POLITECNICO DI TORINO

Master Degree course in Artificial Intelligence and Data Analytics

Master Degree Thesis

Reinforcement learning algorithms for optimizing real-time purchase and sale of energy produced by a renewable energy plant

Supervisors
Prof. Bruno Giuseppe Averta
Dott. Tullio Re

Candidate
Fabio Grillo

Academic Year 2022-2023

Abstract

This thesis investigates the use of reinforcement learning algorithms for optimizing the purchase and sale of energy produced by a renewable energy plant in real-time. The study explores the effectiveness of different reinforcement learning algorithms in achieving optimal energy trading strategies, considering different environmental and economic factors. The goal is to build an environment as close to reality as possible and investigate the use of algorithms that can process and refine trading techniques. The study is expected to provide valuable insights into the potential of using intelligent algorithms to optimize trading in intraday electricity markets.

Contents

Chapter 1

Introduction

This thesis arises from the need to deepen some topics covered in the courses of Politecnico di Torino and from the desire to put into practice what I have learned on real cases, trying to find solutions with a well-founded and reasonable scientific basis through an accurate and in-depth study; tying together my skills, my passions and my ability to look for solutions to unsolved problems.
In recent years there has been a growing interest in renewable energy as an alternative to traditional fossil fuels. This interest is due to the ability of renewable energies to act on global warming and reduce all the problems that traditional fossil sources procure. In this sense they prove to be a viable solution to many problems in modern society.
However, even these have characteristics that need careful and careful analysis in order to be deployed in our energy system: variability and intermittency, which are at the root of the challenges that arise in renewable sources. Specifically, this thesis aims to analyze their environment from the perspective of the electricity market, studying and evaluating the best approaches for buying and selling energy through Reinforcement Learning algorithms.
Reinforcement Learning is a branch of machine learning that allows an agent to learn and make decisions based on a reward signal. Adapting it to the electricity market problem, specifically the reward signal consists of the profit earned from selling into the grid. With different Reinforcement Learning approaches we therefore aim to study how to optimize sales and purchases of energy produced by renewable energy plants given energy demand and price fluctuations in the market while maximizing profit and therefore reducing environmental impact. The study will generally proceed in exploring optimal trading techniques and strategies considering several variables. With such a study we expect to answer many questions that arise from an algorithmic and therefore data science point of view.
In conclusion, the purpose of this thesis is to investigate, adapt and optimize the current techniques that are still deployed in energy trading in the market today and, where possible, provide valuable insights in parametric terms for the deployment of Reinforcement Learning approaches to this type of problem.

1.1 Intraday Electricity Market

We now want to focus on the detailed study and analysis of the market in which we are going to carry out our analysis; dissecting its technical and theoretical aspects, so as to understand in more detail its trading methodologies and related issues, in order to have a more complete view of our problem and the correct perspective of resolution through the Reinforcement Learning technique.
First, it is necessary to give a definition of the continuous market in which we are going to operate: essentially intraday continuous markets have the particularity of referring to a market in which products - in our case volumes of electricity in MWh - are bought and sold on the same trading day.
Since it is possible to make buying and selling transactions at instants very close together in terms of time, the goal of intraday trading is to make profit on frequent and sometimes small price movements. This type of market sets the stage for a fairly different type of trading than other types of markets, which may include longer-term trading such as swing trading or position trading, where trades are held for days, weeks or even months. These types of trading strategies focus on larger price movements over a longer period of time. So it is clear from the outset that this peculiarity will push us in a clear and defined direction, as related strategies differ markedly from market to market. An additional key aspect to consider is risk management, as it is necessary for the trader to be aware at each moment in time of what the losses and gains are so as to unearth the ideal conditions for generating buy and sell transactions. The reader should keep in mind that there is no "deterministic" behavior in the market, so the same contextual conditions will not necessarily establish a given price of electricity in terms of MW/h, but, nevertheless, these will be an indication of the fluctuation of the price, which are fundamental data for traders.
Nowadays there are disparate strategies and tools for market analysis and trading: starting with the most common tools that include candlestick patterns, chart patterns, indicators and trade setups; other techniques include swing lows, money flow index, volume weighted average price, moving averages and so on. However, all of these strategies are not without problems that limit their effectiveness, leaving open space for research on the subject.
There are several challenges and problems associated with current trading techniques and that is also why we want to try to use a new paradigm outside the traditional patterns. The purpose of this thesis fits into this space, attempting to solve the problem from the perspective of the paradigm called "Reinforcement Learning".
Before proceeding further, however, it is still necessary to define what the Limit Order Book (LOB) is and how it works.
In fact, the data required for future analyses are generated and regulated by a tool that is as simple as powerful: the Limit Order Book. This allows us to reconstruct the state of the market and its evolution over time, through the orders submitted by various agents and the transactions generated. The importance of the LOB also lies in its encapsulation of price volatility, a fundamental characteristic for intraday market analysis. As we shall see later, the construction of such an object will not be without any problems and implementation choices, which in one way or another will partly determine the results

obtained. For now, it is necessary to think that the paradigm we want to use will rely on this fundamental object, which will make it possible to determine a logic both on the price of each buy/sell order and on the volume of the order itself.
As a further step, it is necessary to define what "trading" is, what regulates it and how it is done.
It is possible to think of trading as the activity of buying and selling goods of any kind or services in exchange for money or any kind of asset. Certainly the best known nowadays is trading on stock markets, commodity markets and currency markets, but in recent times trading in cryptocurrencies has become widespread and expanded. Because trading is a complex and risk-taking activity, it is necessary for those who do it, the traders, to follow a strategy in order to pursue their objectives, depending on the risk itself, the inputs and outputs. Traders need to know very well the conditions of the type of market on which they are trading, its specific characteristics, trends, news or any other main features that allow them to outline a strategy of one type or another. Last but not least, traders must reflect certain characteristics including, above all, patience, which makes them adept at dealing with the most diverse situations in the evolution of any market.
In conclusion, we would like to point out that the focus of this thesis is not on trading techniques, nor at least on the functioning of continuous markets: however, it is however necessary to have provided a concise explanation and as exhaustive as possible in order to put the reader in a position to be able to fully understand the topics that we will then treat.

1.2 Reinforcement Learning Paradigm

In order to take a data-driven approach to the problem of trading in continuous energy markets we now go on to describe what will be the tool that will enable us, given its characteristics, to trade electricity from a new perspective.
As described by [20] the idea of interacting with an environment is probably the first one that comes to mind when we think about the nature of learning: this is generally true for anyone who has to learn to perform an action, more or less complex, that will produce consequences on the surrounding environment and that will change the state in which that subject is within it. For example, if we think of a child, he will learn to talk, play, and walk through the same feedback that the environment returns to him through his sensory organs. Hence the need to create a paradigm that helps in the resolution of complex problems through the implementation of this concept that may seem trivial, but at the same time is extremely powerful. Indeed, it is well known how through this simple idea of continuous interaction between an agent and its respective environment it is possible to deal with different kinds of problems with high efficiency. This is the origin of what, from here on, we will call Reinforcement Learning, which is an approach that breaks away from the well-known Supervised Learning and Unsupervised Learning and implements a logic of operation that we will later go on to explain, both at the theoretical level and in terms of its application in the field of the renewable energy market. The key concept is simple: RL problems involve trying to find a mapping between actions and states that maximizes reward. In this way the agent is never told what action to take, but

rather will learn to recognize and do the action that achieves in the best overall reward.
In summary, what needs to be accomplished is: to find a mathematical abstraction that can represent well the different aspects of the problem, through the description of the environment, the agent, the states in which it may be in the environment, and the actions it may take.
Shifting the focus to our problem, we will try to use the Reinforcement Learning paradigm to enable an agent to trade energy volumes in terms of MWh in order to make a profit from these trades. The agent will be identified with the one who will enter orders to buy and sell in the market, the performable actions of the agent will be precisely the new orders to be entered, and the reward will be outlined based on what is to be gained, which as already specified, will correspond, in part, to profit. Once this is done, it will be useful to evaluate the ways in which the agent has taken certain actions, depending on various parameters and factors.
Before proceeding further, it is necessary to understand what the fundamental mechanism governing RL is: it is in fact the so-called "policy." In purely qualitative terms by that term we mean the strategy that an agent follows to do its actions, the policy governs how the agent chooses to do one action rather than another so that it can maximize the final cumulative reward it will get.
In order to get an idea of how this seemingly complex gear works, one can think of Pavlov's conditioning experiment on dogs. He was a physiologist who studied the mechanism of dogs' salivation in the presence of food. He noticed that his dogs salivated not only when they saw the food, but also when they heard the sound of a bell preceding its administration. Pavlov hypothesized that dogs had learned to associate the bell with food, and to test this hypothesis, he conducted several experiments that tracked dogs' salivation under different conditions. He soon realized that the dogs salivated even just at the sound of the bell, even if the food was not subsequently administered: they had associated the food itself with the sound of the bell, responding to this stimulus with salivation. Compared to the RL, dogs are the agent who has learned to perform actions and through the states of the environment have associated the ringing of the bell with a given situation, a given behaviour, salivation. This metaphor is not intended to be exhaustive in gathering the information necessary to explain Reinforcement Learning in its entirety, but rather a concept that encompasses its fundamental features.
We will give a more detailed explanation of each component and its role later, but at the moment define in general what the problem is and how we will try to deal with it.

1.3 Agent-Environment Interface

Let's now turn to a description of the actors that make up the Reinforcement Learning paradigm more in detail.

Following the image 1.1, it is possible to note:

- Agent: the one who takes actions and makes decisions
- Environment: formed by everything that remains outside the agent, including the things which it interacts with

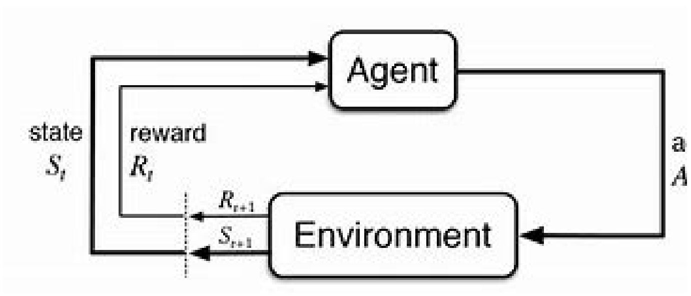


Figure 1.1. Example model of agent-environment interaction

- State: the set of parameters that describe the environment in detail, it also corresponds to what is changed by the actions of the agent
- what is offered to the agent as a result of the action taken; this can be chosen according to the problem addressed

A further important aspect defining agent-environment interactions is given by time t. As described in [20], at each time step t, the agent has a clear view of the state in which he is in function of the environment, $S_t \in \mathbf{S}$, where $\mathbf{S}$ is the set of possible states, and on that basis selects an action, $A_t \in \mathbf{A}(S_t)$, where is the set of actions available in state S_t. As a consequence for the action made the agent receives a reward signal, in practice a real positive, negative or zero number, $R_{t+1} \in \mathbf{R}$, and finds itself a new state, S_{t+1}. Conceptually we can imagine that the agent builds its own mapping from the states to the probabilities of selecting every possible action; this mapping is called the agent's *policy* and is denoted π_t, where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. The ultimate goal of the agent is to maximize the total amount of rewards obtained during each training interaction with the environment.
In conclusion, we want to learn how to take decisions, or rather actions, by consulting states that can be helpful in this process.
As we can imagine, it is possible to work within these well-defined rules in order to structure our working-point and apply it to a custom problem; but to do so, we need to define a few more points.
What we want to achieve is that after training the agent learns to take actions independently given the environment, maximizing the aforementioned reward. Now we ask ourselves: how does the agent train? Well, to be able to answer it is necessary to define two key concepts: "exploration" and "exploitation". Imagining that at each time step the agent has to choose which action to take, he can try to take one trying to maximize the reward given his knowledge up to that specific moment, or he can try every now and then to take a random one in order to verify the the existence of new, more profitable roads, i.e. which will lead to higher rewards; here, these are the ideas that underlie exploration and exploitation: the first respectively indicates the tendency to try to undertake actions for which the reward to which they will lead is not known, which may prove to be higher or lower than to that which would have been obtained by following the current knowledge, which is the second way previously indicated.

These concepts will come in handy later when we discuss agent training and possible strategies that can be followed.
Now that we have broadly defined the RL paradigm ann its key concepts, we want to pay attention to the problem described in 1.1 and ask how this approach will help us address the challenges posed at the heart of the intraday electricity market.

