, we recall that here we consider $\gamma = 1$. Furthermore, one difference on the sampling process itself must be highlighted: in Mountain Car, we collected a sample every k time-steps, having at the end g/k tuples for each episode. In that case, we were forced to do this because, as we had to sample the states directly on-policy, we could only collect data from the states we passed. In FX, instead, we can collect a nine different samples for every time step regardless of the persistence considered: in this way, $|\mathcal{D}_{ep}| = 9(g - k)$. The main reason behind it is that in FX we rely on historical data. Therefore, on top of them we are able to recreate different situations: in fact, at each time-step we can create nine different situations, depending on what type of x_t and a_t we consider[2]. Moreover, given that the possible (x_t, a_t) combinations are only nine, at each time-step HPFQI will be able to retrieve all the possible policies in the dataset. In Mountain Car this was not possible, because we did not have historical data to rely on. Actually, the choice of the persistence influences the collection of the final data of an episode: indeed, with persistence k we must stop creating tuples k steps before g, because,

[1]Actually, in FX case each tuple represents 9 tuples, one for each $(x_t,\, a_t)$ combination.

[2]Indeed, given that $x_t \in \mathcal{A}$ and $x_t \in \mathcal{A}$, the total combinations are $|\mathcal{A} \times \mathcal{A}| = 9$

if we went ahead, we would need a state $s_{t+k} = s_{(g-k+1)+k} = s_{g+1}$ that is impossible to retrieve.

6.3. Data Exploration

Before going to the core of the results of our algorithm, we decided to perform an exploratory analysis of our data to see if we can extract any useful information or if there is something wrong in them.

To train our model, we chose the EUR/USD data of 2018 and 2019 sampled to the minute. For simplicity, we downloaded the time series from HistData[3], that already processed them by converting them from the tick to the minute. In addition, another advantage of this platform is that there are less missing values, crucial aspect for the proper training of our algorithm.

6.3.1. Stationarity

Our first analysis focused on verifying one basic property of the time series: the *stationarity*. Roughly, one time series is stationary if over the time its mean is equal to 0 and its variance is constant. Surely, the mid-price time series itself is not stationary. However, we are not concerned about the stationarity of prices since, in its state, our algorithm uses the normalized price differences (as defined in Equation (6.2)). Therefore, our goal is to test the stationarity of said normalized price differences d_t^k.

We did so exploiting the concept of *autocorrelation* which is estimated as:

$$\rho(k) = \frac{1}{nS^2} \sum_{t=1}^{n-k} (X_t - \mu)(X_{t+k} - \mu)\,, \tag{6.7}$$

where $\{X_1, X_2, \dots, X_n\}$ is a discrete process (*i.e.* an ordered series of the same random variable evaluated in consequential steps), n is the number of observations, S^2 is an estimate of the variance of the process and μ is an estimate of the mean of the process. The term k, instead, is called *lag* and it is a positive integer which represents the order of autocorrelation we are interested in: for instance, $\rho(1)$ stands for the autocorrelation between the process itself and the process translated by one time step. In light of this, in a process, the autocorrelation function tells how much the variables, or observations, at distance k from each others are correlated. This function is useful because, if $\rho(k)$ with

[3]https://www.histdata.com/

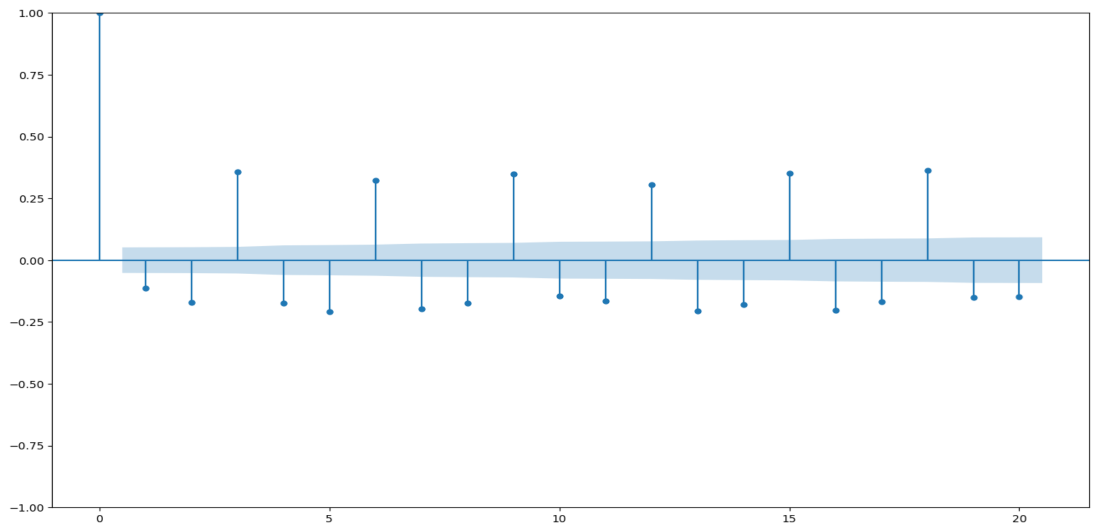


Figure 6.1: Autocorrelation function over the process defined in (6.8), considering HistData data of 2018 and 2019. In the x-axis we find the lag value and on the y-axis the value of the autocorrelation itself.

$k \geq 1$ is large[4], probably the process is non-stationary[5].

Hence, we computed the autocorrelation of the time series $\{X_1, X_2, \dots, X_d\}$, where the random variable X_t is defined as:

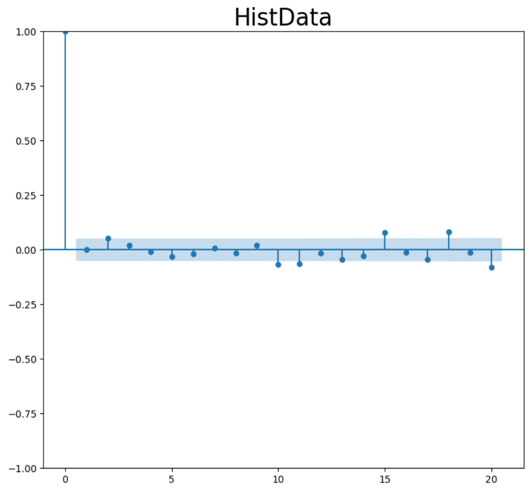
$$X_t \coloneqq \frac{1}{d}\sum_{i=1}^{d}(p_{t+1,i} - p_{t,i}), \tag{6.8}$$

where d indicates the number of days present in the dataset and $p_{t,i}$ stands for the exchange rate value at t-th minute of day i. Thus, basically, X_t represents the average over each day of the difference between the $t+1$-th mid-price and the previous one. The result is shown in Figure 6.1.

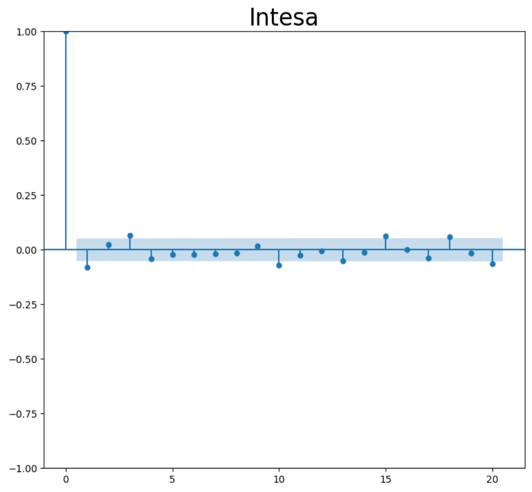
Unfortunately, Figure 6.1 shows a repetitive pattern: indeed, it seems that the value of X_t is significantly correlated positively with $X_{t+3}, X_{t+6}, X_{t+9} \dots$ and negatively with $X_{t+1}, X_{t+2}, X_{t+4}, X_{t+5} \dots$ This is really strange, because if this were the case, it would be possible to easily predict the future value of the exchange rate just by knowing the current one. For this reason, we fear that there may be some problems in the data itself.

Therefore, we decided to gather data from another source to compare them. In particular, we managed to get the data collected by another company, Intesa SanPaolo, over the

[4]We do not consider $\rho(0)$ since one can easily see that always $\rho(0) = 1$.
[5]Actually, this conclusion does not always hold. However, empirically we can rely on it.



(a)



(b)

Figure 6.2: The first 20 lags of the autocorrelation function over the random variables defined in (6.8) for HistData (left) and Intesa (right) data over dates approximately from April 2022 to December 2022.

EUR/USD rates from April 2022 to December 2022 approximately. We then downloaded the corresponding data from Histdata in order to compare them. The autocorrelations of the two are shown in Figure 6.2.

From Figure 6.2 one fact immediately catches the eye: the autocorrelation values of both HistData and Intesa in 2022 are dramatically lower than the one showed in Figure 6.1.

To understand the reason for this, we computed also the autocorrelation function of HistData with all the data it has during that specific period in 2022; the results are in Figure 6.3.

Figure 6.3 shows a more pronounced pattern than Figure 6.2. In addition, the pattern develops almost in the same way as the one of Figure 6.1. Indeed, we found out that the data coming from Intesa have a higher number of missing values. Therefore, we think that the less pronounced patterns in Figure 6.2 are simply due to less data. This would justify the different behaviour present in Figure 6.3 and also the one of Figure 6.1. Anyway, we are unable to use the Intesa data since too many missing values would create major problems to the training process. Therefore, we decided to check more carefully whether the autocorrelation pattern of HistData could actually affect our agent or not.

In order to do so, we introduce a new definition:

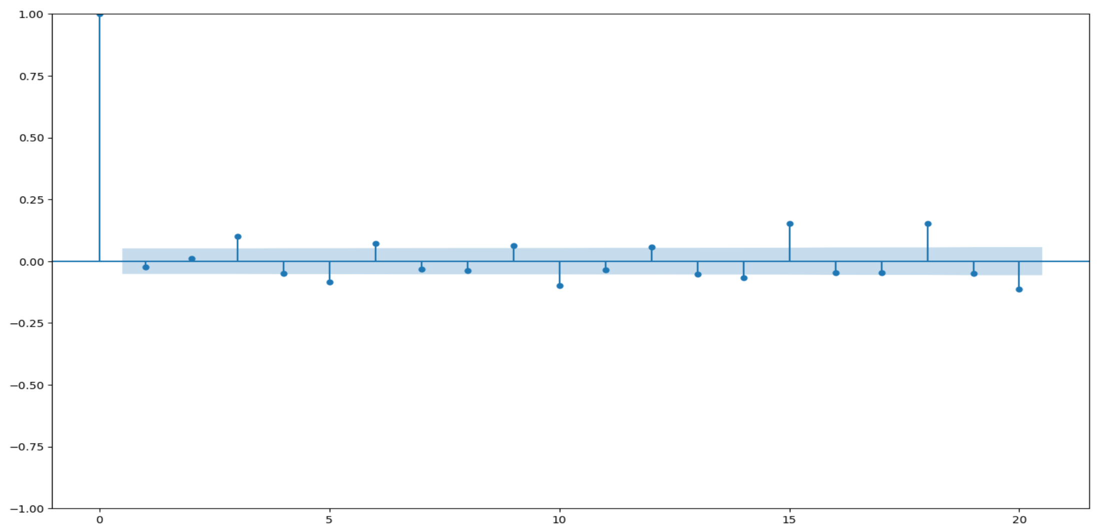


Figure 6.3: Autocorrelation function over the process defined in (6.8) with the total amount of HistData data over a specific period of 2022. In the x-axis we find the lag value and on the y-axis the value of the autocorrelation itself.

$$K(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \mu)(X_{t+k} - \mu)\,, \tag{6.9}$$

which is an estimate of the *autocovariance* function. We introduce it for the following reason: if $\sqrt{|K(k)|}$ is less than half of the average bid-ask spread of our data $\forall k$, we can assess that the autocorrelation pattern does not influence our algorithm. Why can we say that? Let us go back to why we did this analysis, namely that we did not want our agent to learn spurious signals from the dataset. But when do these spurious signals actually become relevant enough to affect an RL agent's learning? This happens when the reward function is significantly altered by them. In our case, as we can see from Equation (6.3), the reward has a value significantly different from 0 if $(p_{t+1} - p_t) \gg c_t$, *i.e.* only when the profit generated by the action A_t is definitely higher than the cost of sustaining it. Having said that, what are the precise measurements that come into play? Firstly, to recover the correct dimensionality of the spurious signal generated by the autocorrelation, we must pass to another function: the autocovariance. Indeed, the autocorrelation function introduces a normalization term that affects the dimensionality of this calculation. However, as we can see from Equation (6.9), the dimensionality of K is approximately the square of its random variable: therefore, the estimator of the spurious signal is given by[6] $\sqrt{|K(k)|}$. On the other hand, the cost of an operation in FX market is of the order of $\frac{1}{2}\sigma_t$. Since we work with different time-steps, the mean of this

[6]We consider $\sqrt{|K(k)|}$ and not $\sqrt{K(k)}$ in order to avoid problematic situations with the square root.

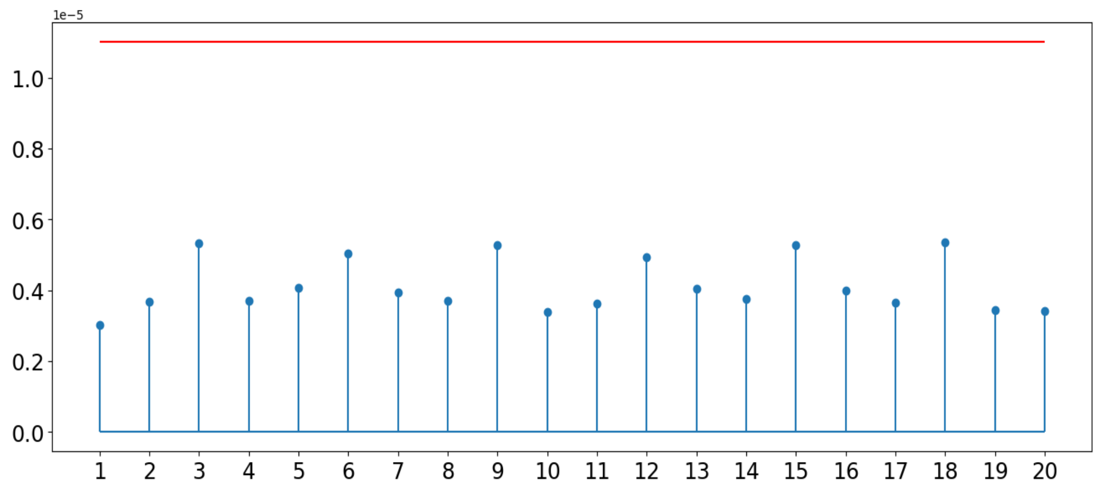


Figure 6.4: The blue dots represent the autocovariance of the process made of random variables as defined in (6.8) evaluated in the first 20 lags. The red line, instead, represents the average value of the bid-ask spread.

term is what is to be compared with the estimator.

As it is evident in Figure 6.4, $\sqrt{|K(k)|} < \frac{1}{2}\mathbb{E}[\sigma_t]\ \forall k \in \{1, 2, \dots, 20\}$, hence we can safely proceed working with this dataset.

6.3.2. Volatility

Now, we want to see if any volatility-related pattern shows up in our data. In order to do so, we introduce a further definition:

$$V_t := \frac{1}{d}\sum_{i=1}^{d} |p_{t+1,i} - p_{t,i}|\,, \tag{6.10}$$

where d indicates the number of days present in the dataset and $p_{t,i}$ stands for the exchange rate value at t-th minute of day i. The difference between V_t and X_t (6.8) relies only on the fact that the former considers the absolute value of the difference between two consecutive mid-prices instead of the pure value. Therefore, V_t tells us how much, on an average day, the modulus of the exchange rate varies between minute t and $t+1$. Hence, this term is a subspecies measure of volatility of the average process; the results are shown in Figure 6.5.

In Figure 6.5 two aspects come to the fore: the average higher volatility in the central hours of the day (approximately between 8:00 and 18:00) and the repetitive fluctuation spikes.

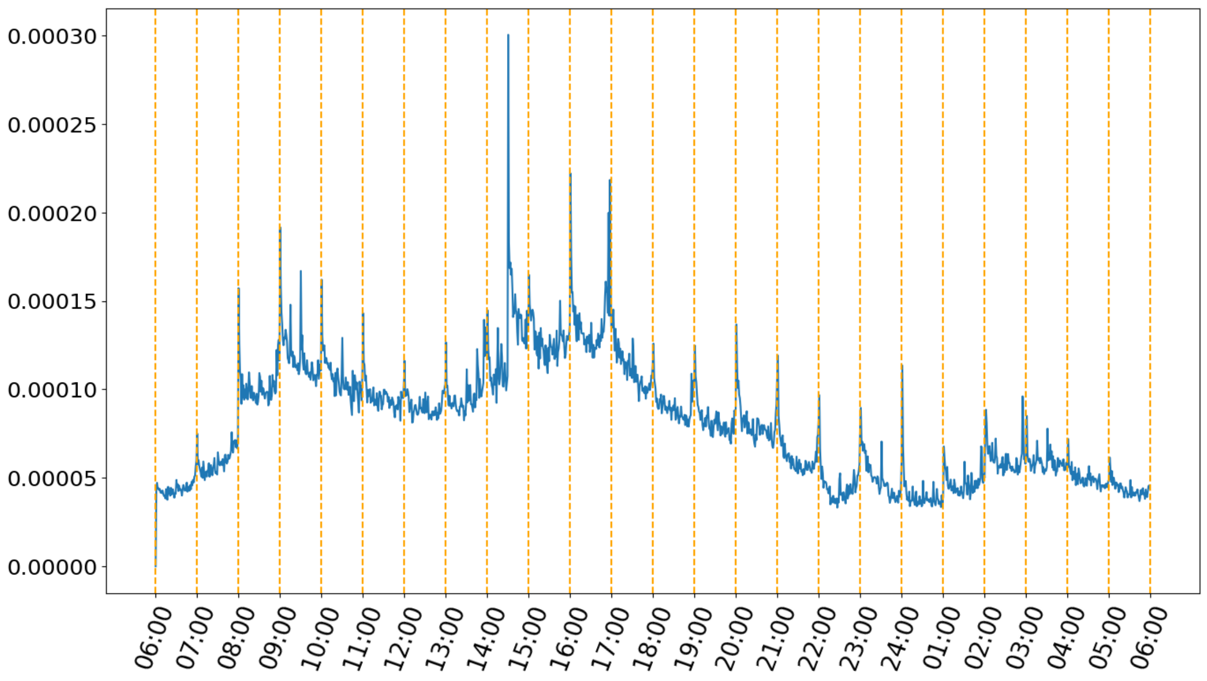


Figure 6.5: Mean volatility (V_t) values evaluated on HistData data over 2018 and 2019. The time labels refer to the CET (Central European Time) time zone.

The former has an easy explanation: in fact, the central hours coincide with all the hours in which the European stock exchanges and some of the American ones are open. More precisely, all the most important European stock exchanges open at 9:00 CET and close at 17:30 CET, whereas the New-York Stock Exchange opens at 15:30 CET and closes at 22:00 CET. Thus, in these hours the volumes quantity is higher, causing both higher volatility and more liquidity. All there reasons positively support our choice of considering only the data between 8:00 CET and 18:00 CET.

The latter, instead, must be analysed more deeply: surely, it is evident that almost all the spikes occur at exactly one hour (8:00, 9:00, 10:00...) or at a half-hour (9:30, 10:30, 14:30...). In our opinion, such a regular cadence can be caused both by the openings/closings of international markets and by automated traders, who only act on a regular basis. Whatever the reason of this phenomenon is, we are interested in how this pattern may influence our algorithm: we think that HPFQI can be consistently influenced by this only when working at persistence equal to 1, since with higher persistences very often it will not consider these instantaneous spikes.

In light of the results of these analyses, we can state that the dataset we have at our disposal cannot adversely affect the performance of our algorithm. Therefore, we can continue with the HPFQI experiments on FX.

7 | Experimental Results FX

In this section we present the results obtained by HPFQI in the FX market setting. In order to better analyse them, they will be compared with PFQI in the same way as we did in Section 5.2. In particular, we decided to divide the experiments in two frameworks: firstly, we train PFQI on 2018 and 2019 data[1], we validate the parameters on 2020 and we test the best ones on 2021. Secondly, with the same parameters, we train PFQI on 2019 and 2020 data and we test it on 2021. For HPFQI we follow almost the same flow: the first time we train it on 2018/2019 data and we test it on 2021; the second time we train it on 2019/2020 data and we test it on 2021. The only difference relies on the validation process: indeed, in both hierarchical cases, we do not perform hyperparameter tuning because we exploit the best parameters obtained during PFQI validation. We decided to structure the analyses with two different training sets so that the validation process is as reliable as possible. To explain it better, if one decided to train the algorithm in 2018/2019 and validate it in 2020, one would certainly be tempted to re-train it with the best parameters in 2018/2019/2020 for eventually test the final model in 2021. However, this is a mistake not to be underestimated: indeed, the parameters that were the best trained in two years would no longer be the best trained in three years. This happens because the hyperparameters are extremely sensitive to a change in the size of the training set, hence validation would no longer be of any use. Furthermore, simply training the algorithm on 2018/2019 and then testing on 2020 would not have been correct, because 2020 is also the dataset in which we performed the validation.

7.1. Model Selection

7.1.1. Regressor Parameters

Unlike in the Mountain Car setting (see Section 5.2.1), to fit the action-value function in FX we exploited XGBoost due to its great performance in terms of accuracy, its high scalability and its computational efficiency when dealing with very large training sets. As we

[1]The data, as anticipated in Section 6.3, were provided by HistData.

have already anticipated in Section 2.5, XGBoost is a state-of-the-art decision-tree-based ensemble algorithm used both for classification and regression. The hyperparameters to be tuned are numerous and can be divided in two classes: one related to the construction of the forest of decision trees and the other to the gradient boosting framework.

The former includes parameters such as the number of trees (`num_parallel_tree`), the minimum weight of each child (`min_child_weight`), the maximum depth of each tree (`max_depth`), and some other parameters that introduce additional randomness (some examples are: `subsample`, `colsample_bytree`, `colsample_bylevel`, `colsample_bynode`). Typically, the most handled parameter is `min_child_weight`. `min_child_weight` specifies the minimum sum of instance weight needed in a child (*i.e.*, leaf) for the splitting to occur. In simpler terms, it controls the minimum number of samples required to be present in each leaf of a tree. Thus, a higher value of `min_child_weight` can help in preventing overfitting and can lead to a more generalized model. However, setting it too high can lead to underfitting as the model may not be able to capture the necessary patterns in the data. `num_parallel_tree` and `max_depth` can help `min_child_weight` to regulate better the correct shape of the trees. Furthermore, as anticipated, XGBoost can rely also on different sub-sample strategies that allow taking into account more complex models keeping under control the risk of overfitting. More specifically, in addition to the possibility of randomly selecting a subset of the training samples before building each tree, three sub-sample ratios of features can be specified: the number of features sub-sampled before training each tree, the number of features sub-sampled at each level, and the number of features sub-sampled at each node.

The latter, instead, includes the number of boosting rounds (`num_boosting_rounds`), the learning-rate parameter η and the regularization parameters α and λ. In particular, η is the parameter that, at each boosting round, regulates the weight of old models, whereas α and λ respectively define the L1 and L2 regularization terms inside the optimization function. Although they pursue the same objective in different ways, all these parameters allow to control the model complexity and limit the risk of overfitting.

7.1.2. Other Parameters

The only parameters left to consider are the number of iterations and the persistence values. For what concerns the first one, we have noticed that, in almost every situation, from a certain value onwards there is no change in performance, hence we always fixed its value to 8. For the second one, we considered the persistences $k \in \mathcal{K} = \{30, 15, 10, 5, 1\}$. We chose these values because they seemed interesting to us also in light of the FX market

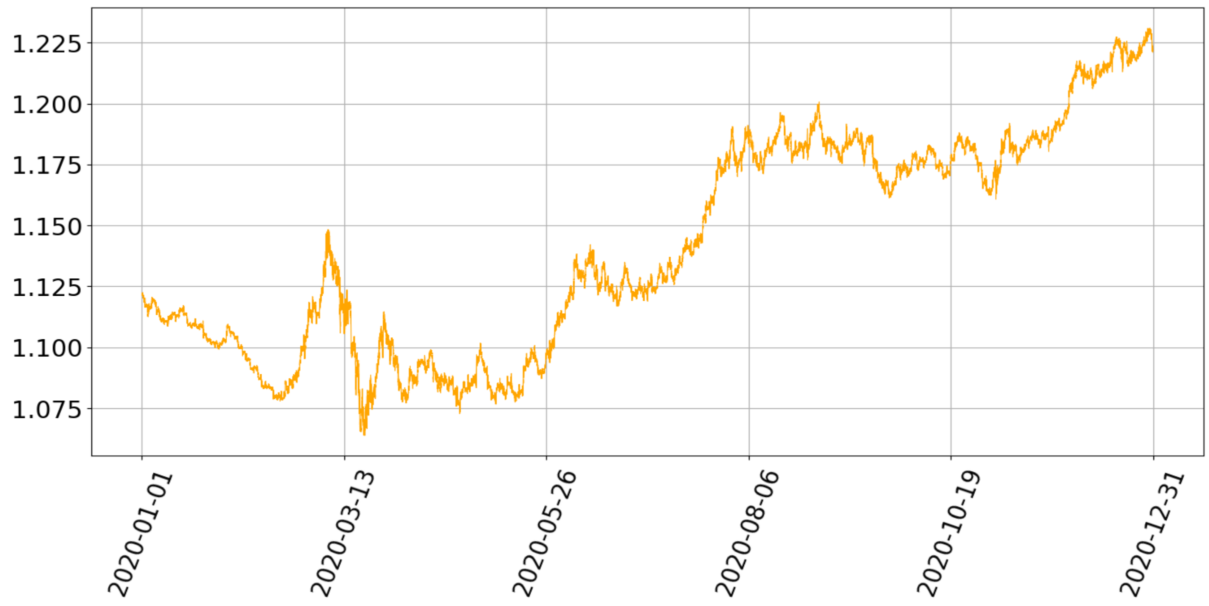


Figure 7.1: EUR/USD exchange rate on 2020.

regularities seen in Section 6.3.

7.2. Results

The further sections are structured as follows: firstly, PFQI results in different shapes are showed, and thereafter HPFQI performances are presented. Afterwords, the outcomes will be compared before coming to conclusions.

7.2.1. PFQI

Hyperparameters Tuning

As aforementioned in the beginning of Chapter 7, for each persistence we first trained PFQI on 2018/2019 data with different sets of hyperparameters in order to validate them on 2020. Notably, depending on the persistence, we exploited a different formulation of the dataset, referring to (6.5) and (6.6).

The last consideration we feel like making before going to the actual results concerns the performance of the EUR/USD market in 2020.

As Figure 7.1 clearly displays, on March and April of 2020 there has been a unusual behaviour: indeed, due to COVID-19 pandemic hitting Italy and soon afterwords the rest of Europe, EUR/USD rates drastically dropped with an enormous day-by-day volatility.

Final Parameters

	k = 30	k = 15	k = 10	k = 5	k = 1
min_child_weight	40000	40000	60000	40000	10000
max_depth	8	6	6	5	6
num_parallel_tree	50	50	50	50	50
num_boosting_rounds	5	5	5	5	5
colsample_bytree	0.8	0.8	0.8	0.8	0.8
colsample_bylevel	0.8	0.8	0.8	0.8	0.8
colsample_bynode	0.8	0.8	0.8	0.8	0.8
subsample	0.8	0.8	0.8	0.8	0.8
η	0.3	0.3	0.3	0.3	0.3
α	0	0	0	0	0
λ	1	1	1	1	1
Iterations	8	8	8	8	8
γ	1	1	1	1	1

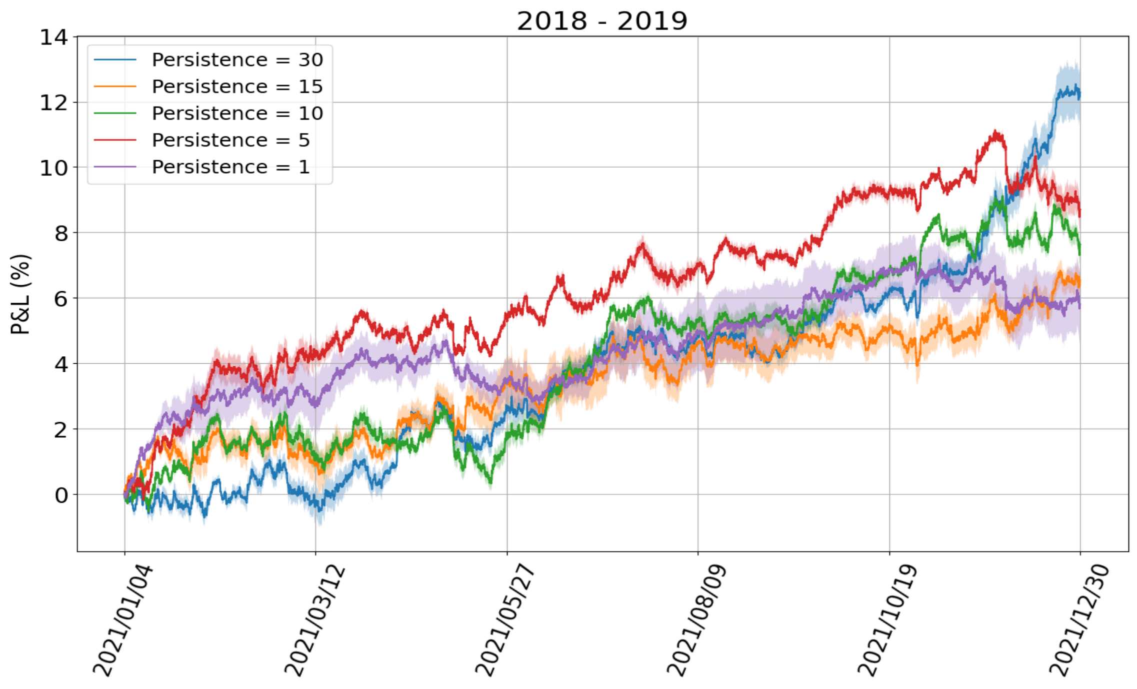
Table 7.1: Final parameters for each persistence obtained after XGBoost (first batch or rows) and PFQI (second batch of rows) parameters fine-tuning.