1.4 Finite Markov Decision Process and Value Functions

The last, but not least, foundational concept we need to address before diving into our problem related to intraday continuous energy markets is the so-called Markov Desion Process (MDP).
Returning to the concepts first explained, as cited in [20], we have an agent that must learn to do actions and an environment that returns rewards to the agent. At each timestep the agent will have to make its own mapping from states to probabilities to select each possible action: this mapping is called policy and is denoted by π_t. There are different methods for finding optimal or near-optimal policies, such as value iteration, policy iteration, Q-learning, and SARSA.
The golden rule that must be kept well in mind is that everything that cannot be arbitrarily changed by the agent is part of the environment; therefore, the agent's goal is formalised in terms of a special reward pass from the environment to the agent. The agent's goal will be to maximise the cumulative reward, not the immediate reward. And in this regard, it should be remembered that the reward is conceptually about 'what' is to be achieved, not 'how'. This apsect will undoubtedly be discussed and dealt with in more detail later, as it will prove decisive in the learning of our agent.
What we would like is a framework that makes us able to model sequential decision-making problems: the Markov Decision Process.
An MDP consists of four elements: a set of states, a set of actions, a transition function, and a reward function. A state is a representation of the agent's situation in the environment. An action is a choice that the agent can make to change its state. A transition function defines the probability of moving from one state to another given an action. A reward function assigns a numerical value to each state-action pair, indicating how desirable it is for the agent. The state conveys all the information available to the agent and is given by some pre-processing systems that are part of the environment. It does not need to be informed of everything, but only of what is necessary to make decisions: in broad strokes, one would reanalyse past experiences in order to retain the most important information; in this sense, a state capable of doing this is a Markov state.
In the most general case one would should think that the response from the environment at time $t+1$ may depend on everything that has happened earlier and this way we specify the complete probability distribution:

$$\Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}$$

As written in [20], if the state signal has the Markov property, the environment's response at $t+1$ depends only on the state and action representations at t, so the dynamics are:

$$p(s', r|s, a) = \Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\}$$

Given this, we can calculate the expected rewards for state-action-next state triplets:

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

The determining factor is that if the state is a Markov state then the response of the environment at time t+1 will depend solely on the state at time t and on the representation of the action at t. Thanks to this property we will then be able to predict the next state and the expected reward given the current state and action. It must be emphasised that if the state is not a Markov state by definition, it will be possible to approximate it as a Markov state. Therefore, the key aspect to keep in mind is that the Markov property is fundamental because decisions and expected values are a function of the current state only. In addition if the state and action space are finite then we talk about "Finite Markov Decision Process".
For example, if we have a system that can be in one of three states: A, B, or C, and the probability of transitioning from one state to another is given by a matrix P, then we have a Markov state model. The following is an example of a Markov state model with three states and the transition matrix P:

	A	B	C
A	0.6	0.3	0.1
B	0.4	0.5	0.1
C	0.2	0.2	0.6

This means that if the system is in state A, then it has a 60% chance of staying in state A, a 30% chance of moving to state B, and a 10% chance of moving to state C in the next time step. Similarly, if the system is in state B, then it has a 40% chance of moving to state A, a 50% chance of staying in state B, and a 10% chance of moving to state C in the next time step. And so on for state C.

Another example of a Markov state model is a weather system that can be in one of two states: sunny or rainy. The probability of transitioning from one state to another depends on the current weather and not on the previous days. Here is an example of a Markov state model with two states and the transition matrix Q:

	Sunny	Rainy
Sunny	0.8	0.2
Rainy	0.3	0.7

This means that if the weather is sunny today, then it has an 80% chance of being sunny tomorrow and a 20% chance of being rainy tomorrow. If the weather is rainy today, then it has a 30% chance of being sunny tomorrow and a 70% chance of being rainy tomorrow.
Here are some examples where we can have Markov Decision Processes:

- Agriculture: how much to plant based on weather and soil state.
- Water resources: keep the correct water level at reservoirs.
- Chess: choose the best move given the board position and opponent's strategy.

- Robotics: navigate a grid map while avoiding obstacles and reaching a goal.

It is necessary to remember that all these concepts, although momentarily detached from real application, will prove to be fundamental in the following pages in order to understand each choice in its fullness: with this in mind, in fact, we have first chosen to provide a clear and generic explanation at a theoretical level in order to provide a basis for the acquisition of concepts which, in any other way, would be extremely complicated and difficult to digest.
Now we are going to add one more piece and talk about value functions.
Value functions are a key concept in reinforcement learning: they allow to measure the expected long-term reward of being in a certain state or taking a certain action in an environment. They help the agent make optimal decisions by comparing different options and choosing the one with the highest value, or rather, the one with the highest expected future rewards: in a nutshell, they allow one to estimate how good it is for an agent to be in a given state. There are two main types of value functions: state value functions and action value functions. State value functions, denoted by $V(s)$, represent the value of a state s under a given policy π, which is a rule that determines how the agent behaves. It can be expressed like this:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s],$$

where $\mathbb{E}_\pi[G_t|S_t = s]$ represents the expected value of the return G_t at time t, given that the agent is in state s at time t, when following the policy π; $\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ represents the sum of all future rewards, where each reward is discounted by a factor of γ^k. Recall that *gamma* determines the importance of future rewards, where values closer to 0 conceptually place more importance on immediate rewards while those closer to 1 place more importance on future rewards. We will discuss this in more detail later in 3.4. Then we have action value functions, denoted by $Q(s, a)$, represent the value of taking an action a in a state s under a policy π. It is expressed as:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s; A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s; A_t = a]$$

Value functions are a key concept in reinforcement learning. They measure the expected long-term reward of being in a certain state or taking a certain action in an environment. The main difference between the two is that the state-value function only considers the current state, while the action-value function considers both the current state and the action taken in that state. Value functions help agents to make optimal decisions by comparing different options and choosing the one with the highest value.
Value functions can be estimated using various methods, such as dynamic programming, Monte Carlo methods, or temporal difference learning. These methods update the value function based on the observed rewards and transitions from the environment.
The goal is to find the optimal value function that maximizes the expected return for any state or action. In conclusion, value functions are essential for reinforcement learning, as they allow agents to evaluate their situations and actions and learn from their experiences.

Value functions are closely related to policies, which define the agent's behavior in an environment.
Let's move on to the last step. How could we adapt our problem as the number of states increases? In fact it would be unthinkable to keep track of every possible state-action as the number of parameters would grow very quickly, the agent could instead keep the same q_π and v_π as parameterised functions and adjust the observed returns from time to time. From here we can define the Bellman equation for v_π as:

$$\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')],$$

where s' represents the next state of the environment after taking action a in state s, r represents the reward received by the agent after taking action a in state s and $p(s', r|s; a)$ represents the probability of transitioning from state s to state s' and receiving reward r after taking action a.
Solving a reinforcement learning problem means finding a policy that achieves very high rewards in the long run. We ask ourselves then: when is one policy better than another? Simple, when its expected value is greater than the expected value of the other. From here we can obtain the Bellman optimality equation:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s; a)[r + \gamma v_*(s')].$$

This fundamental equation of reinforcement learning explains the relationship between the value of the state at time t and that at t + 1. It provides a way to calculate the function that maximises the expected reward. It is a recursive function and this characteristic allows us to calculate the optimal value through an iterative process in which we start with an estimate of the optimal state-value function and iteratively correct it until convergence is reached.
This equation is extremely powerful because through it we can summarise a concept that is as simple as complex at the same time: that is, that the learning we talked about in the introduction, takes place little by little, through actions that lead any agent to perfect the aim action by action, trying to improve the reward it will get in the end. Hence the reference to the thought of Plato, who argued, already about 2400 years ago, that learning takes place through habit and imitation [12] [13] . The philosopher from Athens had, unwittingly, pioneered the key concept that imitation, in greek $\mu\acute{\iota}\mu\eta\sigma\iota\sigma$, is the reproduction of a model or example provided by someone or something and that the more motivated, interested and satisfied you are with what you learn, the easier and deeper you learn. In these ideas lie the foundations of our concepts of reward, policy and value functions.

Chapter 2

The Trading Environment

Now that we have given an explanation and provided background definitions to our problem, we can explore its internal characteristics and begin to outline a solution path. Since the world of trading has its inherent complexities and electrical markets as well, we will try to stay close and rely on the mechanisms of the paper of [6]: in this way we can have benchmarks that will help us better understand the goodness of our solutions. The idea is to model the intraday market as a Markov Decision Process so that we can use and update the policy based solely on the previous state, greatly simplifying the whole treatment and the way our agent will try to trade profitably.
A key difference, however, lies in the fact that, for a variety of reasons, we will not be able to operate on real data, i.e. data derived directly from national energy markets, and to solve this problem we will try to simulate them as closely as possible to reality, effectively introducing approximations that on the one hand will have the effect of distancing us from reality, but on the other will allow us to perform actions and develop our model in a controlled environment free of risks of any kind.
In this regard, it is worth pointing out in the last instance that in this chapter we will introduce the concept of Brownian motion and see it applied to the world of finance, go deeper into the analysis of the so-called Limit Order Book, discover and analyse the peculiarities of a model such as the "XGBoost" and in conclusion we will introduce the OpenAI Gym which will allow us to provide the state of the environment to our agent and the form of actions he can take to play on the market.

2.1 Geometric Brownian Motion

One of the first needs we encounter along the way to solving the problem, since it is not possible to rely on real data flowing from the market, is to generate energy prices that are in line with the real ones, but this is not enough. It is also necessary to model the random fluctuations of the latter in a manner consistent with reality, in order to have an environment that emulates the real one as much as possible. In this regard, we can use the Geometric Brownian Motion which will allow us to obtain simulated prices and their evolution over time.

By definition, as specified in [19], the Geometric Browninan Motion (GBM) is a particular type of stochastic process that is able to model the random fluctuations of a quantity whose logarithm follows a normal distribution. It is widely used in finance, physics, biology and other disciplines to describe growth or decline phenomena affected by uncertainty. In our case it proves to be useful precisely for modeling the trend of a price over time and we use it in conjunction with the prices and daily volatilities of the Italian market. Indeed, in these terms, since by definition the intraday energy markets allow for the entry of purchase and sale orders at any moment and that such orders will then be processed and finalized using the Limit Order Book mechanism, our agent must be able to create such orders based on a comparison with those present in the LOB, and the latter, in turn, will be generated randomly following the fluctuations given by the average unit price and the average volatility of a given day. To continue the discussion on the GBM, we can say that it is widely used to model the dynamics of stock prices. Under some assumptions the GBM can be derived from the Black-Scholes equation, which is a partial differential equation relating the price of an option to the price of the asset under consideration:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0,$$

where V is the price of the option as a function of the stock price S and time t, r is the risk-free interest rate, and σ is the volatility of the stock‘s returns.
In our case we will have possession of all the average unit prices of energy, per MWh, and the average volatility, from 1/1/2019 to 31/12/2021.
The idea is to obtain the evolution of the price during an entire market day thanks to the GBM and, for the reasons we will see later, having a price every 5 minutes to compare in the LOB, this means having 12 prices per hour. These details, as already specified, will be discussed later.
Before proceeding, the concept of "constant short rate" must also be defined. It is in fact a simple model for risk-neutral discounting that assumes constant interest over time. What does it mean? In our case we will use it to calculate the derivative of the previous differential equation. This derivative represents the expected value on the sock. Basically, by assuming that the risk term remains constant over time, we can greatly simplify the calculations and focus on further aspects. It is important to note that while the constant short rate assumption can be useful for certain applications, it may not always be realistic. In practice, interest rates can change over time due to various economic factors. More sophisticated models, such as stochastic short rate models, can be used to capture this behavior.
At this point we specify that it is not in our interest to delve into the theoretical depths of this topic, but to have the knowledge to interpret our workflow with awareness. Now let’s see how we used the GBM for our specific purposes.
What we really want to achieve is to obtain a list of daily prices that are closest, in terms of performance, to the real ones, considering all the fluctuations that these may have. Of all the simulations that we will obtain later, we will use the average of these to obtain the price associated with a specific unit time slot, which in our case has a duration of 5 minutes. From here we choose to produce 2 simulations starting from 1/1/2019, for each available day. We also use a value equal to 0.08 as the Constant Short Range. The result

obtained is shown in the figure below.