For our purpose, this behaviour should be taken with a grain of salt since it could be determinant, positively or negatively, in the choice of the best set of parameters. In order to avoid this issue, we did not select the best parameters based only on the final P&L performance but we weighted it on how the algorithm behaved in that particular period.

Having said that, for each persistence in $\mathcal{K}$ we performed the validation on 2020 over three seeds and the best parameters obtained are shown in Table 7.1.

Training and Testing

After having found the best parameters for each persistence, we trained PFQI on 2018/2019 and on 2019/2020, testing them both on 2021. In both cases, we considered three different seeds. For ease of readability, we call PFQI$_1$ the PFQI algorithm trained on 2018/2019 and PFQI$_2$ the PFQI algorithm trained on 2019/2020. The results are shown in Figure 7.2.



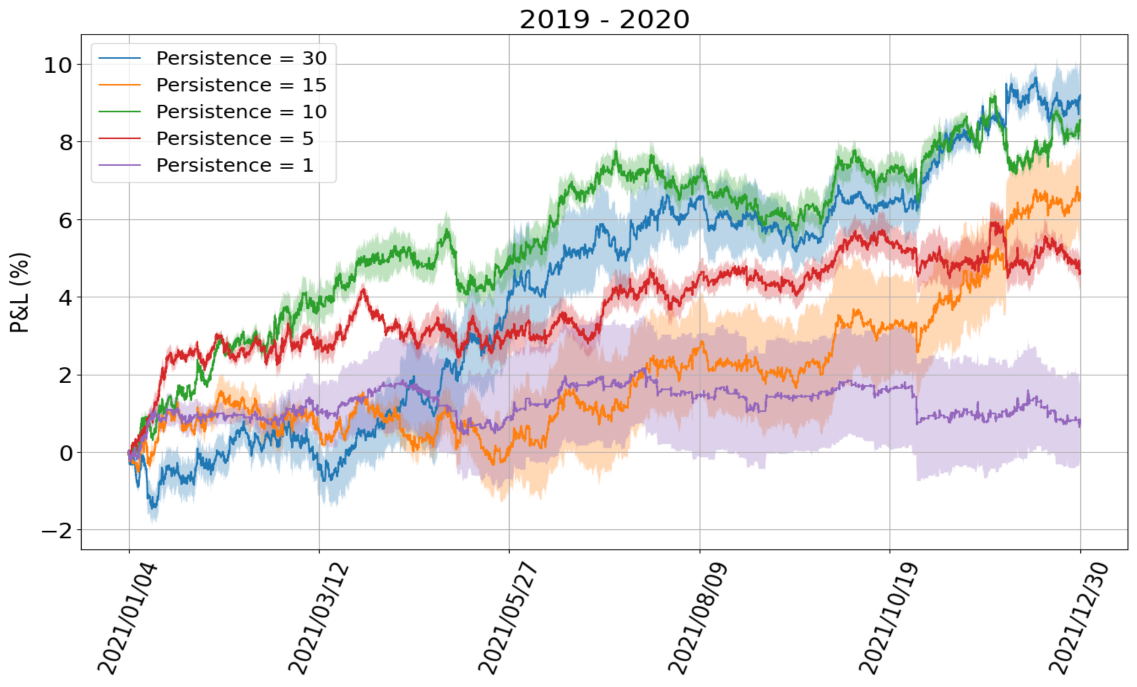


Figure 7.2: PFQI cumulative P&L on 2021 at each persistence when training is done on 2018/2019 (first figure) or on 2019/2020 (second figure) over three different seeds. For each persistence, the bold line represents the average cumulative P&L among the seeds, whereas the shaded color band indicates one standard deviation interval.

The first aspect that stands out from Figure 7.2 is the performance gap between the two training sets. In fact, for almost any persistence, $PFQI_1$ shows a higher P&L than $PFQI_2$.

Furthermore, the confidence bands show a considerably higher variance in the second setting.

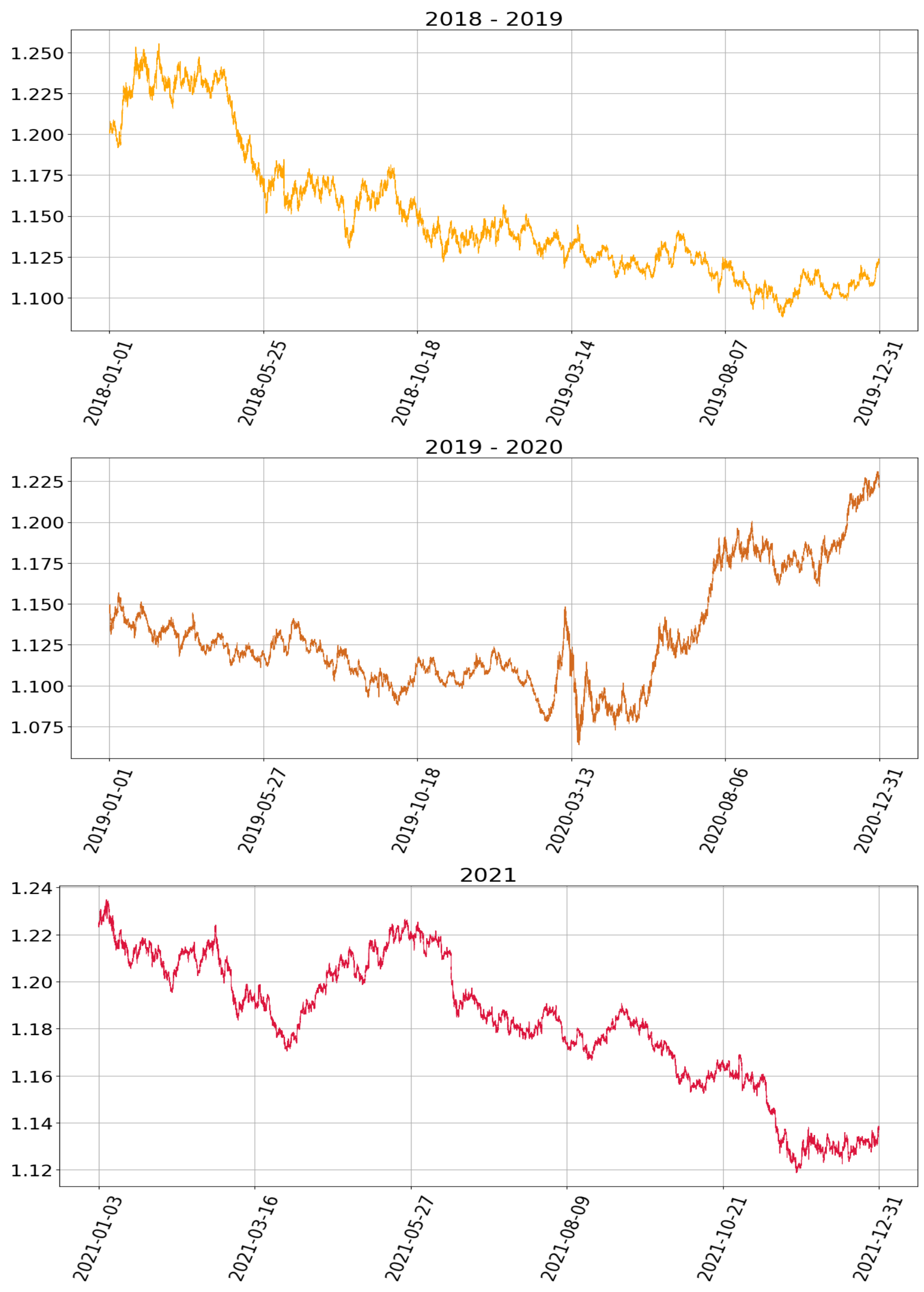
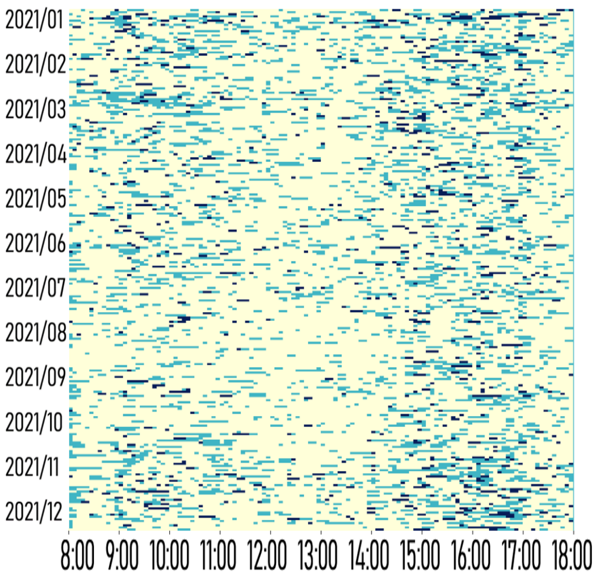
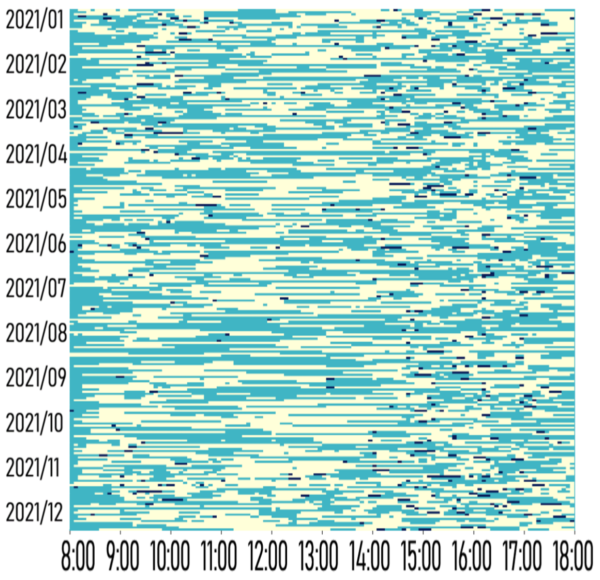


Figure 7.3: EUR/USD exchange rate on 2018/2019 (first figure), on 2019/2020 (second figure) and on 2021 (third figure). Respectively, they represent the training set of PFQI$_1$, the training set of PFQI$_2$ and the test set of both.

We can go into more detail thanks to Figure 7.3, where the training set of PFQI$_1$, the



(a) Persistence = 5 of PFQI$_1$



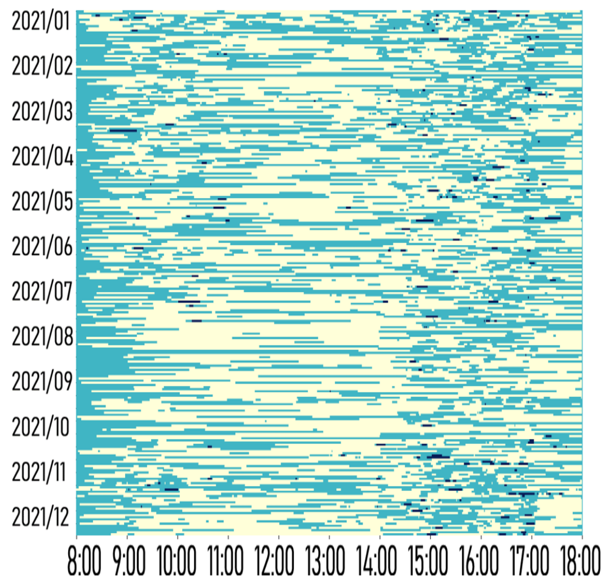
(b) Persistence = 5 of PFQI$_2$

Figure 7.4: Actions taken in 2021 by PFQI$_1$ (left) and PFQI$_2$ (right) at persistence equal to 5. Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.

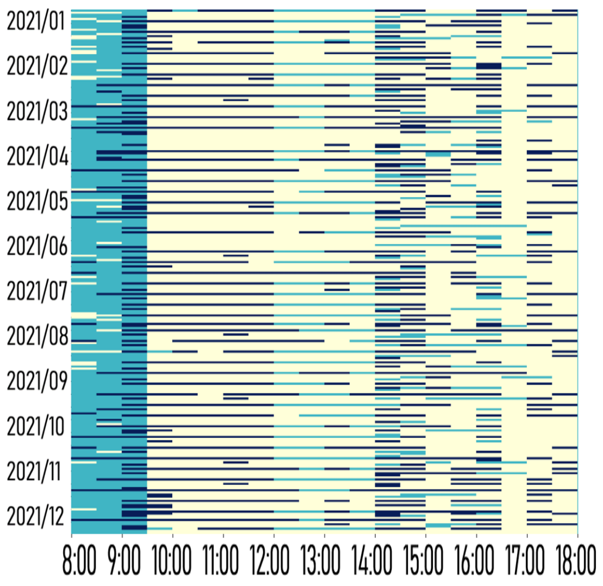
training set of PFQI$_2$ and the test set displayed. In fact, the 2021 trend is closer to that of 2018/2019 than to that of 2019/2020: in particular, the difference is made by 2020, because it is the only year in which there is a long increase of the rate. Therefore, the better results of PFQI$_1$ surely depend at least partially on the higher similarity of its training set with the test set. Furthermore, as we mentioned in Section 7.2.1, the 2020 itself shows also a higher volatility and unpredictability which does not help PFQI$_2$ agent in making perfect decisions.

Figure 7.4 helps us to better grasp what a difference in the training set entails: indeed, PFQI$_1$ is more confident in itself, going pretty often short. On the other hand, PFQI$_2$ is not much helped by its training set, which often leads it to hold no position at all (*i.e.* to stay flat). As a consequence, PFQI$_1$ obtains a P&L that is approximately twice as high as PFQI$_2$'s one. Nevertheless, both PFQI$_1$ and PFQI$_2$ almost always show a positive P&L over all the 2021.