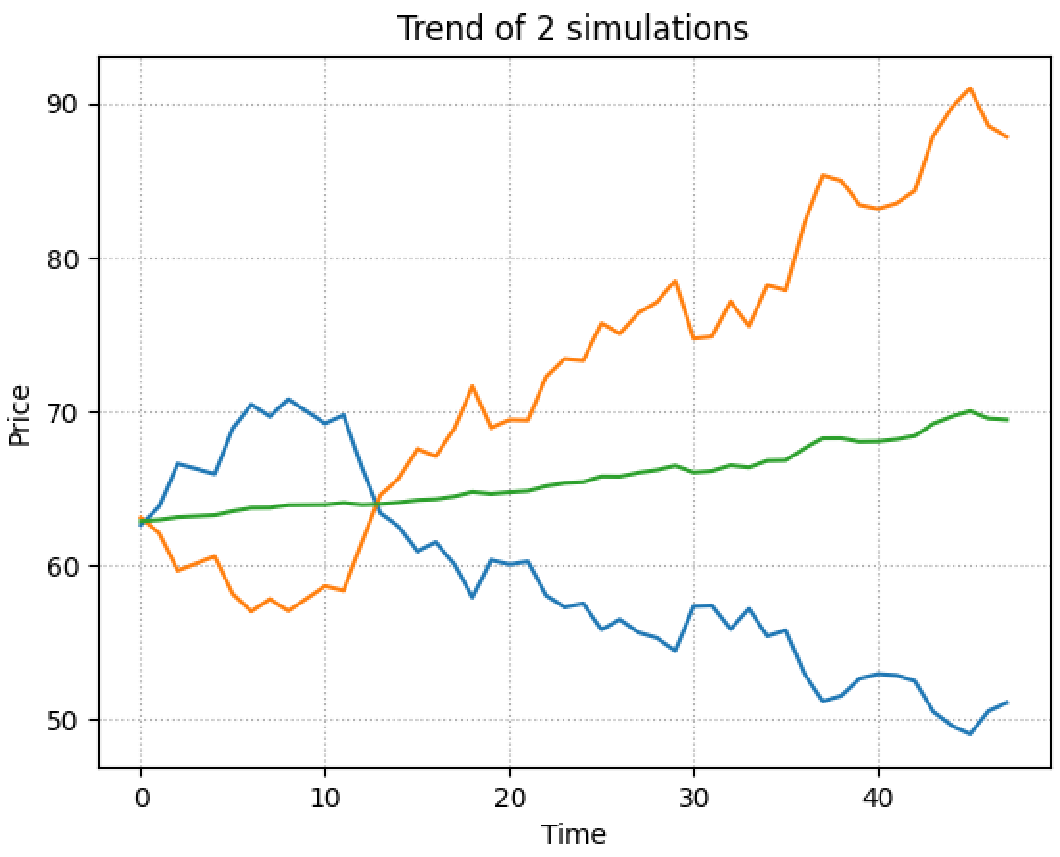


Figure 2.1. Price Simulations with GBM

The image above may seem confusing, but it needs to be analysed in its entirety. As can be seen, we find the trend of 2 simulations for a total of 48 time units. Doing a simple calculation, considering that one hour of time corresponds to 12 units of 5 minutes each, we observe that exactly 4 hours are plotted. Later we will discuss why we therefore have 48 prices for each trading day. It is also interesting to note that the fluctuations are quite different from each other, and for our purposes we will take the average of the two trend values at each time instant. In addition, the standard deviation over the whole time frame is equal to 0.584, representing with good accuracy a real scenario excluding exceptional cases. As a further implementation choice, a smaller number of simulations could have been generated, e.g. 50, 20 or 10, so as to be able to use a more fluctuating average value and thus deal with more particular cases in which prices vary greatly in the face of sudden changes, adding a further degree of complexity. However, we have chosen this parameter in order to fall within a more ordinary situation, so that we can concentrate later on the implementations of the environment and the Gym.
From now on, we can proceed with the analysis and definition of the Limit Order Book,

a fundamental tool of intraday markets. We will be able to do this, however, thanks to GBM, which will support us in the simulation of price trends, allowing us to stay close to the real cases.

2.2 Limit Order Book

Let us now focus our attention on a tool that is as powerful as it is effective in regulating intraday markets: the Limit Order Book. In order to do so, we will rely on [18] [23] and [7] and their analysis, looking for the most practically feasible way and which, again, comes closest to reality; all this, of course, not without approximations. Precisely, the Limit Order Book (LOB) is a tool that allows traders to buy and sell in the markets; it consists of a list of buy orders and sell orders for a given asset, ordered by price and execution time of the order itself. These two sub-lists are called ask orders and bid orders. The best buy order forms the best bid and respectively the best sell order forms the best ask. The difference between the best bid and the best ask is the so-called spread. The fundamental role of the LOB is to adjust the incoming orders by matching them with those already in the bid and ask lists, allowing it to update the status of the orders in the market and consequently for traders to work out new strategies for their orders to be placed in the market from time to time. When a new order arrives, it can either be executed immediately against an existing order on the opposite side of the book, or be added to the book if it cannot be executed at the current market price. The LOB is constantly updated as new orders arrive and old orders are executed or cancelled. Let's see a graphic representation.

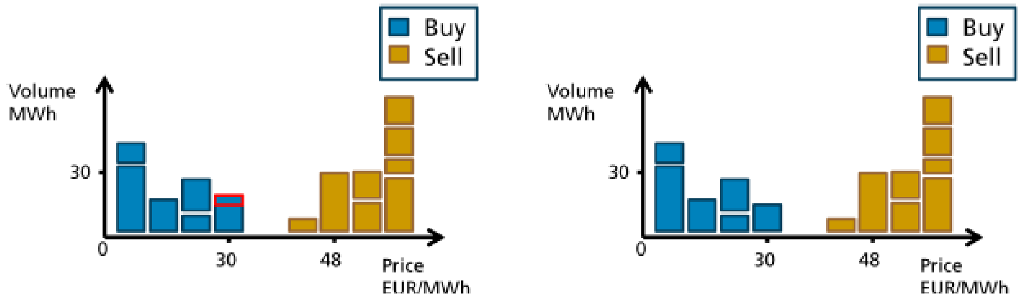


Figure 2.2. Evolution of LOB after a 5MWh market order to sell, that matches the best limit order to buy power at 30 EUR/MWh. This figure is taken from [7]

A further point to consider is the fact that there can be different types of orders:

- limit order: set the price at which to buy or sell a given amount of volume
- iceberg-order: in short it is a large volume order split into smaller orders that take over the orderbook sequentially; allows for limited influence in the market for large volume trades

For simplicity of implementation from here on we will assume that all orders placed on the market in our simulated environment will be 'Limit orders' and that if they do not match any orders in the respective side of the book they will be discarded and not placed in the book in order to have them in the LOB at time $t+1$. This assumption could prove to be very strong and therefore could make the reader think that a very high level of approximation is reached from here on, however it is necessary to consider that in our case the LOB at each time instant t will be generated from the price constructed in the previous section. This means that for each time instant - recall that for each day we only have 48 - we will have a limit order book with bids and asks generated randomly around the average price; in this sense the LOB and its internal orders will follow the price at each 5-minute time instant and our future agent will have to deal with them.
Let us now try to show how we implemented the LOB and above all to clarify what assumptions we undertook to do so. The reconstruction of the LOB in intraday markets is a widely studied topic, precisely because of the need to get as close as possible to real cases. Our requirements start from the fact that we want to keep as close to reality as possible while at the same time trying to maintain a level of simplicity that allows us to focus on the real topic of this thesis: trading. For this reason, the ideas of implementing it using an Ordinary Differential Recurrent Neural Network in [18] and a Deep Convolutional Neural Network in [23], are extremely time consuming in order to advance in our workflow in acceptable times. We therefore decided to rely entirely on the capabilities of the GBM to reconstruct average prices over time and modelled the LOB construction according to the following assumptions. Our LOB will be formed from an average price and volatility. Both of these variables were taken from the national unit prices of the Italian market from 2019 to 2021. With these we can derive the standard deviation by following the formula:

$$\text{Standard Deviation} = \text{Volatility} \times \text{Price}$$

We then added random noise at a rate of 10%. From here it is necessary to construct the various LOB levels for the bids and asks with randomly generated orders. Before proceeding it is necessary to add that for the sake of simplicity we assume that the bid and ask orders are respectively divided into 5 levels progressively ordered according to the GBM generated price; ea variable will also be added to each order to be understood as the specific volume of that order in MWh. First, 100 orders are randomly generated with an average equal to the average price considered and with a standard deviation equal to that calculated in the previous step. For bids, only those below the average are considered, vice versa for asks. But that is not all. In fact, in order to constitute a more realistic LOB of 5 levels, 5 bids and 5 asks closest to the average price are chosen with greater probability, respectively. Finally, a multiple value of 0.25 will be added to each order, in addition to the price just generated, to represent the available volume. Again, realistically this value is free and therefore adds a further degree of complexity, but for the reasons explained earlier we have decided to fall back on this case in order to also simplify the analysis later on. Below, the reader will find a representation of our LOB at a given instant of time t in order to get an idea of how it is structured.

From the previous example, let us first observe the distinction between bids and asks. It is possible to note that the levels are sorted according to the price and that each of

Table 2.1. Example of Limit Order Book at time t generated with mid-price

	Price €	Volume MWh
Bid 0	48.93	3.0
Bid 1	54.9	1.5
Bid 2	56.71	2.25
Bid 3	59.98	2.0
Bid 4	60.69	2.75
Ask 0	63.34	2.5
Ask 1	67.63	1.0
Ask 2	67.64	1.5
Ask 3	83.76	1.25
Ask 4	92.76	1.25

them is associated with the volume, as a multiple of 0.25. Specifically, the best bid is at a price of €60.69 while the best ask is at a price of €63.64. Hence it follows that the spread is worth €2.65 with a total volume of 19 MWh. We can start thinking that our future agent will be able to consult this specific LOB and try to post, for example, a sales order of 1 MWh at €60. This order can be matched with Bid 4 but not with Bid 3, allowing him to sell the desired quantity. If, on the other hand, it tries to sell 5 MWh at the same price, it will not be able to meet a complete match. In any case, this case study will be discussed later.
Now that we have clarified what a LOB is and how it works, we can explain in more detail its use within our environment. The fundamental concept, as also shown in [6] is to enable the agent to observe the LOB's essential information from the environment while reducing complexity. The assumptions we made in order to achieve this were: first, energy trade is done in the order of milliseconds and for this reason it is continuous, we however decided to trade by discretizing in time units of five minutes. Secondly, we decided to restrict the agent's trading time to the last 4 hours to the last 30 minutes before delivery: this is because we noticed, as specified in [6] that most trades are executed in this time interval. Thus it should now be clear to the reader why for each trade day there will be exactly 4 (hours) $\times$ 12 (units of 5 minutes in an hour) different trade offers. An example of this behaviour can be seen in the figure 2.3. We have therefore completed our discussion of the Limit Order Book. We will now proceed to the analysis of a further fundamental tool for obtaining electricity price forecasts: Extreme Gradient Boosting.