A further element that shines through Figure 7.2 is the different performance between the various persistences. In general, $k = 30$ shows the highest P&L whereas $k = 1$ shows the lowest one. The values in between, instead, behave in a less defined manner. For instance, PFQI$_1$ with persistence equal to 5 performs better than PFQI$_1$ with persistence



(a) Persistence = 1



(b) Persistence = 30

Figure 7.5: Actions taken in 2021 by PFQI$_1$ at persistence 1 (left) and PFQI$_1$ (right) at persistence equal to 30. Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = -1 (yellow) means that the agent sells.

equal to 10 and 15. On the other hand, PFQI$_2$ with persistence equal to 10 performs better than PFQI$_2$ with persistence equal to 15 and 5. The difference between the various performances surely depends on the different way of acting by the agent at a given persistence, as it is shown in Figure 7.5.

Indeed, with $k = 1$ the policy is enormously fragmented, the agent often stays flat and almost never buys. Therefore, given that almost the only active position it holds is being short, the positive final P&L of PFQI$_1$ at persistence equal to 1 is uniquely obtained because during 2021 on average the exchange rate went down. On the other hand, with $k = 30$ the policy is definitely organized. In fact, often both hourly behaviours (e.g.: at 9:30 CET the agent almost always buys) and daily behaviours (e.g.: every Monday the agent follows approximately the same pattern) are replicated along the year. Surely this configuration helps the agent not to act chaotically, but it creates the risk that the agent does not grasp the subtleties or is unable to react to sudden changes. This is also why we have created HPFQI, the results of which we display in the following section.

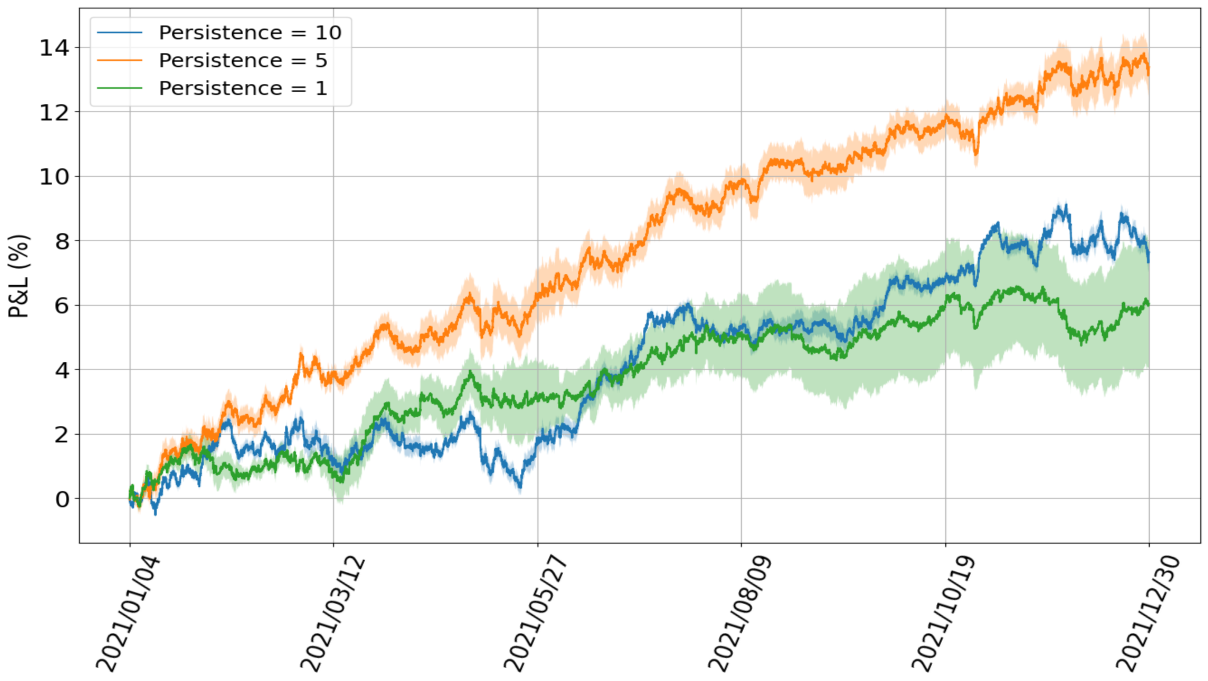


Figure 7.6: HPFQI($\mathcal{K}_1$) test results with training in 2018/2019.

7.2.2. HPFQI

In this section we first analyse the results of HPFQI trained on 2018/2019 and then the ones with the training set made of 2019/2020 data. In both cases, we considered different hierarchical structures, in order to better understand both the potentialities and the limitations of HPFQI. Before going to essentials, we must take some precautions to make the reading more fluent. Firstly, from now on the term HPFQI will identify the HPFQI algorithm trained on a different training set depending on the section in which we talk about that. Secondly, we will write HPFQI($\mathcal{K}_1$), HPFQI($\mathcal{K}_2$), HPFQI($\mathcal{K}_3$) to identify HPFQI trained and tested on $k \in \mathcal{K}_1 = \{10, 5, 1\}$, $k \in \mathcal{K}_2 = \{15, 10, 5, 1\}$ and $k \in \mathcal{K}_3 = \{30, 15, 10, 5, 1\}$ respectively.

Training Set: 2018 - 2019

After having tested several persistences combinations, in 2018/2019 HPFQI showed the most interesting results on the following ones: $\mathcal{K}_1 = \{10, 5, 1\}$, $\mathcal{K}_2 = \{15, 10, 5, 1\}$ and $\mathcal{K}_3 = \{30, 15, 10, 5, 1\}$. Therefore, the results we about to present are related to HPFQI trained with these three combinations on 2018/2019 and tested on 2021, all this with three different seeds.

First of all, let us take a look at HPFQI outcome with persistences in $\mathcal{K}_1$, displayed in

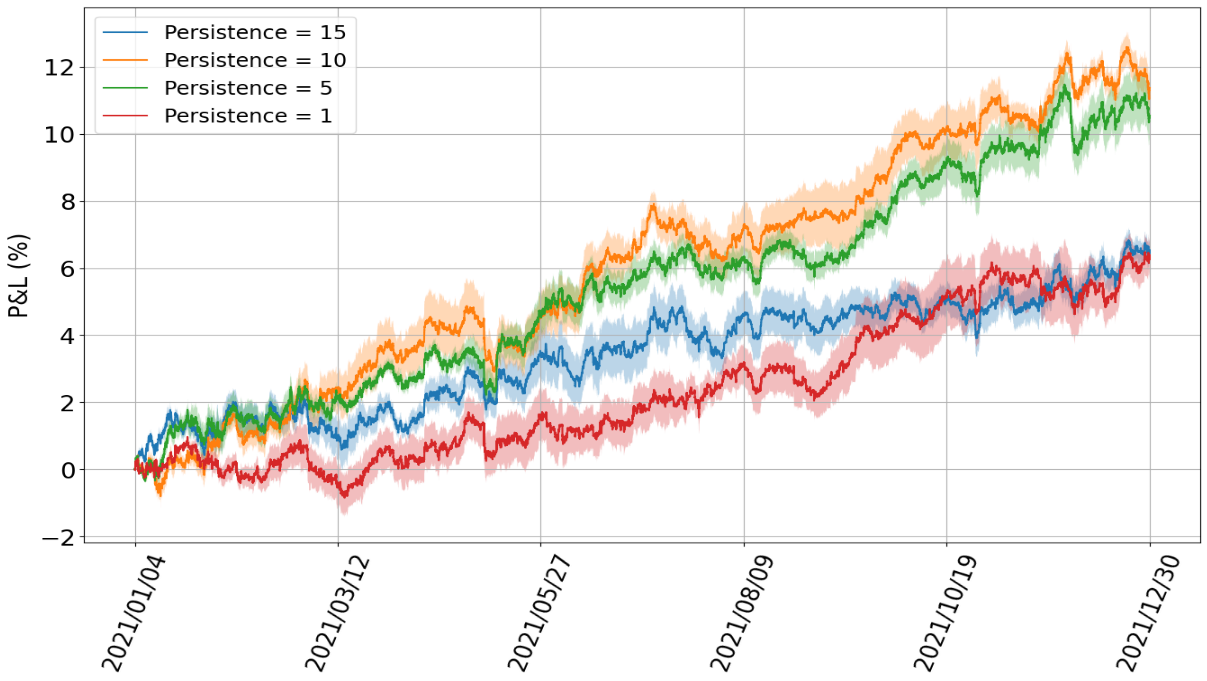


Figure 7.7: HPFQI($\mathcal{K}_2$) test results with training in 2018/2019.

Figure 7.6.

Obviously, in this case and in all the other HPFQI declinations, the training of the highest persistence of HPFQI coincides exactly with PFQI at that persistence[2]. Therefore, the relevant persistences to watch are all excluded the highest one.

What we can see from Figure 7.6 is that at persistence 5 HPFQI outperforms persistence 10 by quite a lot, not showing any long decreasing trend along all the year. The transfer of the action-value function from $k = 5$ to $k = 1$, however, does not bear its fruits: in fact, the P&L cumulated with $k = 1$ slightly underperforms HPFQI at persistence equal to 10 in terms of both P&L and variance.

When we consider HPFQI trained and tested on $\mathcal{K}_2$ (Figure 7.7) we spot a similar behaviour: at the persistences in between (10 and 5) the P&L is definitely higher than the one obtained at the highest persistence, testifying that the passage of the action-value function worked. However, at persistence equal to 1 the performance declines, reconnecting with the one obtained at the highest persistence. In particular, we can notice two aspects: with respect to HPFQI trained and tested on $\mathcal{K}_1$, here the profit with $k = 10$ is higher, whereas with $k = 5$ is slightly lower. In other words, the former states that PFQI at persistence equal to 10 is definitely outperformed by HPFQI at persistence equal to

[2]Indeed, APFQI at the first persistence compares its action-value function with the ones present in $\mathcal{Q}$, but in this situation $\mathcal{Q} = \emptyset$.

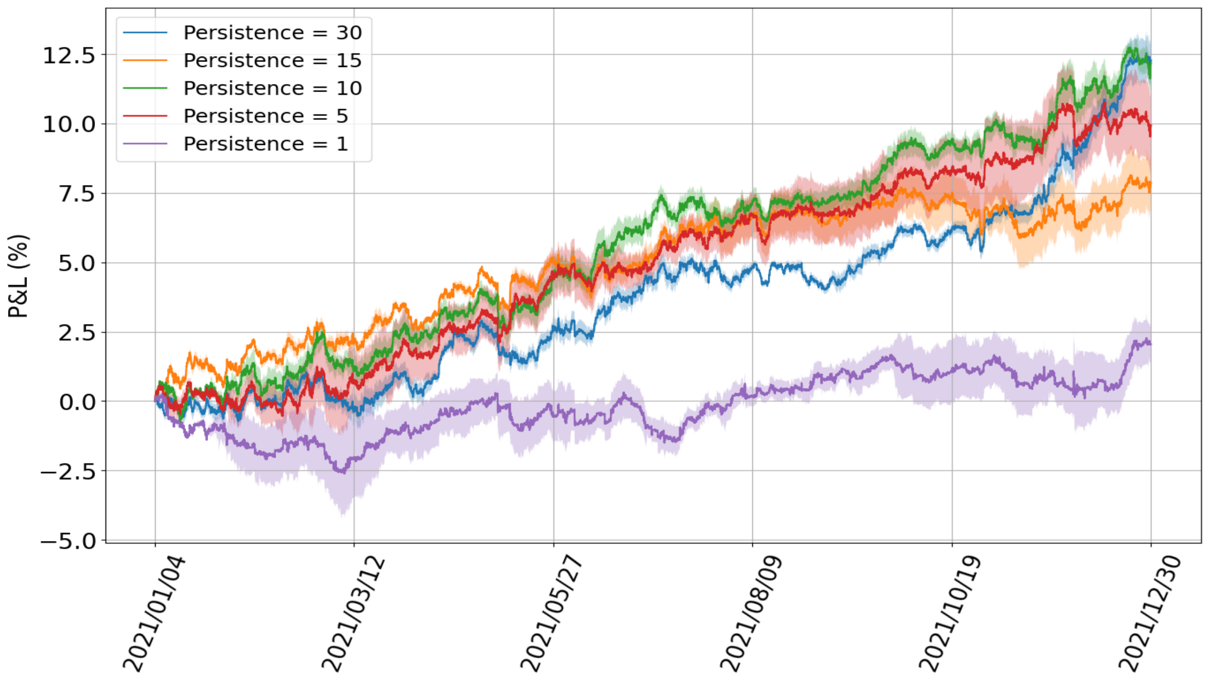


Figure 7.8: HPFQI($\mathcal{K}_3$) test results with training in 2018/2019.

10 reached after one hierarchical step; this means that the hierarchy helps the agent to act better. The latter, instead, suggests that maybe a too long hierarchical chain could worsen the results at lower persistences or at least could slightly deface them.

In order to investigate better the last consideration, we trained and tested HPFQI with $k \in \mathcal{K}_3$. As it is visible from Figure 7.8, the overall pattern is almost replicated: HPFQI at persistence 15 performs better than PFQI at that persistence (see Figure 7.7), at persistence 10 and 5 is actually quite good but at persistence equal to 1 the performance drops drastically; this trend suggests once again that the hierarchical chain was too long.

Training Set: 2019 - 2020

For what concerns the second training set, *i.e.* the data of 2019 and 2020, interesting results have come out only from combinations $\mathcal{K}_1$ and $\mathcal{K}_3$.

Unlike what we saw in Figure 7.6, Figure 7.9 shows an opposite pattern: with $k = 5$ HPFQI worsens the performance with respect to $k = 10$ but with $k = 1$, reaching eventually the profit obtained at persistence equal to 5 also with less variance, it achieves an unhoped result. In addition, as we will see later, HPFQI with $k = 1$ achieves a profit way higher that PFQI at that persistence.

However, when the hierarchy becomes longer, the outcomes are similar to Section 7.2.2.

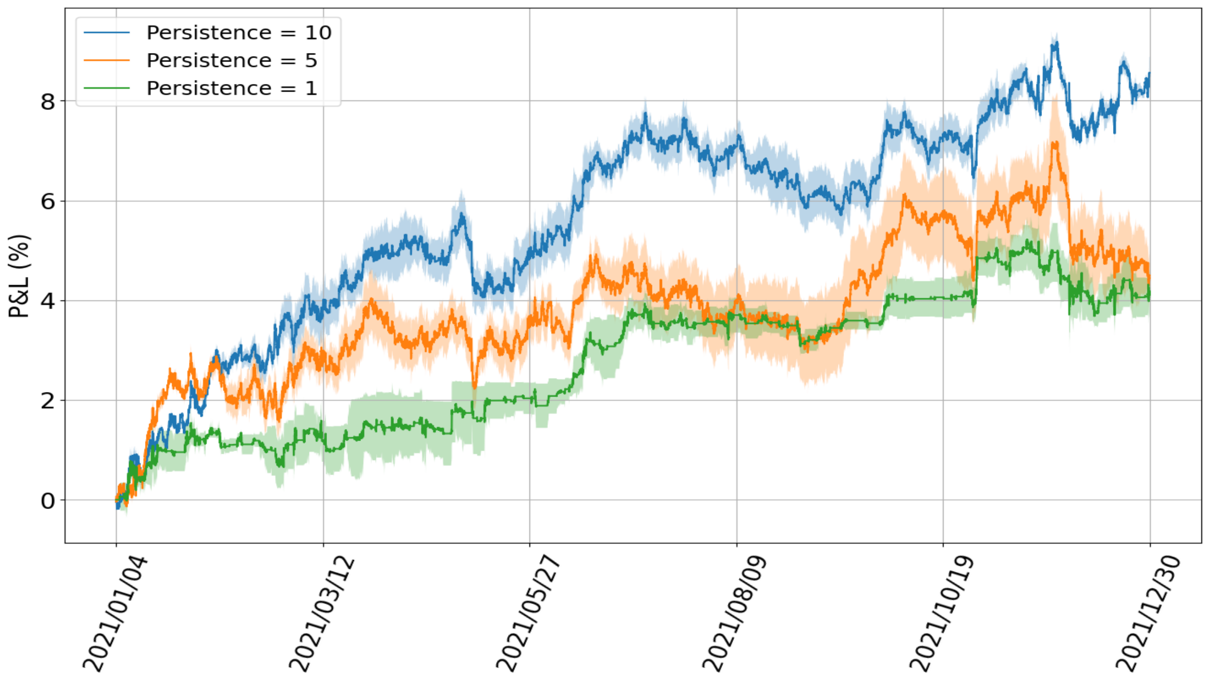


Figure 7.9: HPFQI($\mathcal{K}_1$) test results with training in 2019/2020.

In fact, some good results are obtained in the persistences in the middle, whereas at persistence equal to 1 all the advantage gained is lost.

7.2.3. HPFQI vs PFQI

This is the crucial section of the entire work. Indeed, we will analyse in detail all the results obtained by HPFQI comparing them with our benchmark, PFQI. We will investigate all the differences, both positive and negative, in order to draw meaningful conclusions and finally to propose new ways to tackle this kind of problem.

For sake of clarity, both in Section 7.2.3 and in Section 7.2.3 we never compare the performances of the higher persistence of a certain HPFQI structure with PFQI at that persistence, since always they are exactly the same algorithm (see Section 4.2).

Training Set: 2018 - 2019

Firstly, we take a look at HPFQI($\mathcal{K}_1$) results compared to PFQI at the respective persistences, supported by Figure 7.11.

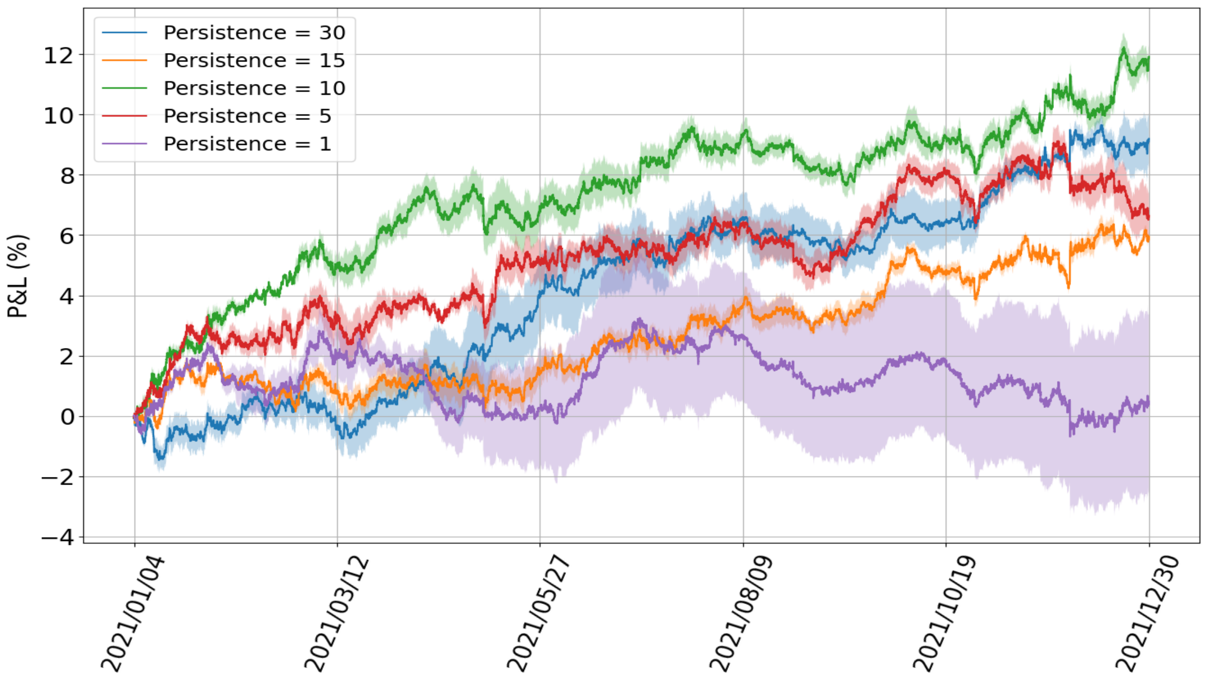
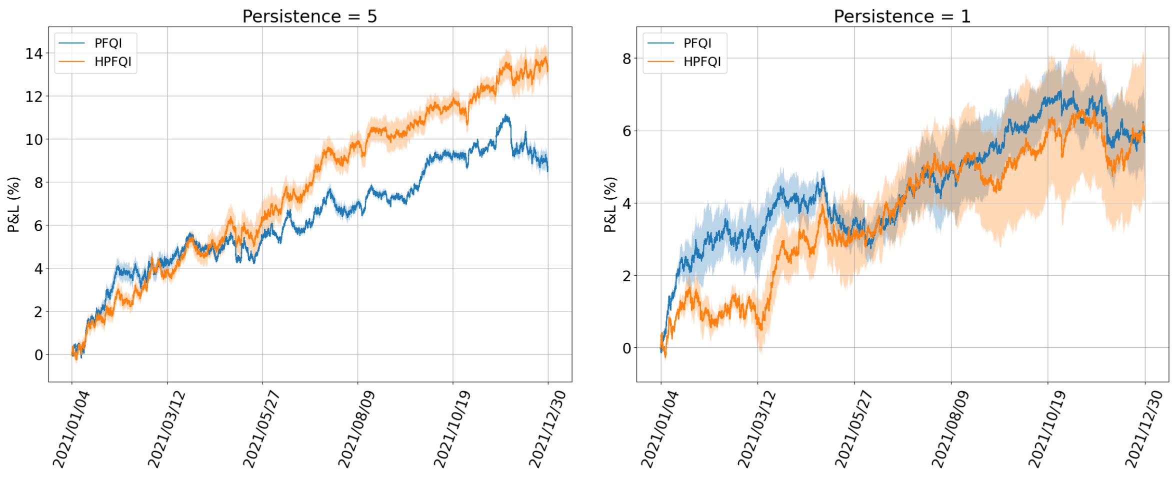


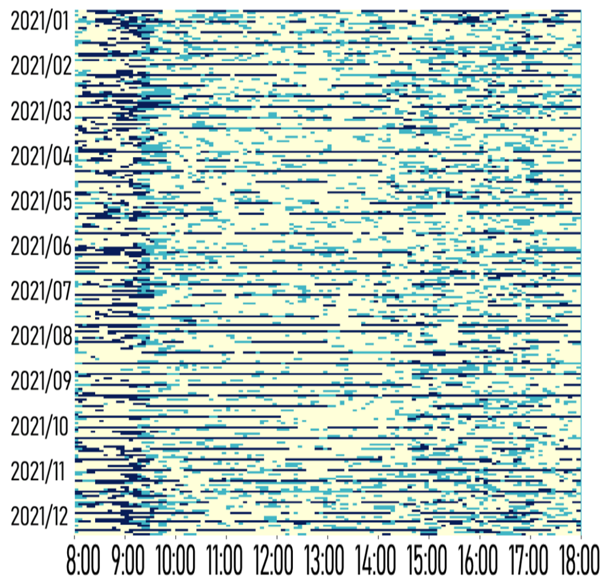
Figure 7.10: HPFQI($\mathcal{K}_3$) test results with training in 2019/2020.



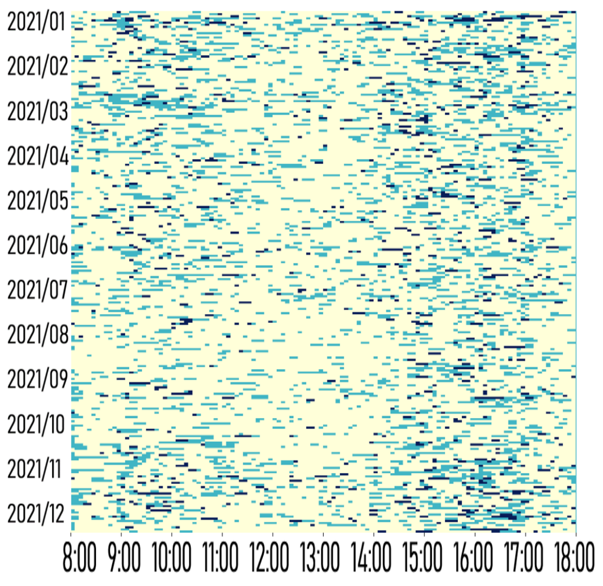
(a) (b)

Figure 7.11: Comparison between the test P&L of each HPFQI($\mathcal{K}_1$) persistence and PFQI test P&L at the same persistence.

Fortunately, Figure 7.11a shows that transporting the action-value function from persistence 10 to persistence 5 helped the latter to track the exchange rate trends better than PFQI at persistence 5. In particular, HPFQI profit almost always shows an upward trend, whereas PFQI's often flattens out. On the other hand, however, Figure 7.11b shows that HPFQI at the last persistence loses the advantage gained at the previous hierarchical



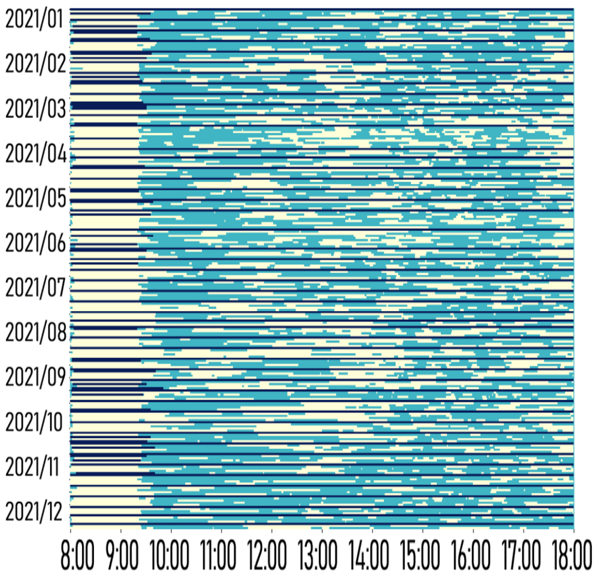
(a) Persistence = 5 of HPFQI($\mathcal{K}_1$)



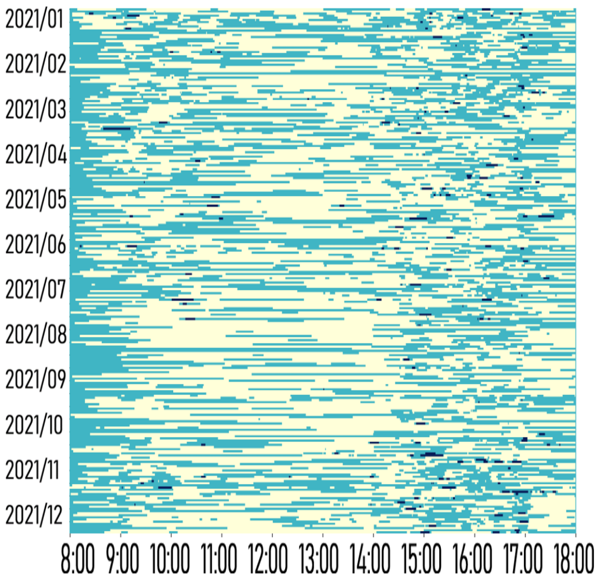
(b) Persistence = 5 of PFQI

Figure 7.12: Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 5 (left) and PFQI at persistence 5 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.

step, also heavily increasing its variance. These differences are determined above all by the distinct way in which the two agents behave: Figure 7.12 shows that, at persistence equal to 5, HPFQI($\mathcal{K}_1$)'s policy is better organized than PFQI's. Indeed, HPFQI (left) displays repetitive patterns both hourly-wise and daily-wise and in general it buys more frequently. This behaviour could be the main reason why HPFQI performs better than PFQI, although we cannot state this with certainty. On the other hand, PFQI's policy is more chaotic as it does not show any visible pattern.



(a) Persistence = 1 of HPFQI($\mathcal{K}_1$)

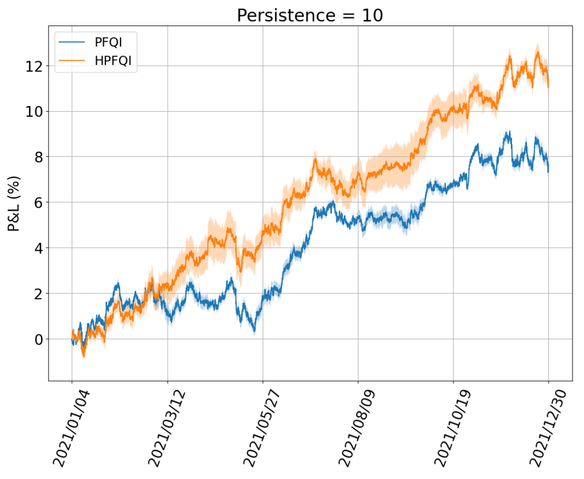


(b) Persistence = 1 of PFQI

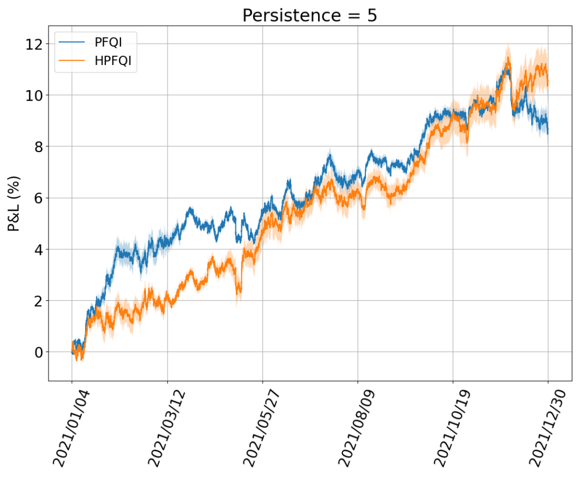
Figure 7.13: Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 1 (left) and PFQI at persistence 1 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.