2.3 XGBoost model

Along our workflow, there arises the need to provide the future agent with forecasts of subsequent electricity prices on the basis of which he will have to decide what action to take. For this purpose, an efficient system widely used in machine learning was chosen: Extreme Gradient Boosting [3] (XGBoost). XGBoost is a very powerful tool that implements tree boosting, which is a technique of supervised learning. It combines a set

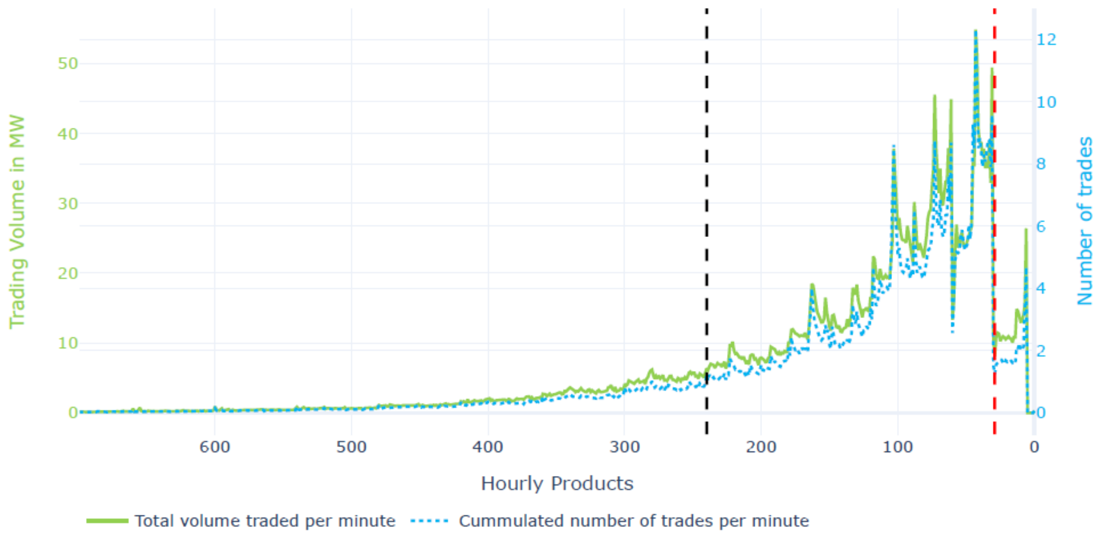


Figure 2.3. Average number of trades in volume in the intraday electricity market in Germany in 2018. . This figure is taken from [6]

of weak learners, the decision trees, into one strong learner by iteratively adding new trees that attempt to correct the errors of the previous ones. XGBoost itself introduced several improvements to previously used techniques, allowing for improved performance and speed of tree boosting. An additional feature is that it can be used in many domains, allowing it to achieve excellent results in all its applications.
Among other advantages, it has a high number of hyper-parameters that can be tuned to optimise the model's performance and prevent overfitting, it allows the easy handling of missing values and outliers, and it also allows classification, regression and clustering tasks to be performed. On the other hand, it is also necessary to remember its disadvantages: it is difficult to interpret as the number of trees increases, it is very sensitive to the choice of hyperparameters requiring a time-consuming trial and error approach, and it is prone to noise as the learning rate increases and the regularization term decreases. However, it was chosen for its positive aspects.
As explained in [15] the optimization of XGBoost over other algorithms lies in three key concepts.

1. Parallelization: the tree construction process is carried out through parallelized implementation, possible thanks to nested cycles; the external one which enumerates the leaf nodes of a tree, and the internal one which calculates the features; loops are swapped in a way that allows improvement from the runtime point of view by globally identifying all instances and then sorting with parallel threads

2. Pruning: the max_depth parameter is used for the stopping criterion, and pruning of the trees backwards begins

3. Hardware optimization: available hardware resources are efficiently utilized by the

algorithms; this is all due to the allocation of internal cache buffers that allow gradient results to be saved.

The objective function consists of a loss function that measures the difference between the predicted values and the true values, and a regularization term that controls the complexity of the model. The general form is as follows:

$$\mathcal{L}(t) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

where y_i is the true value, $\hat{y}_i^{(t-1)} + f_t(x_i)$ is the predicted value calculated as the sum of the previous prediction $\hat{y}_i^{(t-1)}$ and the output of the current tree $f_t(x_i)$ for observation i; $\Omega(f_t)$ represents the regularization term that controls the complexity of the current tree f_t penalizing complex trees to prevent overfitting.
The regularization term can be expressed in the form:

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda\|w\|^2$$

where γT represents the complexity of the tree structure, having T as the number of leaves in tree f_k, and γ controls the importance of this term: a larger value of γ will result in simpler trees with fewer leaves; $\frac{1}{2}\lambda\|w\|^2$ represents the L2 regularization commonly used in leaf scores, w is the vector of leaf scores, and λ controls the importance of this term: a larger value of it will result in smaller leaf scores and a simpler model.
Now that we have defined the loss function, let us go into the details of the implementation and see what results have been achieved on electricity price forecasts.
First of all, it is necessary to specify that we have used the implementation of [2], this will give us advantages in terms of simplicity and speed. In order to be able to do this, we will have to construct a framework suitable for our purpose. Conceptually, this uses tabular data, which we will soon list, to perform the actual forecasts. The final model will be trained from the average prices and volatilities for the entire year of 2019, which will be the historical data from which we will derive further data for XGB training.
Let us now show how we constructed the historical data, assuming that we tried to keep as close to reality as possible. For each daily average price in 2019, a simulation of price fluctuations was initially generated via the GBM, and for each of these, transactions occurred in the LOB were simulated; finally, all of this data was entered in tabular form to form the previously mentioned framework. Specifically, for each price given by the GBM per five-minute time unit, 1 to 3 transactions were randomly generated following a normal distribution with mean equal to the price considered and standard deviation equal to the deviation of all prices generated by the GBM. Thus, to this *Transaction Price* an *Execution Time*, a *Transaction ID*, the *Daily Mean* and further temporal data were then added.
In addition, since each transaction corresponds to an update of the LOB, additional features describing its state, including the five bid and ask levels, the spread, the total volume and the volume-weighted average price, were added. An example of an observation of historical data can be seen in Table 2.2.

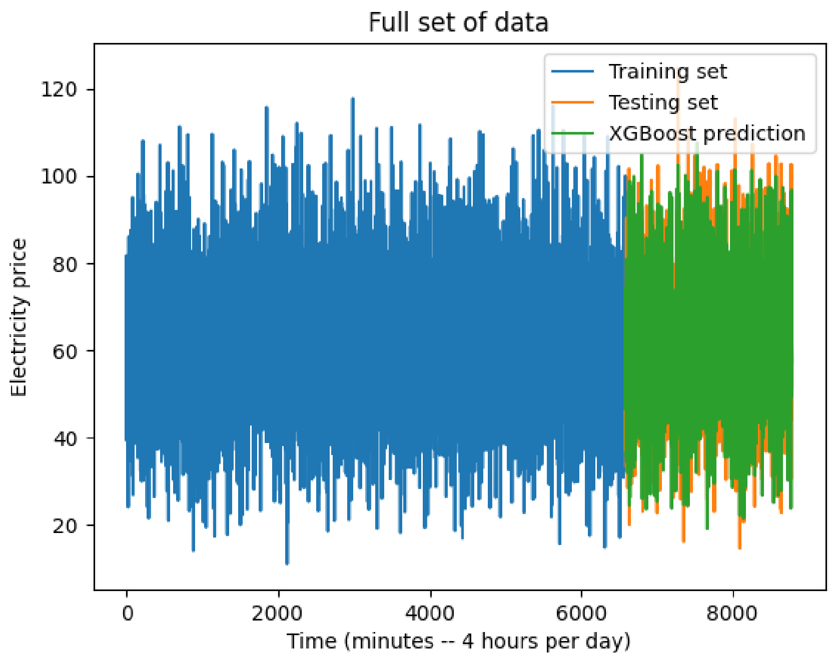
Feature	**Value**
Transaction ID	1
Transaction Price	63,64
Execution Time	2019-07-02 10:00:00
Daily Mean	64,59
Previous Daily Mean	63,36
Hour	10
Weekday	2
Week of the Year	27
Month	7
Quarter	3
Ask Level 1 ... Ask Level 5	68.24 . . . 75.7
Bid Level 1 ... Bid Level 5	63.64 . . . 49.8
Spread	4.6
Available Trading Volume	17.5
Volume Weighted Price	64.39

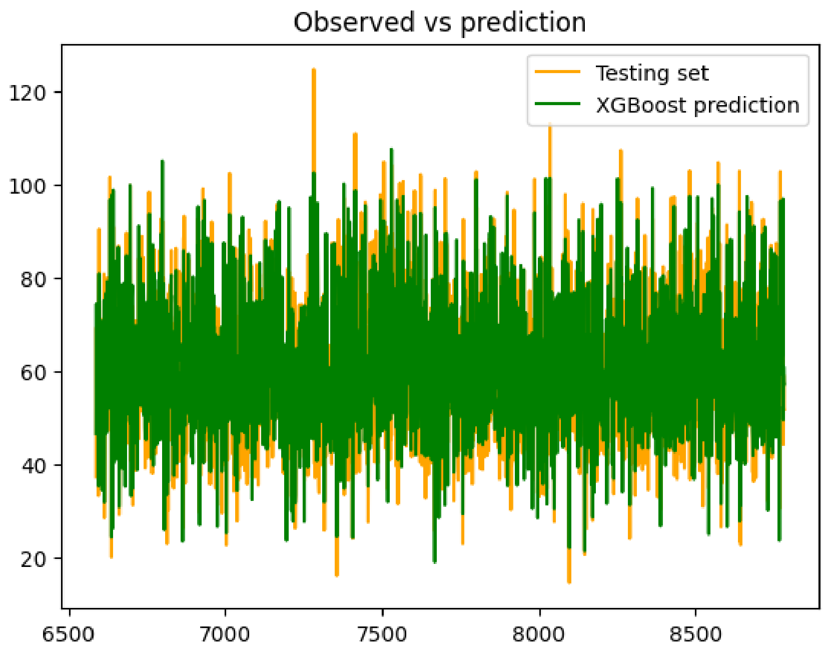
Table 2.2. Example of a row of the Historical Data Framework for XGB

For our purposes we have used the entire framework with the exception of the columns *Transaction ID*, *Execution Time* and *Volume Weighted Price* so as to obtain a forecast on the selected target, i.e. the transaction price -highlighted in green in the table above-. From here we performed the division into train set and test set with a test/original data ratio of 0.25, meaning that the test data will be 25% of the original data and the training data will be 75% of the original data. Now using the parameters in table Table 2.3 we obtained a model with the following metrics:

Mean Absolute Error (MAE): 5.849
Root mean square error (RMSE): 7.121
Weighted Absolute Percentage Error (WAPE): 0.096

To clarify the performance of the model, we can observe the graphs 2.3.
As can be seen, our XGB seems to perform very well, and indeed it does, however we would like to point out that this high level of performance is induced by the fact that the input data on which the training is carried out is not 'real' but 'artificial', constructed to reflect reality as closely as possible. This, however, is enough to make future prices more predictable as they are detached from the dynamics given by real contexts and allows these performances to be observed. In a real context, certain contextual facts, e.g. the outbreak of a war, would alter price fluctuations beyond recognition, drastically reducing performance.
Now, thanks to this discussion and analysis on XGB, we will be able to provide, at each time step, a forecast on the next electricity price, which will help the future agent to analyse the environment and understand what action is best suited for its goal. Let us therefore imagine that given a context at time t, the XGB will provide a prediction for time $t + 1$ with a certain percentage of accuracy.
In the next section we will explore the Gym OpenAI environment which will allow us to define the set of rules and methods that determine what the agent can and cannot do, what it can observe and how it must behave.





2.4 OpenAI Gym environments

In this section we will show what tool we will use to define everything we have talked about so far. In particular, we will define how it was necessary to describe the agent and the environment, what is the space of the actions granted to the agent and what is the space of the observations of which the environment is made up, what will be the

Parameter	Value
Col-sample by tree	0.82
Gamma	0.01
Learning rate	0.11
Max depth	6
N-estimators	187

Table 2.3. Hyperparameters of the XGBoost for the price forecast

methods and functions needed to make all the mechanisms, which we now know very well, of reinforcement learning applied to trading in the intraday electricity markets work.
The tool under discussion is the Gym environment from OpenAI, all its documentation can be found here [11]. Basically, the Gym environment allows the agent to interact with the environment, defined in advance, in a controlled manner, so that all actions, states, etc. can be tracked over time. It is necessary from the outset to lay down some definitions to clarify the next steps.
First, we speak of a *timestep* as a discrete unit of time in which the agent can observe the state of the environment, choose an action and receive a reward. The episode, on the other hand, corresponds to a sequence of consecutive timesteps that ends when the agent reaches a final state or a predefined time limit. This helps us to understand the dynamics we are about to describe. As specified above, in our use case, a timestep corresponds to a time unit of 5 minutes in which the agent, given the current LOB, will generate a transaction that may or may not be matched with the offers on it and consequently generate profit. A set of 48 timesteps will constitute an episode.
An *episode* can be defined as a sequence of states, actions and rewards that end when a terminal state is reached or a maximum number of steps is exceeded. We can see an episode as an attempt at learning on the part of the agent, remembering that its goal is to maximise the total reward it receives during the episode. In our case, one episode will correspond to a fixed sequence of 48 timesteps and thus to one day's trade.
Thus, as depicted in the original paper, the Gym environment from OpenAI is a toolkit for developing and comparing reinforcement learning algorithms. This provides us with a common interface for interaction with our customised environment, as well as tools for visualisation and performance analysis. Let us now go into detail on the methods required for implementation.
First, we need to define the action space and the observation space. These will actually constitute the way in which the agent can interact with the environment and the set of information that constitutes the environment that the agent can access to take its actions, respectively.
Ideally, we would like our agent to be able to post buy and sell orders at a certain price. He will therefore be able to perform three different types of actions: buy, sell or do nothing, and consequently set a price in a given range. In addition, a further assumption made during the unfolding is that the buy-sell will be set at a volume of 1MWh, this is to simplify the way the LOB is updated and to narrow the scope of the agent's actions. So ultimately the agent will be able to set a price and decide to post a buy, sell or do nothing

order. With regard to the observation space, i.e. the set of values accessible to the agent on the state of the environment, a set of eleven consultable variables was defined. The variables are specifically divided into three groups: price-related variables, volume-related variables and an episode-related variable. A list with a detailed explanation can be found in the table Table 2.4.