When descending to persistence equal to 1, instead, the hierarchy imposes an over-structured policy whose performance is easily achieved by a much less organized one, as it is evident by taking a look at Figure 7.13. HPFQI's behaviour is confirmed by the feature importance results of buy action generated at the end of the testing procedure, which places time as the fourth most important feature, whereas for PFQI it is not even in the top 20.

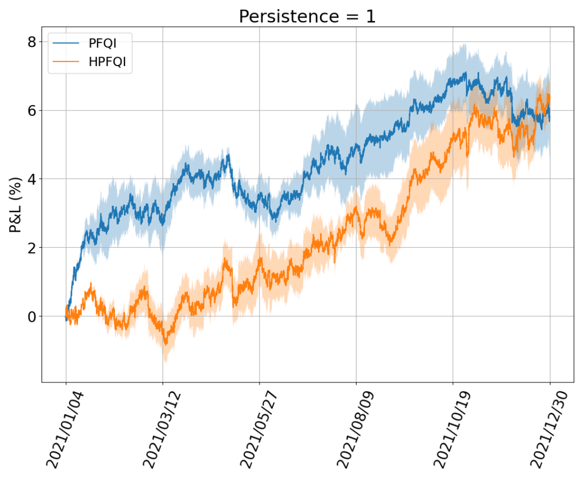
Now we take a look at Figure 7.14 to examine HPFQI($\mathcal{K}_2$)'s performances compared with PFQI ones.



(a)



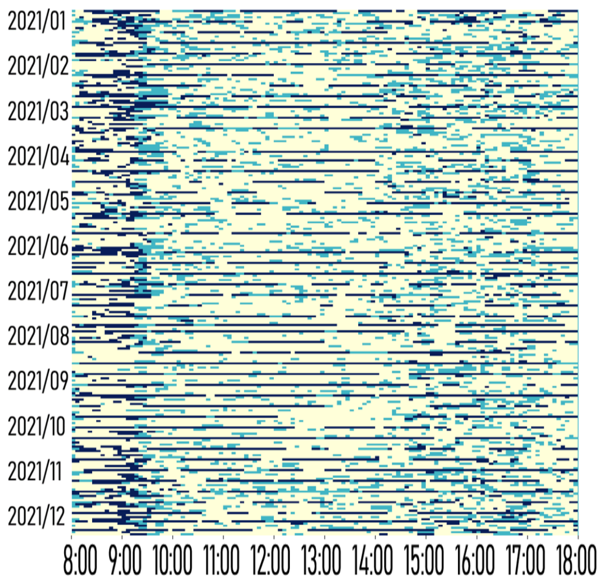
(b)



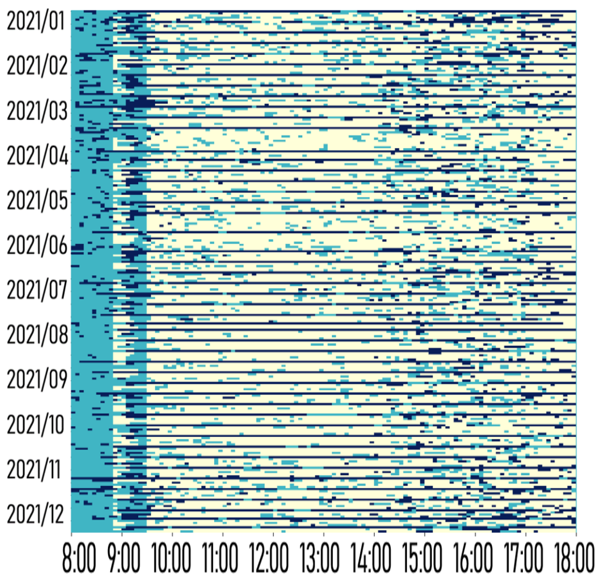
(c)

Figure 7.14: Comparison between the test P&L of each HPFQI($\mathcal{K}_2$) persistence and PFQI test P&L at the same persistence.

In these graphs it seems to be revisiting the outcomes just mentioned but stretched on a longer hierarchy. Indeed, at the first hierarchical step, *i.e.* at persistence 10, HPFQI achieves a profit definitely higher than PFQI at that persistence. At persistence 5, instead, HPFQI starts worse than PFQI at that persistence but eventually the former just overtakes the latter. Finally, at the last persistence for almost all the year HPFQI performs definitely worse than PFQI at persistence equal to 1. The comparison with HPFQI($\mathcal{K}_1$) is rather straightforward: with $k = 10$ HPFQI($\mathcal{K}_2$) clearly outperforms HPFQI($\mathcal{K}_1$), with $k = 5$ the opposite occurs and with $k = 1$ the performances are similar. This whole thing seems to justify the reasoning we had already mentioned: at the first hierarchical step HPFQI brings considerable advantages with respect to PFQI but, as the hierarchical chain grows longer, this positive effect gradually fades away.



(a) Persistence = 5 of HPFQI($\mathcal{K}_1$)

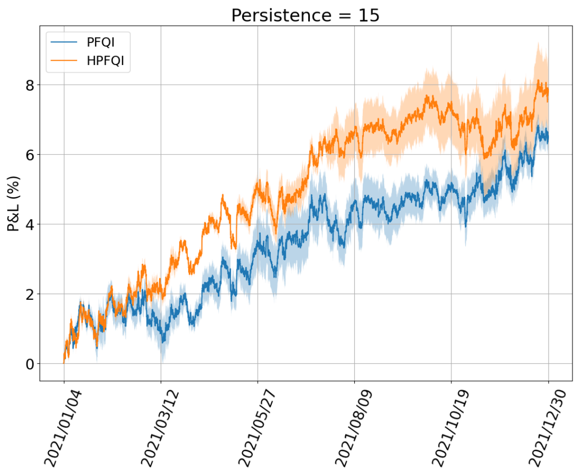


(b) Persistence = 5 of HPFQI($\mathcal{K}_2$)

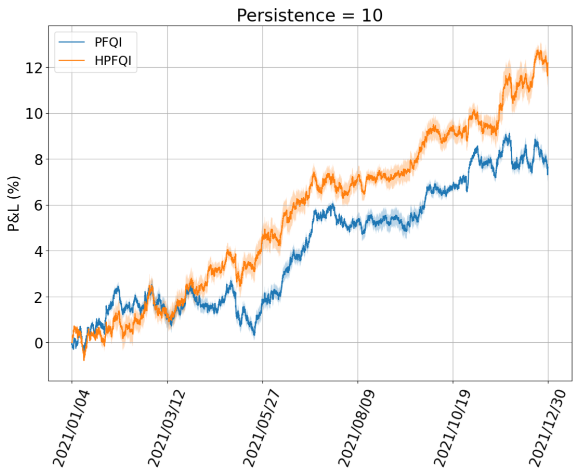
Figure 7.15: Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 5 (left) and HPFQI($\mathcal{K}_2$) at persistence 5 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.

In particular, it is interesting to compare the two hierarchical algorithms seen so far in terms of actions taken by their agents. As it emerges from Figure 7.15, with $k = 5$ the difference lies in how much the policy is temporally structured: if there is only one hierarchical step, *i.e.* for HPFQI($\mathcal{K}_1$), a behavioural organization is shown but there is still room for improvisation dictated by a possible anomalous behaviour of the exchange rate curve. When persistence equal to 5 is reached after two hierarchical steps, instead, the time pattern is way more evident, making the agent more prone to a pre-set but less flexible policy.

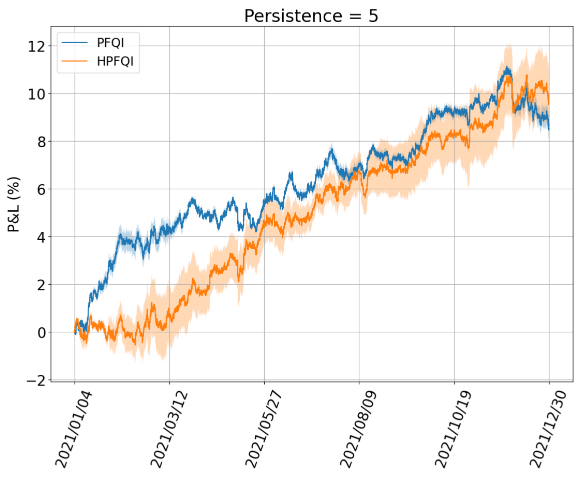
From HPFQI($\mathcal{K}_3$) we expect a similar conduct and Figure 7.16 does not contradict us. Indeed, in the first two hierarchical steps ($k = 15$ and $k = 10$) HPFQI obtains a profit definitely higher than PFQI at the respective persistences. At persistence 5 the performances are similar, whereas at persistence 1 HPFQI completely misses a decent policy, even risking ending the year with a loss. Remarkable what can be seen from Figure 7.17: after four hierarchical steps, the policy of the HPFQI agent is almost all deterministic, being the whole graphic made of horizontal and vertical bands. Indeed, the hierarchical agent seems to act based only on the time of the day and on the day itself. This is confirmed by the feature importance performed on all the possible actions: indeed, in all three cases the two most important features are the day of the week and the time of the



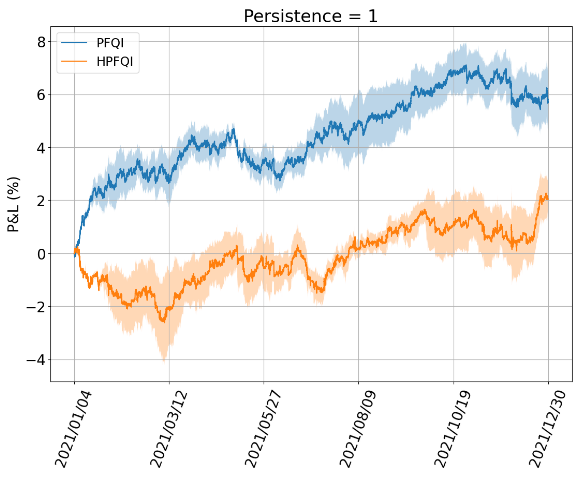
(a)



(b)



(c)



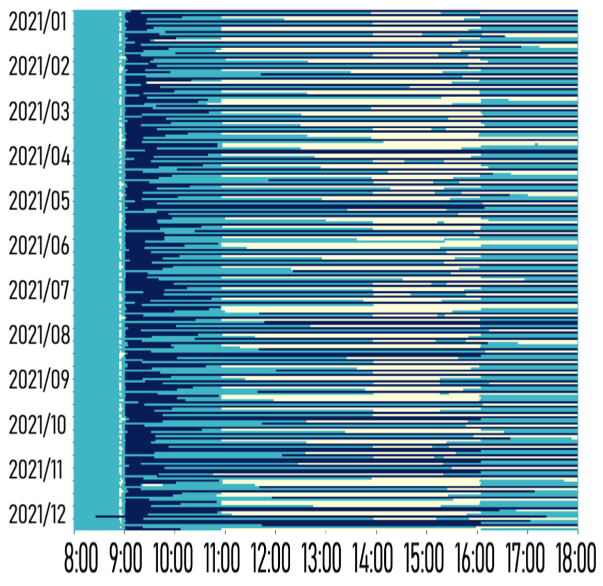
(d)

Figure 7.16: Comparison between the test P&L of each HPFQI($\mathcal{K}_3$) persistence and PFQI test P&L at the same persistence.

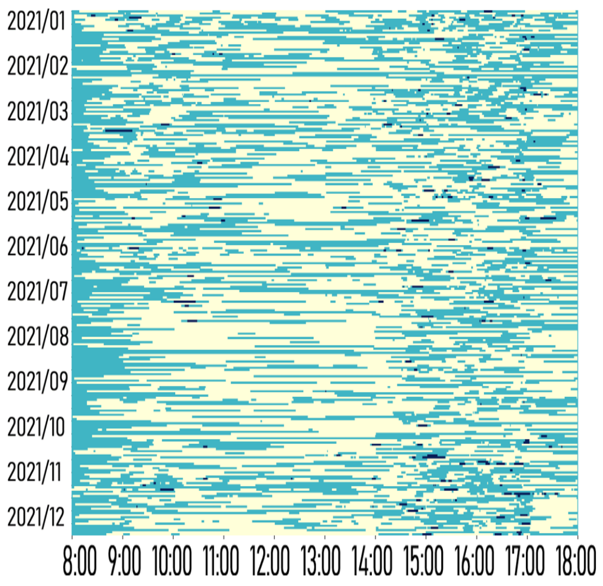
day.

Training Set: 2019 - 2020

For what concerns the other training set, 2019/2020, the results are quite different. For instance, let us consider HPFQI($\mathcal{K}_1$) compared with PFQI at the respective persistences.

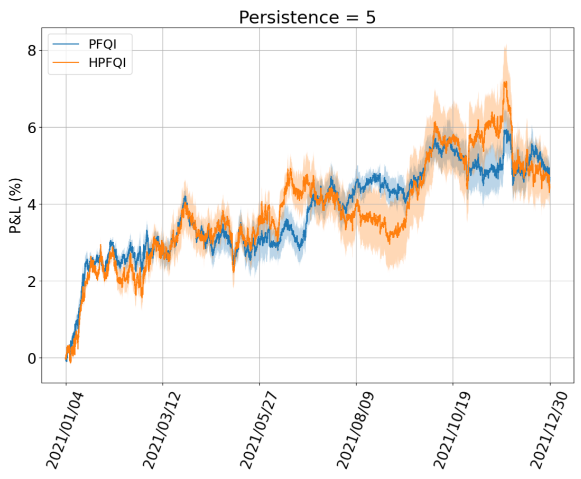


(a) Persistence = 1 of HPFQI($\mathcal{K}_3$)

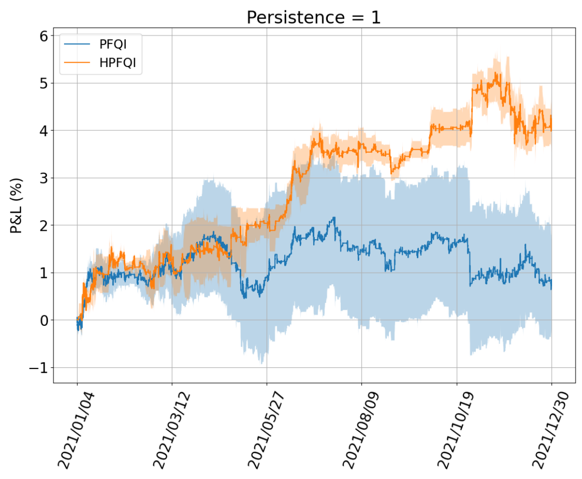


(b) Persistence = 1 of PFQI

Figure 7.17: Actions taken in 2021 by HPFQI($\mathcal{K}_3$) at persistence 1 (left) and PFQI at persistence 1 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.