Table 2.4. Observation Space variables

Variable group	Variable Name	Description
Price variables	Current VWAP	Volume-weighted average price on the current orderbook
	Previous VWAP	Volume-weighted average price on the orderbook of the previous timestep
	Day-ahead Price	Average price for GBM on the current day
	Current Forecast	Next average price forecast
	Previous Forecast	Forecast of the next average price of the previous timestep
	Current Difference	Difference between the current volume-weighted average price and the current forecast
	Previous Difference	Difference between the volume-weighted average price and the forecast of the previous step
	Price Maker	Describes the price trend in terms of fall or rise
Volume variables	Absolute Current Volume	Total value of volume traded in absolute value
	Current Volume	Current volume availability
Episode Variable	Time to End	Number of steps at the end of the episode

Once actions and observations have been established, it will be possible to describe the three fundamental methods of the Gym environment: the step() method, the reset() method and the optional render() method. We will gradually proceed into the specific implementation of these methods, beginning by discussing their function and how agent-environment interaction takes place.
As can be learned in the official documentation, the *reset()* method has the task of resetting each observation to an initial state. This will be called specifically at two particular times: at the start of the trade session, when the generic environment condition must be shown to the agent, and later at the conclusion of each episode. In our specific case, the method will initially need to initialise the states of the environment from day one with all values observable at that time. Those not available, such as the *Price Maker* in Table 2.4, will for simplicity's sake be set to 0. Subsequently, as the steps and thus the episodes themselves progress, the values will be updated correctly and the agent will be able to use each individual variable to determine what action to take. We also add that every thirty episodes, and therefore every thirty days, the model that provides the forecast useful for some price variables will be re-fitted, making it possible to capture new

variations and new price ranges on the market. As a further clarification it is necessary to add that most of the values have been normalized between 0 and 1 for implementation issues, and in this regard a scaler has been used, also this fitted every thirty days for the same purposes.
Now that we have understood the dynamics governing the use of *reset()* and the reasons why it is necessary to use it at certain times, let's proceed with the explanation of the *step()* method, which conceptually carries out the bulk of the work.
The *step()* method in OpenAI Gym is a function that takes an action as an input and returns four values: observation, reward, done, and info.
Ideally, therefore, it will have to take the action chosen by the agent and modify the environment on the basis of this, calculate the reward, increase the timestep, check the termination status of the episode and update the new observation space. From here you can understand why it represents the core of the whole structure. In our case, the function will receive the action of the agent and start a series of check chains. First it will process the order placed by the agent, remembering that the agent can buy, sell or do nothing. First, the existence of a possible match on the bid/ask levels of the current LOB will be verified: in the event that it is not verified - or that the action chosen by the agent is to do nothing - and therefore no transaction effective recorded, we will proceed to update a special register, created alongside, which keeps track of each single event, and move on to the next timestep. In the event that a match is actually detected (later we will see how) all the event tracking registers will be updated and the actual gains or losses on the balance sheet will be calculated. It is now necessary to clarify how profit and loss are actually intended to be recorded. We have to imagine that in a real environment, a trader can buy and sell at any time. At the close of the market for a certain hourly product, i.e. at the delivery of a certain volume in MWh of energy, the trader must try not to unbalance the position: he will have to try to have a volume of MWh sold equal to the MWh bought.
If so, then at the end of the episode, the sum of the respective prices of volumes sold and volumes bought is actually calculated, determining the final balance. More interesting is the case where an imbalance occurs at the end of the episode. In a real-life context, a trader with an unbalanced position will have to compensate for his negative or positive volumes by means of requests to other sellers or buyers at prices outside the standard range, thus resulting in major disadvantages in terms of profit. This case study is interesting because we would like our agent to learn in the long run not to have unbalanced positions.
This is why the actual computation is only performed on unbalanced positions. Let us now give an explanation of how matches are actually detected on transactions generated by the agent. Let us pretend that we are at a certain time t and that the current LOB is Table 2.1. If the generated action is to buy at a price of 62€there will be no match, because the lowest available ask is 63.34€; if instead a price of 68€was generated then there would be an actual match on the first ask. Remembering also that for simplicity's sake all shares are to buy/sell at 1MWh, we could therefore buy 1MWh of the 2.5 available on the first ask level. In case there is not enough volume available, we would buy the remaining required amount on the next levels, provided the respective prices are lower

than our 68€. The same applies speculatively on the bids in case the chosen action is to sell at a certain price.
However, our implementation of *step()* proceeds to increment the timestep. In the event that this is the 48th timestep then we would update the observations in Table 2.4 and proceed to the eventual calculation of the balance; otherwise we proceed to the obtaining of the reward. As mentioned above, the reward is in fact the reward given to the agent to direct him towards his goal. The way in which it is defined may cause behaviour to change completely. A more detailed explanation of our implementation will be provided in section 3.2; for now, it is necessary to know that this is calculated at this point in the *step()* method and that it will be returned as the method's return value. Finally, in our implementation, an update of all track registers will follow, and finally the method can be considered complete.
Let us now give a brief description of what the render() method proposes to do. The need that arises when implementing this method is to visually and graphically monitor the agent within the environment. For logical reasons, our implementation will display four graphs at the end of the entire training period showing the balance trend, the cumulative balance, the average reward over time and finally the imbalances over time. Figure 2.4 is an example of render() over ninety days. At this point, the reader should

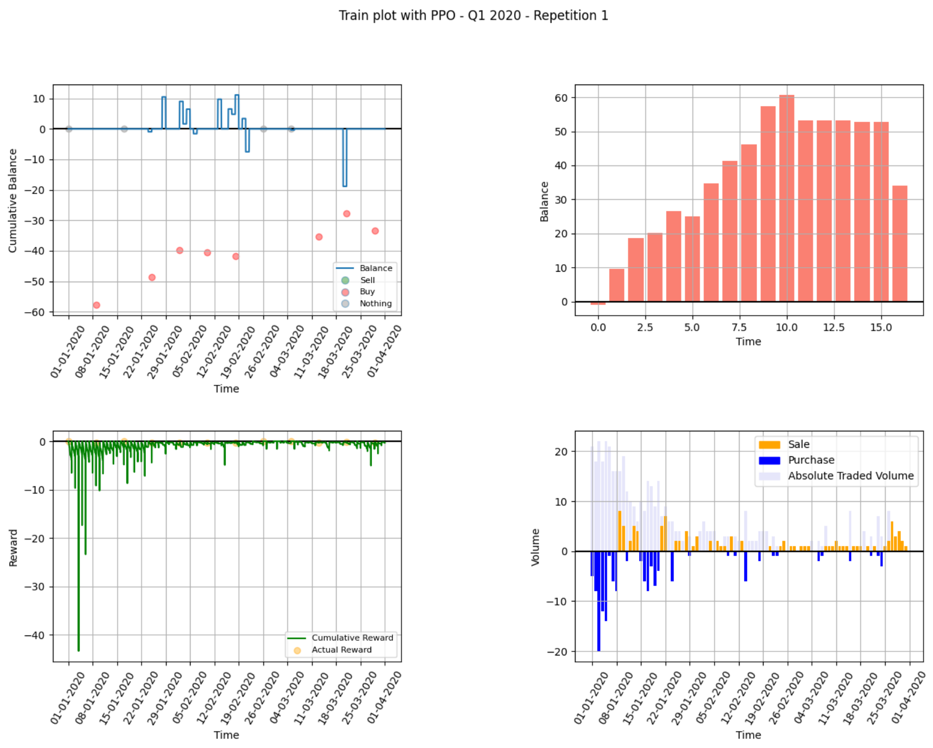


Figure 2.4. Example of plots on 90 days period

be fully aware of all the tools used for this specific task.
We have therefore concluded the discussion on the Gym environment from OpenAI and

it only remains for us to go deeper and discuss the training process and the tuning of the hyperparameters.

Chapter 3

Training Process and Fine Tuning

In this chapter, we will go on to define in more detail and with greater attention the mechanisms that govern the learning of our agent and determine its behaviour within the environment. However, at this point in the discussion, the reader should be more familiar with the dynamics described so far, and although we go into the mathematical details of the matter, he or she should not find it too difficult to understand them.
Our intention is to describe in the following sections the difference between on-policy and off-policy algorithms; to deal adequately with the issue of rewarding as a conditioning mechanism for learning; to mention Proximal Policy Optimisation and give a detailed description of it; and finally to deal with the fine-tuning of hyperparameters. All this, of course, while keeping a constant eye on our specific task: electricity trading.

3.1 Off-Policy or On-Policy?

One of the key problems concerning value functions is that of generalization, in fact, assuming we have a table with one entry for each state, not only the appearance of memory when it comes to large tables, but also the time to fill them and the data with which to do so. In this sense, we wonder how experience can be generalised in a useful way to get good approximations. When we tackle problems with the reinforcement learning paradigm, cases may occur, in terms of descriptions of the environment, that the agent has never seen before, and so the only way to learn something from these is to have a generalisation from the cases seen before. The type of generalisation we refer to is described by the "value function" which attempts to generalise to construct an approximation of the entire function. One class of methods for approximating functions is that of gradient descent methods, which are among the most widely used and widely deployed. In the gradient-descent methods, we have a vector with a fixed number of real components, $w = (w_1, w_2, w_n)^T$, which will be updated as a series of time steps. In addition we have the approximation function $\hat{v}(s, w)$ which is a smooth differentiable function. We also have the states we observe S_t. Assuming that the states appear with the same distribution, d, on which we try to minimise the Root Mean Squared Error

(RMSE)

$$RMSE(w) = \sqrt{\sum_s d(s)[v_\pi(s) - \hat{v}(s,w)]^2},$$

what you want to do is minimize the error on the observed examples, so with gradient-descent you try to do this by adjusting the parameter vector after each example by a small amount in the direction that most reduces this error.

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t,w_t)]^2 = w_t + \alpha[v_\pi(S_t) - \hat{v}(S_t,w_t)]\nabla\hat{v}(S_t,w_t)$$

From the previous formula the term α is a positive step-size parameter, and $\nabla f(W_t)$ denotes the vector of partial derivatives with respect to the components of the weight vector. This vector denotes the gradient with respect to w_t. Hence the method is named gradient descent because the step of overall in w_t is proportional to the negative gradient fell the quadratic error of the example. In this way, this method ensures that we converge to a local optimum.
Very simply we can have both linear gradient-descent methods that work very well in practice and depend on the choice of features, and backpropagation methods for multi-layer neural networks that are nonlinear instead. On-policy algorithms converge through gradient-descent methods to a solution with the mean square error related to the minimum possible error.
On the other hand, we have so-called off-policy algorithms, i.e., those that are not related to the use of value functions and do not directly use stock values to select future actions: in this regard we briefly discuss Actor-Critic methods. These methods are so called because they have a clear separation between policy and value function in terms of memory. Conceptually, the policy is the actor choosing which action to take, while the estimated value is the critic in that it makes a judgment about the chosen action. The actual critic has the form of a temporal difference error: this signal is able to drive learning in both the actor and the critic. In order to understand it better, we can refer to the figure Figure 3.1. Basically once an action is taken, the critic checks whether the new state is better or worse than the previous one. That is called the TD error and follows the formula:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V(S_t),$$

where V_t is the value function implemented by the critic at time t. This error can be used to evaluate the action selected A_t taken in state S_t and after that we can have two different cases: if the TD error is positive it highlights that selecting A_t should be favored in the future, while if it is negative it suggests the tendency to consider less that action in future.
Generally, Actor-Critic methods have two major advantages:

- requiring little computation for action selection
- explicitly learning optimal probabilities to select various actions

To recapitulate, on-policy and off-policy algorithms are two types of reinforcement learning methods that differ mainly in the use of data collected from the environment. On-policy algorithms use a single policy π and work with the data generated by it (actions,

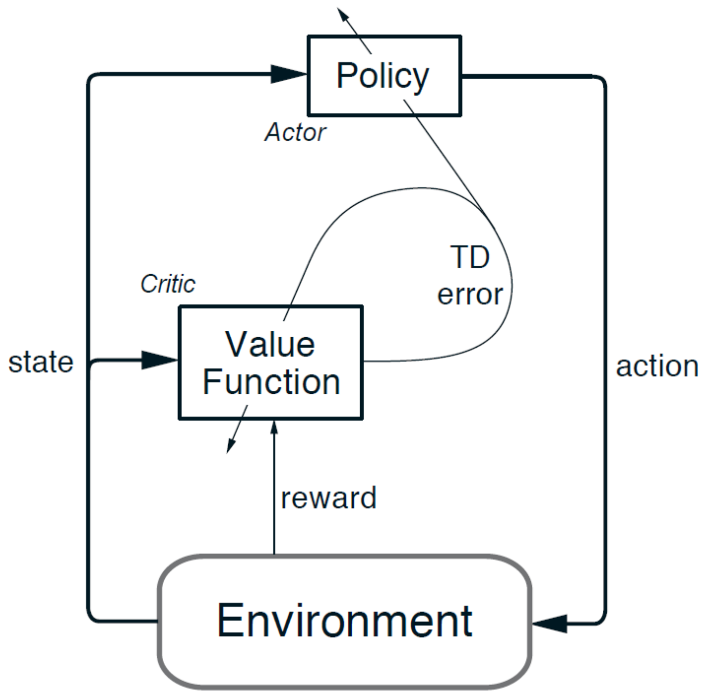


Figure 3.1. The actor-critic model. This image is taken from [20]

states, etc.). Off-policy algorithms on the other hand work with two policies: a policy that the agent tries to learn and a behavioural policy that the agent uses to interact with the environment. Some examples of on-policy algorithms are TRPO (Trust Region Policy Optimization), PPO (Proximal Policy Optimization), A2C (Advantage Actor-Critic) and SARSA (State-Action-Reward-State-Action). These algorithms directly maximize the performance objective or an approximation of it. Some examples of off-policy algorithms are DDPG (Deep Deterministic Policy Gradient), DQN (Deep Q-Network), SAC (Soft Actor-Critic) and TD3 (Twin Delayed Deep Deterministic Policy Gradient). These algorithms use importance sampling to estimate the value function of the target policy from the data generated by the behavior policy.
Well, now that we have this distinction between algorithms clearly in mind, let us move on to an equally important argument: rewarding. Next, in section 3.3, we will give a more detailed explanation of the algorithm chosen for our task: Proximal Policy Optimisation (PPO).

3.2 Choose the Reward Properly

The choice of reward regarding the agent's actions is one of the trickiest parts of the entire process. In fact, depending on the desired behaviour, the reward may totally affect

the behaviour each agent assumes within the environment.
Trivially, one may start from the consideration that an action that leads to a better state than the previous one, in terms of single steps or even episodes, should be evaluated positively, e.g. with a numerical value +1. On the other hand, an action which would lead the agent into a worse state, again in terms of steps or episodes, may be evaluated negatively with a negative reward of -1. But this can be misleading in some respects and in order to explain this concept which is as simple as it is in reality complex, we will refer to an example.
Let us imagine that we want to instruct an agent to play chess and that we obviously want to do this by reinforcement learning. Now, in order to give our agent the right direction, we might think that whenever he succeeds in eating an opponent's piece we would bring him to a better state and thus want to reward him with a positive reward; on the contrary, if he reverts to a state in which he gets a piece eaten we would punish him with a -1. By this reasoning we would be making a major mistake: it is not necessarily the case that eating an opponent's piece is a correct action, since it could expose the queen to being taken on the next step or, more dramatically, it could expose the agent to a checkmate; on the contrary, it is not necessarily the case that having the opponent eat a piece leads to a worse state at the next time step, as it could lead the agent to a checkmate or to the capture of some important piece of the opponent. This reasoning should make the reader understand not only the importance of choosing the rewarding function, but also secondary aspects such as the "granularity" of the reward, understood as rewarding at each time step or only at the end of the episode.
During our discussion we will see more forms of rewarding in detail and in relation to these we will analyze the behavior of the agent combined with the choice of model hyperparameters. The results are discussed in section 4.2. The starting point of our reward directly follows the formula from [6]:

$$r_t = {r_t}^{vol} + {r_t}^{trade}.$$

From the agent's point of view, we must reflect the fact that selling more capacity on the market corresponds to a decrease in volume availability and conversely, buying more capacity would correspond to an increase in available volume. Furthermore, the choice was made to give a reward for each time step. Specifically, the r^{trade} follows this formulation:

$$r_t^{trade} = p_t \Delta v_t - c_{fee}|\Delta v_t|$$

$$with\ \Delta v_t = a_t - a_{t-1}$$

In the experimental phase, it was sometimes chosen to insert a fee corresponding to the opening of a position of the agent in the market. From this it follows that a growth in volume corresponds to a negative reward, whereas a decrease corresponds to a positive reward. As far as the second factor ${r_t}^{vol}$ is concerned, it was decided to calculate it only at the end of the episode, at the last step, and is defined as follows:

$$r_t^{vol} = \begin{cases} 0 & \text{for } t < \mathrm{T} \\ -0.1 \Delta {v_{penalty}}^2 & \text{for } t = \mathrm{T} \end{cases}$$

where $v_{penalty}$ is a penalty reward measuring the imbalance in terms of volumes bought and sold. What we are trying to achieve with the latter is to prevent the agent from ending the episode with an imbalance and thus closing a position that is not at break-even: this would entail the need to buy or sell MW of energy from third-party suppliers at prices significantly higher than the standard market prices. Through the quadratic term, therefore, large deviations are penalised.

3.3 Proximal Policy Optimization

The algorithm chosen for training our model is Proximal Policy Optimisation (PPO) [17], which combines the ideas of Asynchronous Advantage Actor Critic (A2C) [8] and Trust Region Policy Optimization (TRPO) [16]. Conceptually, the key idea is to assume that the new policy is not too far from the old policy, after each update. For this reason, the PPO uses clipping to avoid overly large updates.
Let us now go into detail and see what is the theory behind PPO and what are its implications for our model.
Proximal Policy Optimization is built in OpenAI and has proven successful in a wide variety of tasks: from automation to Atari games and so on. One of the first thoughts to make when studying reinforcement learning is the fact that, unlike other paradigms, the data with which the model will be trained are self-generated, or rather are generated through the steps and thus the actions that the agent takes within the environment, so it is first necessary to remember that the training data are constantly changing. Later we also observe that this type of learning is largely susceptible to hyperparameters: but we will discuss this in section 3.4.
To address many of these pesky problems in reinforcement learning, the first requirement leading to the emergence of the PPO is to find the exact trade-off between simplicity of implementation, efficiency, and furthermore, simplicity of tuning the hyperparameters.
Since PPO is a policy gradient method, which means that it does not use a buffer to store past experience, we can see that its learning method goes directly from the agent's encounters with the environment, and so when the last batch of experience is used for the policy update this is discarded and no longer used. So to define this method of policy optimization, it is necessary to define the formula that explains how the policy updates:

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t]$$

The first term π_θ is our policy: it corresponds to a neural network that takes the observed states from the environment as an input and suggests actions to take as an output; the second term $\hat{A}_t$ is the advantage function which in practice tries to calculate an estimate of the value of the action taken in the current state. In order to compute the advantage we need two things: the discounted sum of rewards and a baseline estimate. The first part

$$\sum_{k=0}^{\infty} \gamma^{\kappa} r_t + \kappa,$$

is a weighted sum of all rewards the agent got during each time step in the current episode; the discount factor γ which is usually somewhere between 0.9 and 0.999 accounts for the

fact that the agent cares more about reward that is going to get very quickly versus the same reward it would get a hundred time steps from now. In practice, the advantage is calculated at the end of the episode when all the experience is collected from each step so in fact there is no prediction work in the calculation of the discount because we basically know exactly what happened, step by step. The second part of the advantage function is the baseline or the value function, what it is trying to do is provide an estimate of the sum of the rewards from this point on: it's basically trying to guess what the final return will be in this episode starting from the current state. As you train this neural network which is the value function, it will iteratively update itself using the experience our agent collects in the environment because this is essentially a supervised learning problem: taking states as input and the neural network trying to predict what the sum of the rewards will be from this state onwards. Since the value estimate corresponds to the output of a neural network, this will be a noisy estimate, so there will be variance and our network will not always be able to predict the exact value for those states. If we then subtract the baseline estimate from the actual return we obtained, we get what we call the advantage estimate which answers the question: how favorable is the action you took based on the expectation of what would ordinarily happen in the state in which it was in? Was the action taken by our agent better or worse than expected? So then by multiplying the log probabilities of the policy actions with this advantage function we get the final optimization objective that is used in policy gradient. In the case where the result is positive then the implication is that the actions taken by the agent in the action path resulted in a significantly greater than average return, so what we want to do is to increase the probability that these will be selected again in the coming time. On the other hand, if the advantage function is negative, we will reduce the likelihood of the selected actions.A consistent problem, however, is that continuing to update the gradient via a single batch of data would lead to updating the parameters in the neural network far outside the range in which these data were collected and for this reason the advantage function that is a noisy estimate of the real advantage will be completely wrong: in this way the state-action transitions calculated so far will be ruined by continuing to use gradient descent on a single piece of experience, and if one want to solve this problem you have to make sure that you don't move too much from the old policy with each update. This idea was introduced extensively in [16] and is in fact the entire foundation upon which PPO is built. In order to allow the updated policy to be not too far from the current one, the TRPO uses a Kullback-Leibler (KL) constraint to the optimization objective and what it does is make sure that the new updated policy doesn't move too far from the old policy so it is necessary to stay within the region where at a high level we can observe that everything works as it should. However, these constraints triggered by KL conditions can lead to unwanted training behavior, so it would be nice if we could somehow include this extra constraint directly in our optimization goal. This is exactly what PPO does.
Let's define a variable

$$r_\theta = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_old}(a_t|s_t)},$$

representing the relationship evidently between the new updated policy and the old one. So for example, if we had a state-action sequence of n-steps, the result of the formula would be greater than 1 if the action taken has a higher likelihood now than before, whereas if it were between 0 and 1 it would be inferred that the previous action had a higher likelihood than before. Now, thanks to this brief explanation we can better highlight the objective function from PPO. Here it is:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t]$$

The function is basically an expected value calculated on the minimum between two variables. The first of these two terms is $R_\theta \times \hat{A}_t$: so this is the standard form of Objective for policy gradients that seek to direct policy towards actions that will lead to a greater positive advantage. The second term is very similar to the first one except that it contains a truncated version of this R_θ ratio by applying a clipping operation between $1-\epsilon$ and $1+\epsilon$, where ϵ is usually like 0.2 and then lastly the min operator is applied to the two terms to get the final result.
From Figure 3.2 we can observe that the clip function can have both negative and positive values. On the left half of the figure, the advantage function is positive, indicating that