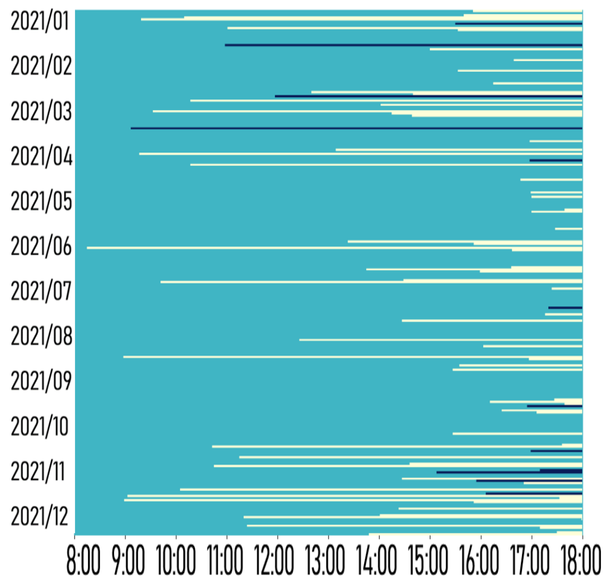
(a)



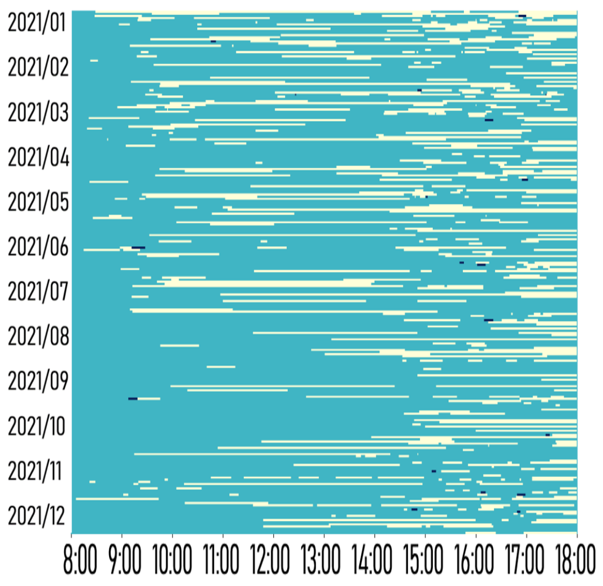
(b)

Figure 7.18: Comparison between the test P&L of each HPFQI($\mathcal{K}_1$) persistence and PFQI test P&L at the same persistence.

Figure 7.18 suggests that at the first hierarchy step, *i.e.* when HPFQI algorithm passes from $k = 10$ to $k = 5$, the situation in terms of profit obtained in test remains unaltered with respect to the standard PFQI at persistence equal to 5. Conversely, at the second



(a) Persistence = 1 of HPFQI($\mathcal{K}_1$)



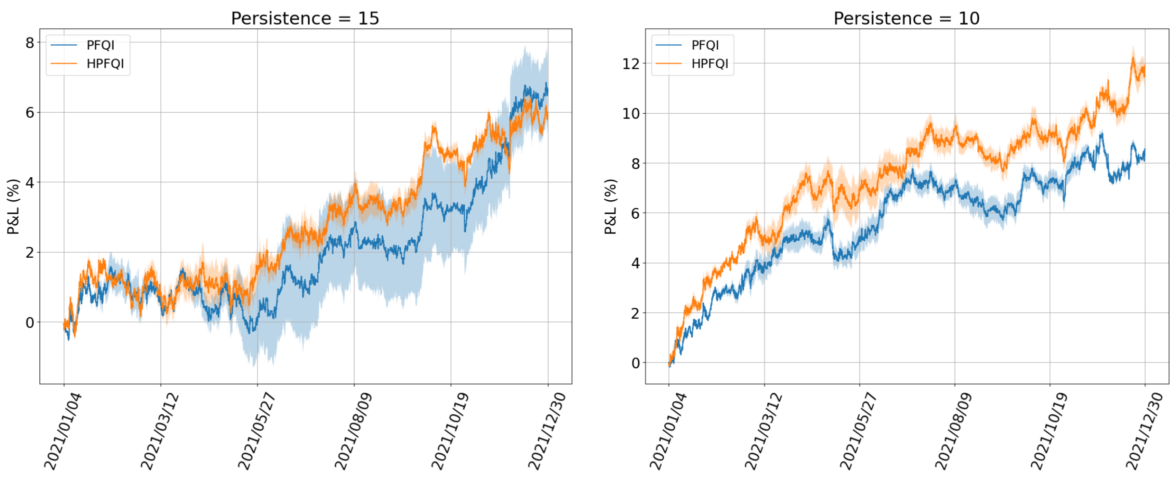
(b) Persistence = 1 of PFQI

Figure 7.19: Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 1 (left) and PFQI at persistence 1 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.

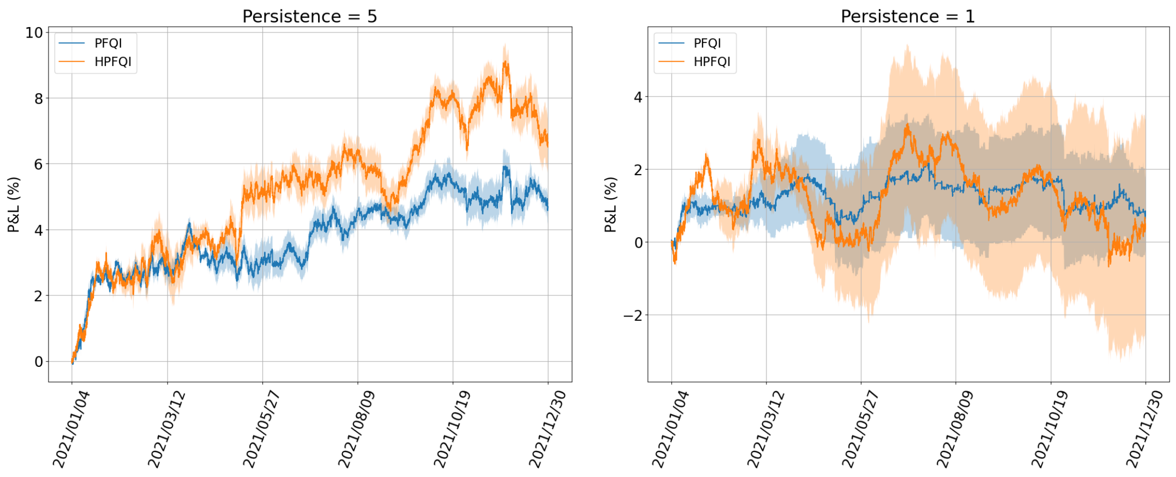
step, *i.e.* at persistence equal to 1, HPFQI clearly outperforms PFQI trained and tested at that persistence. Here we see two novelties compared to Section 7.2.3: firstly, where previously in the first hierarchical step performance always improved, here it follows the trend of PFQI. Secondly, where before, once various steps had been taken and/or especially once unit persistence had been reached, performance got progressively worse, here it apparently gets better.

However, it is interesting to compare the actions taken by HPFQI and PFQI at persistence equal to 1, helped by Figure 7.19. In this particular case, HPFQI obtains a higher profit just by changing action less often than PFQI. Looking at this graphics, it occurs to us HPFQI chooses wisely when to buy or to sell and holds that position for longer. In addition, HPFQI's variance is significantly lower than PFQI's: this may be caused by the fact that HPFQI's policy has become almost entirely deterministic. Nevertheless, this behaviour is definitely strange if compared with Section 7.2.3. Probably, both PFQI and HPFQI are forced not to hold any position often because their training set was not enough explanatory of the situation they would have encountered in test.

Now we are curious to see if the same thing happens with HPFQI($\mathcal{K}_3$).



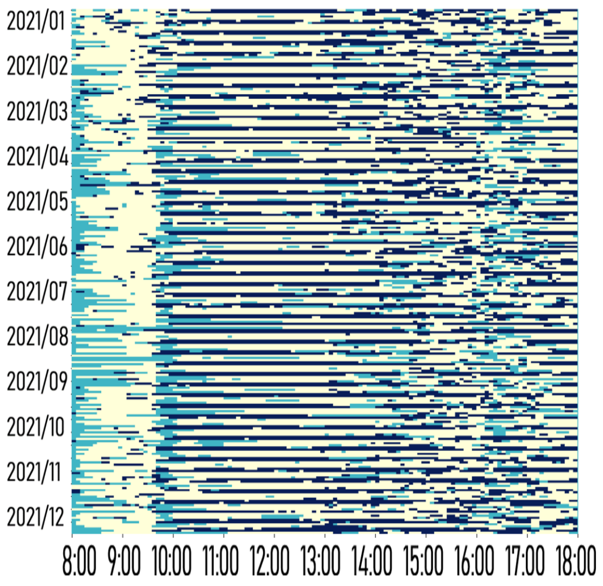
(a) (b)



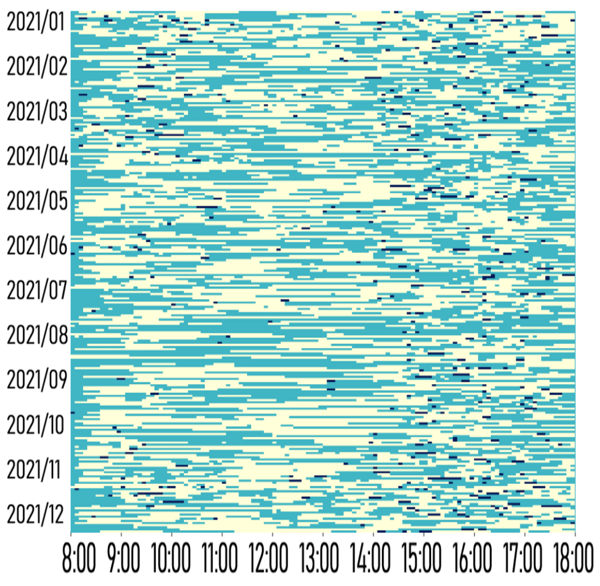
(c) (d)

Figure 7.20: Comparison between the test P&L of each HPFQI($\mathcal{K}_3$) persistence and PFQI test P&L at the same persistence.

As we can see from Figure 7.20, HPFQI behaviour comes closest to the one of Section 7.2.3 than the one displayed in Figure 7.18. Indeed, although at the first hierarchical step HPFQI follows PFQI trend, at persistences 10 and 5 it gains a higher profit than PFQI at the respective persistences. Once reached persistence equal to 1, however, its performance worsen drastically in terms of both profit and variance.



(a) Persistence = 5 of HPFQI($\mathcal{K}_3$)



(b) Persistence = 5 of PFQI

Figure 7.21: Actions taken in 2021 by HPFQI($\mathcal{K}_3$) at persistence 5 (left) and PFQI at persistence 5 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells.

Figure 7.21 shows us the benefits of the hierarchical algorithm: unlike PFQI, which struggles to structure well its policy, HPFQI manages to find both hourly and daily patterns. In addition, HPFQI agent is more induced to buy.

8 | Conclusions and Future Developments

In this thesis, we have implemented a novel batch Hierarchical Reinforcement Learning algorithm, called Hierarchical Persistent Fitted-Q Iteration (HPFQI), to train an artificial agent to autonomously find profitable trading opportunities in the FX market based on quoted exchange rates observed in the last 60 minutes. This work was built upon the preliminary results presented in [38], where the Persistent Fitted Q-Iteration (PFQI) algorithm was applied to FX trading. The main contributions made in this thesis with respect to the benchmark are the followings:

- the introduction of a new hierarchical reinforcement learning algorithm (HPFQI) that allows the agent to encapsulate both lower and higher-frequency information. This aspect is crucial because the former facilitate the understanding of the effect of an action and the latter give more control to the agent;
- the detailed quantitative inspection of the data used for training and testing our algorithm, with the aim of being more confident in the results obtained;
- the implementation of the XGBoost algorithm as the regression method used to approximate the action-value function, a measure of the quality of an action, at each PFQI and HPFQI iteration. XGBoost reduces the computational time and increases the accurateness of the estimation.

HPFQI was designed to make sure that an agent can exploit trend information at low frequencies while maintaining high control over its actions. This was done by exploiting a hierarchy of auxiliary algorithms, APFQI, trained sequentially each one at a different control frequency. Specifically, at each hierarchical step, the trained agent can choose whether it is better to use the policy generated at the current step or to exploit those obtained at previous frequencies.

In order to test the novel algorithm, we first worked on Mountain Car, an OpenAI gym environment typically used to evaluate reinforcement learning algorithms. We did so mainly because it is a less complex setting than FX. In this environment we have had mixed results: it is evident that the passage of information between the various persistences had taken place but the overall performance deteriorates as the number of iterations performed by the algorithm increases. Therefore, we carried out more in-depth analyses, which showed that the performance drops are probably caused both by the fact that the action-value function struggles to adapt to the many new states that are introduced and by the overestimation of the action-value function after many iterations of APFQI are performed.

Although these results were not too positive as a whole, we believed that the FX environment might be a better evaluation task for HPFQI given that sampling is not necessary as we use historical data and the rewards are not sparse as in Mountain Car. Before getting the results, however, we decided to do some analyses on the specific EUR/USD dataset to see if the assumptions we rely on are verified. Specifically, although the autocorrelation evaluated on the returns showed a pattern that could have violated the stationarity hypothesis it was shown however to not give lucrative information to the agent.

Drawing on these results, we tested our algorithm on the task of FX trading. Training and testing HPFQI on different frameworks, we have noticed how, in general, the hierarchical algorithm is able to exploit information from lower frequencies to learn a more structured policy with respect to PFQI. In several cases, this ability resulted in a clear improvement in the profit obtained in the test set. However, if the hierarchical chain becomes too long, the policy often becomes over-structured, generating profits even lower than the ones deriving from the less organized PFQI policy.

8.1. Future Developments

Although for the same accuracy of the approximation of the action-value function HPFQI introduces less overestimation than PFQI thanks to the higher persistences, the overestimation could be reduced further. One way to do so is to use the Double Q-Learning method within the hierarchical algorithm. Indeed, this approach has been already used in [38] and it has been shown to make noteworthy improvements.

Another approach that might be useful for the purpose of improving the performance of our algorithm is to change the way the hierarchy is developed: in fact, in the case presented in this thesis, information from previous frequencies was passed by transferring the action-value function. However, as we noticed especially in the Mountain Car environment, in some cases the action-value function may struggle to adapt to new states. For this reason, another way to address the problem might be not to transfer the action value function, but rather to sequentially increase the dimensionality of the state by gradually including in it variables containing information from previous frequencies (e.g., actions that have been taken or something more sophisticated).

Bibliography

[1] S. Ahilan and P. Dayan. Feudal multi-agent hierarchies for cooperative reinforcement learning. 2019.

[2] B. An, S. Sun, and R. Wang. Deep reinforcement learning for quantitative trading: Challenges and opportunities. *IEEE Intelligent Systems*, 37(02):23–26, 2022. ISSN 1941-1294.

[3] P.-L. Bacon et al. The option-critic architecture. *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, pages 1726–1734, 2017.

[4] M. Baykal-Gürsoy. *Semi-Markov Decision Processes.* John Wiley & Sons, Ltd, 2011.