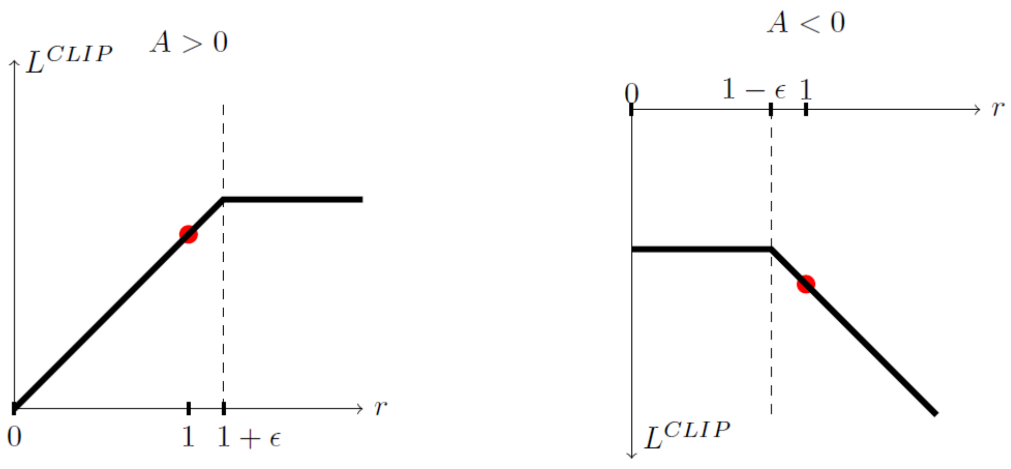


Figure 3.2. Plots showing a single time step of the function L^{CLIP}, for positive advantages (left) and negative advantages (right). The figure is taken from [17]

the selected action had a better than expected effect on the outcome; on the right side of the diagram, the action had an estimated negative effect on the outcome. We can see how the loss function flattens out on the left side the value on the y-axis becomes too large, this behavior is precisely because the chosen action is closer to the current policy than the previous one, and thanks to the clip parameter ϵ we can limit the gradient update by intervening directly on the function. The figure on the right, on the other hand, shows the same behavior in reverse, namely that when the value on the y-axis drops toward zero, we want to prevent the policy update from destroying the results obtained so far, and to do so, again, we use clipping thanks to the ϵ parameter. The function falls into this space only when the last gradient update makes the value of the chosen action higher. Here we would like to go back one step from the last update step and thanks to

the mechanism seen we can do so. Starting with the fact that there is a negative value here, the gradient update result indicates not only to move in the opposite direction, but also by how much to do so, in proportion to how negative, in terms of expectation, the previously chosen action was. Only in this space is the value of the clipped function higher than the unclipped one, and this is what the minimization operator returns.
Essentially, the PPO objective does the same thing as a TRPO objective in that it forces policy updates to be conservative if they deviate too far from the current policy: the only difference is that PPO does this with a very simple objective function that does not require tall these additional constraints or KL divergences, and it turns out that the simple PPO objective function often outperforms the more complicated TRPO variant.

3.4 Hyperparameters Tuning

Let us now proceed to describe the fine-tuning methodology used, first by detailing each hyper-parameter and then indicating how it influences the learning process.
Fine-tuning is a special technique to be applied to pre-trained models that consists of slightly adjusting its hyper-parameters in order to make it perform better in its specific field of application. With it, it is possible to drastically improve the performance and capabilities of a model using feedback from the environment and a reward function. We can give many examples: a language model can be fine-tuned to generate better quality text for a chat bot; a vision model can be fine-tuned to improve the behaviour of a robot. A further problem that can be overcome by this process in reinforcement learning is to overcome the problem of scattered rewards that occurs when most of the generated samples and collected experience are predicted as being of low quality by the target function.
Since we have used Proximal Policy Optimisation as an algorithm, and since the latter uses a gradient method, updating the policy by following the direction of the gradient, we now want to highlight each of its hyper-parameters specifically. It is also necessary to specify that the implementation used is that of Stable-Baselines3 [14] which is a library containing a set of implementations of reinforcement learning algorithms in PyTorch, including our PPO. Some hyperparameters have already been explained in detail in the precedent section, but we will repeat them in the following list in order to give a better overview.

- ϵ corresponds to the clipping parameter, it controls the clipping level of the object function; where *clipping* means that if the ratio between the new policy and the old policy is too large or too small then it is replaced with a fixed value. This prevents the policy from changing too much, ensuring stability first. Respectively, a smaller value corresponds to a larger policy update, whereas a larger ϵ corresponds to a smaller policy update;

- γ controls the discount factor for future rewards; in essence, it gives an indication of how much consideration should be given to rewards received in the immediate future or in the future. A higher γ value highlights an agent more focused on future rewards, while a smaller value highlights a tendency to focus on rewards received more in the current time

- **batch size** controls how many samples to use to calculate the gradient of the object function at each step; it basically determines how often the policy is updated in terms of time steps. A smaller batch size corresponds to more frequent updates with a smaller variance, while a higher value means that the number of steps to be collected before updating the policy is larger, consequently the number of updates is smaller and the variance increases

- **learning rate** basically this is the hyperparameter found in many models using Stochastic Gradient Descent (SGD); it determines how much the policy parameters need to be updated after each gradient step; a higher value can lead to convergence very quickly but also instability; a lower value, on the other hand, can lead to more stable but slower learning and sub-optimal solutions

- **entropy coefficient** controls how prone the agent is to explore different action paths; in this sense, the agent might have a policy that would take it to the highest expected reward, however by exploring new paths it might discover with some probability paths that can return higher rewards to it than the current policy; a high value helps to avoid getting stuck in a local optimum, but reduces exploitation; a low value, on the other hand, reduces exploration, not allowing it to find better actions.

- **value function coefficient** used to estimate the expected return value of each state, minimizing the mean squared error between the predictions and the empirical returns; a high value helps speed learning, but reduces policy improvement; a low value can help improve the policy but can also lead to an increase in the variance of the policy gradient.

As it has been possible to observe, fine-tuning can help our model, and therefore our agent, to improve its performance considerably as the parameters change. However, the whole process, as it has been possible to see, is not free from compromises: in fact, where greater rewards are obtained, perhaps longer learning is needed and so on. It is also interesting to note how the agent's behavior observable during training is influenced: in our specific task we can observe the agent who actually does not play during the episodes so as not to unbalance and therefore maximize the reward, he plays once by buying and once by selling or developing more complex strategies. From here selecting the correct parameters was not trivial, but we will discuss this more in section 4.2. A way as trivial as brute force for tuning is the *grid search*, which consists in testing a grid of hyper-parameters and evaluating the performance of each possible combination: the big disadvantage is the total time, as it would be necessary to try for every possible combination: too long. Now, for our purposes we've used an open source hyperparameter tuning framework called Optuna [1]. This offers us several features that make it suitable for our purposes: automated search for optimal parameters, efficient search with pruning techniques to speed up the process, simple parallelization of search using multiple threads

without code modification.
Through this framework, the whole process is divided into three phases:

1. define the objective function
2. create a study object
3. run the optimization process

The objective function is the main part considered by Optuna for the selection of hyperparameters: it actually only seeks to optimize, maximizing or minimizing, the metric we provide. This process is iterated over an arbitrary number of trials and keeping track of the returned value to evaluate which is the best set of hyperparameters, returned by the objective function. Table 3.1 shows the set of values to choose from for each hyperparameter. In our case, for reasons of time and computing power, we used a relatively low number of trails equal to 200, over a trial period of one quarter, using the average reward per episode as a metric and therefore trying to maximize it. Thanks to the best sets found, we then routed the training process.
As previously specified, the results will be presented in the next chapter. We will finally be able to observe the results obtained by our agent and therefore evaluate it, both in terms of profit and in terms of acquired and adopted strategies. We will try to highlight the correspondence between the selected parameters and the model itself, showing how the actions taken change, with the help of graphs and tables. Finally we will make a brief discussion of the in-depth work that could be done in this regard.

Hyperparameter	Set of values
Batch size	48, 240, 480, 960, 1440
Discount factor (γ)	0.9, 0.95, 0.99, 0.995, 0.999
Learning rate	0.0001, 0.0002, 0.0003
Entropy Coefficient	0.00001, 0.0001, 0.001, 0.01, 0.1
Clip range (ϵ)	0.2, 0.3, 0.4
Value Function Coefficient	0, 0.5, 1

Table 3.1. List of selectable values for hyperparameters

Chapter 4

Results and related works

We are now going to represent the results obtained with the tools, techniques and considerations discussed in the previous chapters. It is worth reminding the reader that the primary intent of this thesis is not to create an agent capable of profiting from trading, or at least it is secondarily so; first and foremost, the intent has been to demonstrate how, by means of a new paradigm of computation, it is possible to instruct an agent to devise strategies on a specific type of market. Having made this necessary premise, we proceed in the following sections to show the results obtained and to comment on them. But first let's briefly discuss some related works.

4.1 Related Works

Nowadays there are many papers that have tried to cover what we set out to do within this thesis, in some dealing in more detail with market analysis, in others with Reinforcement Learning approaches and algorithms that can be used, and in still others with other aspects such as price forecasting, Limit Order Book reproduction, and so on. First, there are many papers that examine the influencing factors in intraday markets as in [4] and [5]. For what concerns price forecasting we have countless treatises such as [9] [10] and [21]. As the last topic of discussion we find actual trading and its automation in the market by more or less complex Machine Learning methods as in [22]. Also from the point of view of RL itself there are many works dealing with trading on such a paradigm and the one we are closest to is precisely [6] which we have mentioned several times. Our contribution lies in using GBM as a method of recreating the LOB and then building an environment as simple as it is real in order to use the PPO and train the agent with the rules we have instantiated.

4.2 Test Results

Let us now begin the discussion of the results. We have seen above that the agent's behaviour can be largely influenced by the choice of one rewarding function rather than another, and in this regard we will analyze the results on the basis of a fee of 0.2 as in [6] and a completely zero fee. We expect the fee to influence the agent's learning by causing

him to adopt more conservative strategies since, if at the end of the episode his relative position has not been closed, he will suffer a very strong penalty from the rewarding signal; on the contrary, with a zero fee value, the agent will be freer to buy and sell and thus unbalance at the end of an episode, thus having the possibility to adopt riskier strategies.
Since evaluating the agent in a pure and unconstrained manner might be inefficient from the point of view of understanding the results, we will set as a baseline for a benchmark an agent that adopts a random strategy, i.e. that adopts random buying and selling actions over time, at transaction prices that are also random. This aspect will help us better understand, at a low level, how our trained agent is behaving.
As a final consideration, it is worth mentioning that the test period begins on 1 July 2021 and ends on 31 December 2021, and we list below the electricity price data for the Italian market for that period. As can be seen from the Figure 4.1 the months of July

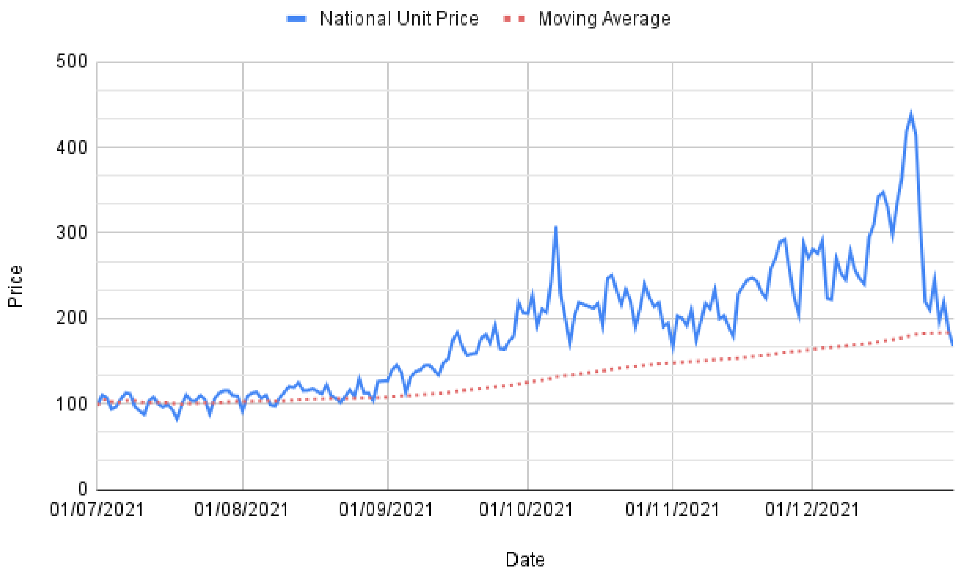


Figure 4.1. Evolution of Italian national electricity unit prices per MWh from 7/1/2021 to 12/31/2021

and August are characterized by having prices with reduced variability compared to the others of the semester. If in fact for the first two the standard deviation is $\simeq 9.88$, for the other three this is $\simeq 60.27$. This figure is very interesting because it already suggests to us a greater trading difficulty in the latter part of the semester induced by a significantly higher price variability.
In 4.2.1 and 4.2.2 we will discuss the results given by adopting a rewarding strategy with a fee cost of 0 and 0.2, highlighting any differences and similarities between a random agent and a trained agent. Finally in 4.3 we will group the results trying to draw conclusions.