[5] A. Biedenkapp, R. Rajan, F. Hutter, and M. Lindauer. Temporl: Learning when to act. *CoRR*, abs/2106.05262, 2021.

[6] H. Chen and V. Krovi. Policy search for path planning of an autonomous ground vehicle with deep reinforcement learning. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3505–3512. IEEE, 2016.

[7] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[8] T. Chen, Y. Zou, S. Chen, H. Ye, R. Zhang, Z. Wang, and H. Hu. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[9] T. Chen, M. Everett, and J. P. How. Learning shared control for autonomous driving with mixed-autonomy traffic. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4858–4864. IEEE, 2019.

[10] X. B. Chen, Y. Wang, and C.-Y. Chen. A survey on reinforcement learning for robotics: From cold-start to sim-to-real transfer. *Information Sciences*, 512:85–105, 2020.

[11] O. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, page 816–823. Association for Computing Machinery, 2005.

[12] W. Dabney, M. Rowland, R. Munos, and G. Ostrovski. Exploration by distributional rl with application to the mountain car problem. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, pages 1230–1239, 2018.

[13] C. Daniel, H. van Hoof, J. Peters, and G. Neumann. Probabilistic inference for determining options in reinforcement learning. *Machine Learning*, 104:337–357, 2016.

[14] P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Proceedings of the 5th International Conference on Neural Information Processing Systems*, page 271–278. Morgan Kaufmann Publishers Inc., 1992.

[15] M. P. Deisenroth, D. Fox, and C. E. Rasmussen. Gaussian processes for data-efficient learning in robotics and control. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4304–4311. IEEE, 2015.

[16] T. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *The Journal of Artificial Intelligence Research (JAIR)*, 13, 2000.

[17] X. Feng, J. Guo, B. Qin, T. Liu, and Y. Liu. Effective deep memory networks for distant supervised relation extraction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4002–4008, 2017. doi: 10.24963/ijcai.2017/559. URL `https://doi.org/10.24963/ijcai.2017/559`.

[18] Y. Gao, Z. Gao, Y. Hu, S. Song, Z. Jiang, and J. Su. A framework of hierarchical deep q-network for portfolio management. In *International Conference on Agents and Artificial Intelligence*, 2021.

[19] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63:3–42, 2006.

[20] M. Ghavamzadeh, S. Mahadevan, and R. Makar. Hierarchical multi-agent reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 13:197–229, Sept. 2006.

[21] S. Gu, M. Ghaffari, A. Almogahed, W. Niu, and H. Yu. Deep reinforcement learning for autonomous vehicles: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 18(11):2961–2972, 2017.

[22] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, M. Hutter, and V. Koltun. Con-

trol of a quadrotor with reinforcement learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2762–2769. IEEE, 2017.

[23] M. King and X. Wang. High-frequency trading in the foreign exchange market. *Journal of International Money and Finance*, 67:383–406, 2016.

[24] G. Konidaris and A. Barto. Building portable options: Skill transfer in reinforcement learning. *Learning*, pages 1–13, 01 2006.

[25] G. Konidaris and A. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. page 1015–1023. Curran Associates Inc., 2009. ISBN 9781615679119.

[26] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, page 3682–3690. Curran Associates Inc., 2016.

[27] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. In *Proceedings of the 29th International Conference on Neural Information Processing Systems*, pages 1339–1347, 2016.

[28] C. Luo, H. Zhang, H. Sun, and B. Zhu. Reinforcement learning for autonomous driving. *IEEE Transactions on Neural Networks and Learning Systems*, 29(12):5614–5629, 2018.

[29] G. Mazzolini. Action persistence, a way to deal with control frequency in batch reinforcement learning. Master thesis, Politecnico di Milano, 2020.

[30] A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, page 361–368, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[31] I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. 2002.

[32] A. M. Metelli, F. Mazzolini, L. Bisi, L. Sabbioni, and M. Restelli. Control frequency adaptation via action persistence in batch reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare,

A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[34] O. Nachum, S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. page 3307–3317. Curran Associates Inc., 2018.

[35] F. Nashashibi and B. Chaib-draa. Mountain car problem: solution with q-learning algorithm. In *IEEE International Symposium on Intelligent Control*, pages 303–308. IEEE, 1997.

[36] X. Pan, H. You, Z. Wang, C. Lu, Y. Wang, C. Sun, and J. Tang. Virtual to real reinforcement learning for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5888–5897, 2017.

[37] S. Pateria, B. Subagdja, and A.-H. Tan. Multi-agent reinforcement learning in spatial domain tasks using inter subtask empowerment rewards. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 86–93, 2019.

[38] A. Riva, L. Bisi, P. Liotet, L. Sabbioni, E. Vittori, M. Pinciroli, M. Trapletti, and M. Restelli. Learning fx trading strategies with fqi and persistent actions. In *Proceedings of the Second ACM International Conference on AI in Finance*, New York, NY, USA, 2022. Association for Computing Machinery.

[39] S. Sun, W. Xue, R. Wang, X. He, J. Zhu, J. Li, and B. An. Deepscalper: A risk-aware reinforcement learning framework to capture fleeting intraday trading opportunities. In *Proceedings of the 31st ACM International Conference on Information &amp; Knowledge Management*, page 1858–1867, New York, NY, USA, 2022. Association for Computing Machinery.

[40] P. A. Sunehag, G. Lever, A. Gruslys, and D. Silver. Efficient exploration in reinforcement learning using relative novelty. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3309–3318. JMLR. org, 2017.

[41] K. Suri, X. Shi, K. N. Plataniotis, and Y. A. Lawryshyn. Trader: Practical deep hierarchical reinforcement learning for trade execution. *ArXiv*, abs/2104.00620, 2021.

[42] R. S. Sutton and A. G. Barto. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 7(3):217–237, 1981.

[43] R. S. Sutton and A. G. Barto. Reinforcement learning with replacing eligibility traces. In *Proceedings of the 15th International Conference on Machine Learning (ICML-98)*, pages 330–338. Morgan Kaufmann, 1998.

[44] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 1998.

[45] R. S. Sutton and S. P. Singh. Reinforcement learning with adaptive state aggregation. In *Advances in neural information processing systems*, pages 237–243, 1996.

[46] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112:181–211, 1999.

[47] C. Tallec, L. Blier, and Y. Ollivier. Making deep q-learning methods robust to time discretization. In *International Conference on Machine Learning*, 2019.

[48] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. page 3540–3549. JMLR.org, 2017.

[49] J. Wang, F. Gao, B. Kiumarsi, Z. Sun, and L. Dong. Control of nonlinear systems with deep reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(6):1698–1711, 2019.

[50] R. Wang, H. Wei, B. An, Z. Feng, and J. Yao. Commission fee is not enough: A hierarchical reinforced framework for portfolio management. In *AAAI Conference on Artificial Intelligence*, 2020.

[51] Z. Wang, B. Huang, S. Tu, K. Zhang, and L. Xu. Deeptrader: A deep reinforcement learning approach for risk-return balanced portfolio management with market conditions embedding. In *AAAI Conference on Artificial Intelligence*, 2021.

[52] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[53] F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[54] Q. Zhang, X. Wang, F. Gao, R. Yu, Z. Sun, and L. Dong. A survey on deep reinforcement learning for autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 22(2):760–776, 2020.

[55] W. Zhang, Z. Guo, L. Li, and H. Yu. Model-free reinforcement learning for control: Survey, taxonomy, and future directions. *IEEE Transactions on Industrial Informatics*, 16(6):3882–3892, 2020.

[56] T. Zhao and M. Gowayyed. Algorithms for batch hierarchical reinforcement learning. *CoRR*, abs/1603.08869, 2016.

[57] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6728–6737, 2017.

List of Figures

2.1 The agent-environment interaction in a Markov Decision Process (Source: [44]). 8
2.2 Agent-environment interaction without (top) and with (bottom) action persistence, highlighting duality. The transition generated by the k-persistent MDP $\mathcal{M}_k$ is the cyan dashed arrow, while the actions played by the k-persistent policy are inside the cyan rectangle (Source: [32]). 23
2.3 Evolution of XGBoost algorithm from Decision Trees (Source: https://towardsdatascience.co

5.1 PFQI test performance for each persistence. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions and null velocity). For each persistence, ten models with different seeds are used. Therefore, for each persistence, the bold line represents the average expected return among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the PFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2). 50
5.2 PFQI test performance for each persistence with initial datasets of equal length ($n_{eq} = 1000$). Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions and null velocity). For each persistence, ten models with ten different seeds are used. Therefore, for each persistence, the bold line represents the average expected return among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the PFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2). . . . 51

5.3 HPFQI test performance with $k \in \mathcal{K} = \{32, 16, 8, 4\}$. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions and null velocity). Ten HPFQI models with ten different seeds are used. Therefore, for each persistence, the bold line represents the average expected return of APFQI among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the APFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2). 52
5.4 Comparison between HPFQI and PFQI test performances for each persistence. HPFQI characteristics are the same as in Figure 5.3, whereas PFQI ones come from Figure 5.1. 53
5.5 Kernel density representation of the action-value function at some iterations of HPFQI. 56
5.6 States visited by the agent during HPFQI training at the second iteration of different persistences. The second iteration is chosen because it is the first iteration in which the target compares the action-value function generated by itself with the action-value function obtained at the previous persistence. The different colors highlight whether in the state the maximum has been reached by the previous action-value function (light-blue) or by the current one (blue). 57
5.7 Percentage use of the action-value function of previous persistences in HPFQI. The percentage over all the states is displayed in blue, whereas the percentage over the new states only is represented in orange. 58
5.8 Kernel density representation of the action-value function of HPFQI. . . . 59
5.9 Kernel density representation of the action-value function at the last iteration of each persistence of HPFQI. 60
5.10 PFQI test performance for each persistence with initial datasets of equal length ($n_{eq} = 1000$) and random both initial position and velocity. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes. For each persistence, ten models with different seeds are used. Therefore, for each persistence, the bold line represents the average expected return among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the PFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2). 61

5.11 HPFQI test performance with $k \in \mathcal{K} = \{32, 16, 8, 4\}$. Specifically, the test performance is the expected return obtained averaging the returns of 20 different episodes (*i.e.* with different starting positions). Ten HPFQI models with ten different seeds are used. Therefore, for each persistence, the bold line represents the average expected return of APFQI among the seeds, whereas the shaded color band indicates one standard deviation interval. The x-axis represents the APFQI iteration step multiplied by the value of the persistence (we take our cue from the reasoning carried out in the Section 5.2.2). 62
5.12 Comparison between HPFQI and PFQI test performances for each persistence with random both initial position and velocity. HPFQI characteristics are the same as in Figure 5.11, whereas PFQI ones come from Figure 5.10. 63

6.1 Autocorrelation function over the process defined in (6.8), considering HistData data of 2018 and 2019. In the x-axis we find the lag value and on the y-axis the value of the autocorrelation itself. 74
6.2 The first 20 lags of the autocorrelation function over the random variables defined in (6.8) for HistData (left) and Intesa (right) data over dates approximately from April 2022 to December 2022. 75
6.3 Autocorrelation function over the process defined in (6.8) with the total amount of HistData data over a specific period of 2022. In the x-axis we find the lag value and on the y-axis the value of the autocorrelation itself. . 76
6.4 The blue dots represent the autocovariance of the process made of random variables as defined in (6.8) evaluated in the first 20 lags. The red line, instead, represents the average value of the bid-ask spread. 77
6.5 Mean volatility (V_t) values evaluated on HistData data over 2018 and 2019. The time labels refer to the CET (Central European Time) time zone. . . . 78

7.1 EUR/USD exchange rate on 2020. 81
7.2 PFQI cumulative P&L on 2021 at each persistence when training is done on 2018/2019 (first figure) or on 2019/2020 (second figure) over three different seeds. For each persistence, the bold line represents the average cumulative P&L among the seeds, whereas the shaded color band indicates one standard deviation interval. 83
7.3 EUR/USD exchange rate on 2018/2019 (first figure), on 2019/2020 (second figure) and on 2021 (third figure). Respectively, they represent the training set of PFQI$_1$, the training set of PFQI$_2$ and the test set of both. 84

7.4 Actions taken in 2021 by PFQI$_1$ (left) and PFQI$_2$ (right) at persistence equal to 5. Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = -1 (yellow) means that the agent sells. 85
7.5 Actions taken in 2021 by PFQI$_1$ at persistence 1 (left) and PFQI$_1$ (right) at persistence equal to 30. Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = -1 (yellow) means that the agent sells. 86
7.6 HPFQI($\mathcal{K}_1$) test results with training in 2018/2019. 87
7.7 HPFQI($\mathcal{K}_2$) test results with training in 2018/2019. 88
7.8 HPFQI($\mathcal{K}_3$) test results with training in 2018/2019. 89
7.9 HPFQI($\mathcal{K}_1$) test results with training in 2019/2020. 90
7.10 HPFQI($\mathcal{K}_3$) test results with training in 2019/2020. 91
7.11 Comparison between the test P&L of each HPFQI($\mathcal{K}_1$) persistence and PFQI test P&L at the same persistence. 91
7.12 Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 5 (left) and PFQI at persistence 5 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = -1 (yellow) means that the agent sells. 92
7.13 Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 1 (left) and PFQI at persistence 1 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = -1 (yellow) means that the agent sells. 93
7.14 Comparison between the test P&L of each HPFQI($\mathcal{K}_2$) persistence and PFQI test P&L at the same persistence. 94
7.15 Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 5 (left) and HPFQI($\mathcal{K}_2$) at persistence 5 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = -1 (yellow) means that the agent sells. 95
7.16 Comparison between the test P&L of each HPFQI($\mathcal{K}_3$) persistence and PFQI test P&L at the same persistence. 96

7.17 Actions taken in 2021 by HPFQI($\mathcal{K}_3$) at persistence 1 (left) and PFQI at persistence 1 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells. 97
7.18 Comparison between the test P&L of each HPFQI($\mathcal{K}_1$) persistence and PFQI test P&L at the same persistence. 97
7.19 Actions taken in 2021 by HPFQI($\mathcal{K}_1$) at persistence 1 (left) and PFQI at persistence 1 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells. 98
7.20 Comparison between the test P&L of each HPFQI($\mathcal{K}_3$) persistence and PFQI test P&L at the same persistence. 99
7.21 Actions taken in 2021 by HPFQI($\mathcal{K}_3$) at persistence 5 (left) and PFQI at persistence 5 (right). Each row represents a day from 8:00 CET to 18:00 CET. Action = 1 (dark blue) means that the agent buys, action = 0 (light blue) means that the agent stays flat, whereas action = −1 (yellow) means that the agent sells. 100

List of Tables

5.1 Final parameters for each persistence obtained after Extra-Trees (first batch or rows) and PFQI (second batch of rows) parameters fine-tuning. 49

7.1 Final parameters for each persistence obtained after XGBoost (first batch or rows) and PFQI (second batch of rows) parameters fine-tuning. 82