4.2.1 Evaluation with 0.2€ fee cost

First, we would like to analyze an agent that pursues random actions that consist of either a buy or sell order at a price between 20€ and 500€: this choice is given by analyzing the Figure 4.1. Below we find the result of the Gym render function. Starting from the reward graph (bottom left) it is possible to see constant negative rewards over time with frequent peaks, this precisely highlights the inability to maximize reward over time, since there is no policy and consequently no updating of the policy. The average value is -9.09, with the lowest negative peak at -42.59 and the highest at -1.29. As a second interesting fact to note we can refer to the plot of traded volumes (bottom right). First, we observe the clear inability to balance positions at each episode: this actually translates into observing that, in both the Q3 and Q4 plots, the sell (in yellow) and buy (in blue) imbalance bars are consistently far from 0, with absolute peaks of imbalanced volumes of 18. The unbalanced percentage is 97.28% with only 5 balanced episodes, 4 of them in Q4. This figure is also visible from the balance-per-episode graph (top right) which shows only 5 bars where in fact the balance is changed, registering a profit or loss. Now let us analyze just that. In fact, as can be seen, in the third quarter only one episode is balanced, and when this happens, because the basis of the stock selection process is solely randomness, the agent registers an economic loss of 123.81€: not good; in the last quarter this situation gets even worse despite the fact that the number of balanced episodes is 4 times higher. The agent continues to record losses on the negative balance by reaching a minimum value of -526.91€ at the 133rd episode, after which he manages to make a slight gain of 17.88€ and close the entire trading period at a negative value of 509.03€. For those who were wondering about the meaning of the top left graph, we can say that it served a useful debugging function throughout the development process, thanks first of all to the red, green and black dots that, randomly sampled once in a while, showed the type of actions taken by the agent, and, corresponding to these, showed the balance trend over time, allowing to observe through a comparison with the plot on the right, during which days positive or negative balance updates occurred, helping to better analyze and observe the agent's strategies. It is trivial to say that as far as this random testing is concerned, we did not need to interpolate this graph almost entirely. As a final observation, we can state that we expect to see, after the agent training, a lowering of the bars in light blue in the background in the traded volume graph, emphasizing the tendency to unbalance little, and a tendency to see the balance lower and then, eventually, rise.
Now we turn to the trained agent. During the process of fine-tuning the hyperparameters, we launched several trials trying to unearth the most promising ones for our purposes. It turned out that this set of hyperparameters in Table 4.1 turns out to be one of the most promising ones based on the potential balancing percentage, 18.68%; therefore, we decided to launch the training of length equal to the number of training episodes available (547) $\times$ 6. This value is dictated by the available computing power that allowed the entire process to be carried out in $\simeq$ 8-10 hours.

Let us now analyze the data from the training phase. The balancing percentage stands at 21.88% with an average reward of -1.1. As can be seen by Figure 4.3, the volumes traded decrease over time, underscoring the agent's tendency to reduce trades in favor of higher reward; the same can be seen clearly from the volumes traded in absolute value

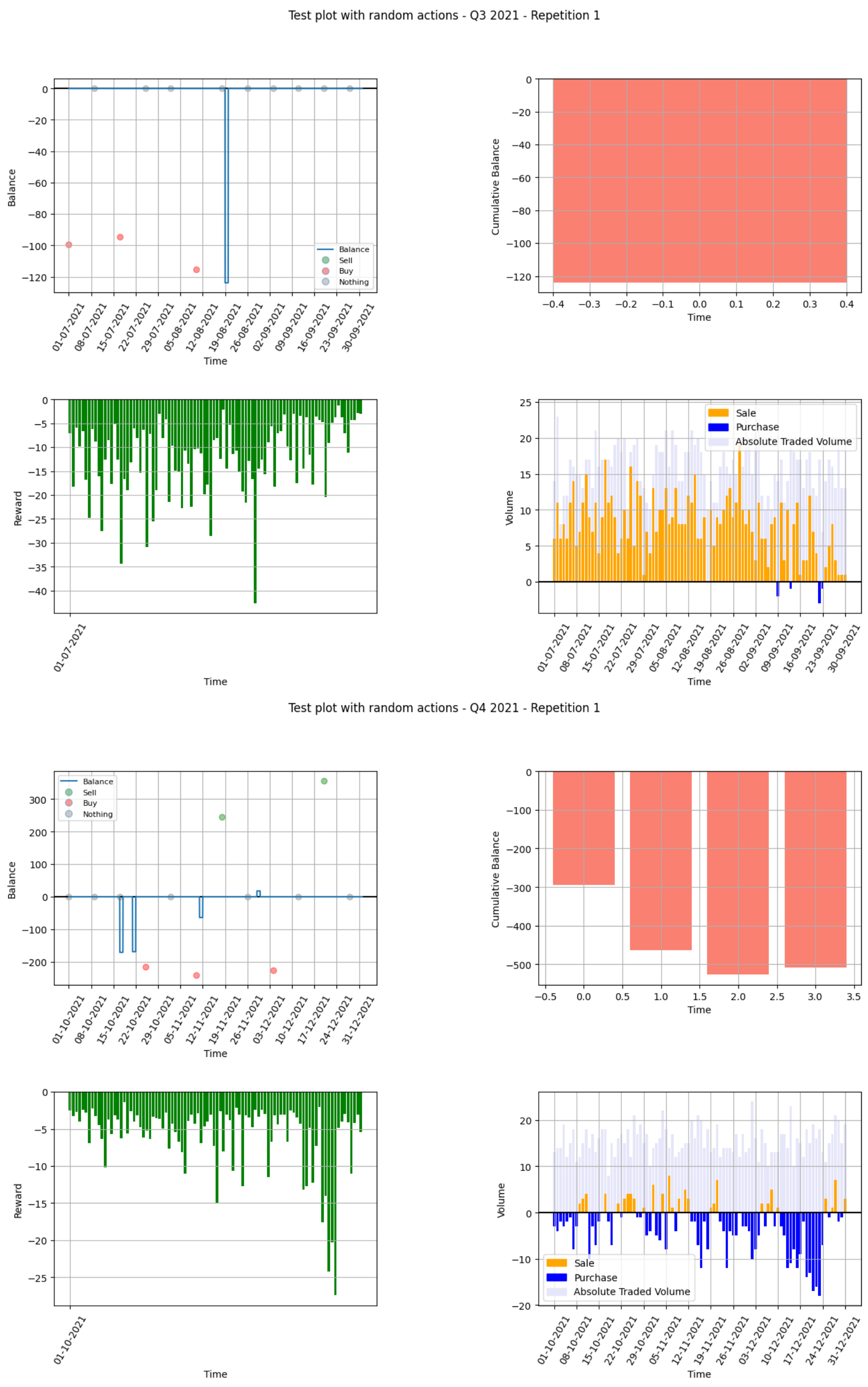


Figure 4.2. Q3 and Q4 rendered figures with 0.2€fee - Random Agent

Table 4.1.

Hyperparameter	Value
Batch size	1440
γ	0.95
Learning rate	0.0003
Entropy Coefficient	0.01
ϵ	0.2
Value Function Coefficient	0.5

(middle graph), which definitely decrease by remaining between 2 and 4 MWh - 1 buy and 1 sell order in the first case, 2 buy and 2 sell in the second-. This shows us that the agent has learned or at least is learning to reduce volumes, balancing episodes, to maximize reward: good. The cumulative reward graph, the one further down, is more complex to interpret. The reader may wonder how it is possible to believe that one has achieved good results if the cumulative balance is always negative, but this can be a confusing concept. In fact, what one expects from the balance plot is not to observe immediate gains, but to see that after an initial loss phase the graph flattens out and then slowly begins to rise. This characteristic begins to be glimpsed in the right end of the graph, which has reached a flat level. However, it is not possible for us to argue anything else about it, because there is no evidence: simply the graph suggests increasing the training period, which for timing issues we will leave in the section 5.1. Well, we just have to test the model and observe the final results. Below we find the plots rendered by Gym. Upon initial analysis of the graphs, the reader should be able to realize the complexity of this task. The result is surprising: the balance is never altered and therefore it is possible that episodes are unbalanced or that in fact, the Gym never plays. It is the latter. In fact, looking at the bottom right graph of the first figure above we can see that for more than half of the third quarter the blue bars in the background do not even appear, a feature also confirmed by the gray dots in the top left graph. This means that the agent we have been training has in fact learned to "not play," that is, to generate prices that are never actually matched on the LOB and for that reason never trade any power volume. This is a rather interesting partial result, and we explain it through 3 non-mutually exclusive conclusions:

- the training period had to be longer and therefore 6 whole repeated rounds of the training set is not enough
- the chosen set of hyperparameters is not suitable
- the 0.2€ fee is too penalizing and pushes the agent to adopt a too conservative strategy

The last one from an implementation point of view is the most immediate to change, and for our purposes, in the next subsection, we will try to cancel the fee by pushing the agent toward a less conservative strategy more aimed at trading volumes. It is also useful to note that from the second half of Q3 until the end of Q4 the agent encounters greater

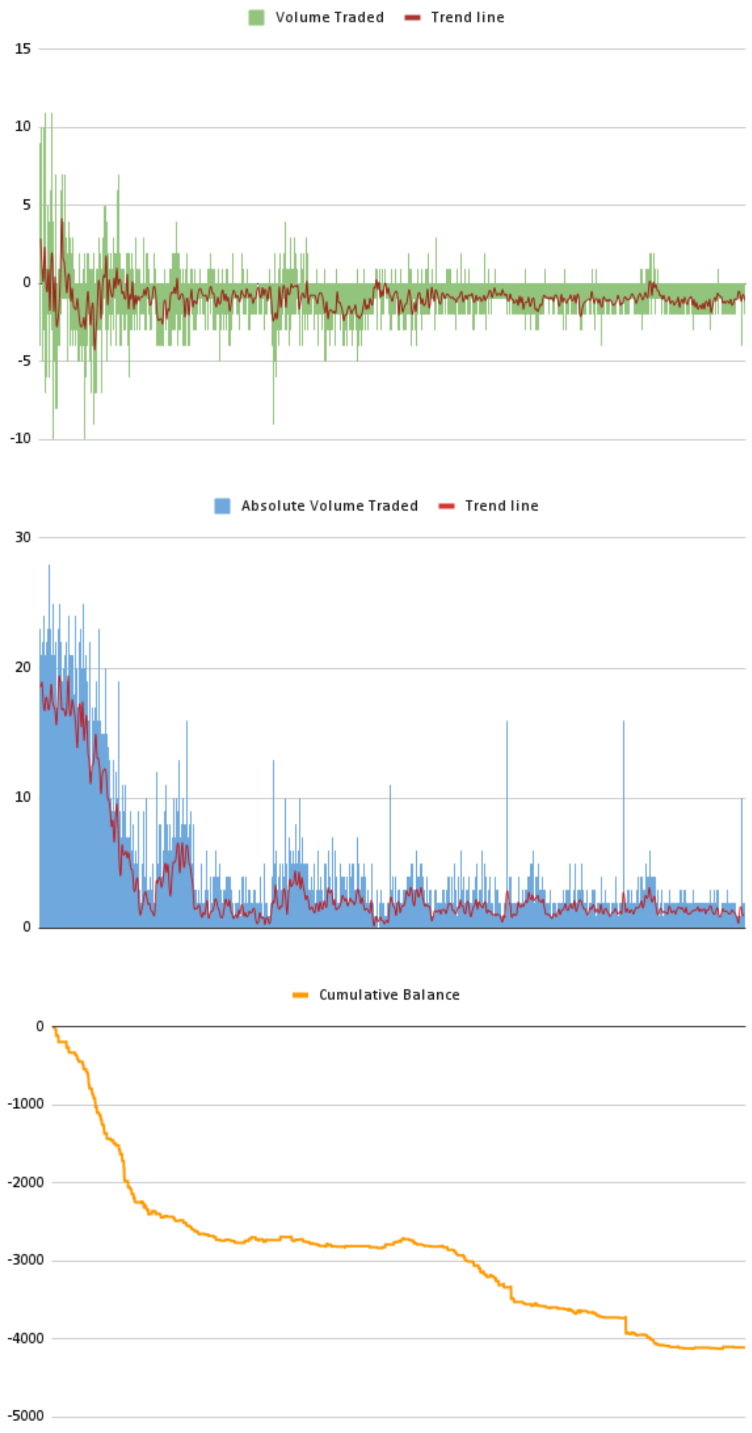


Figure 4.3. Training graphs with hyperparameters of the Table 4.1 with 0.2€ fee

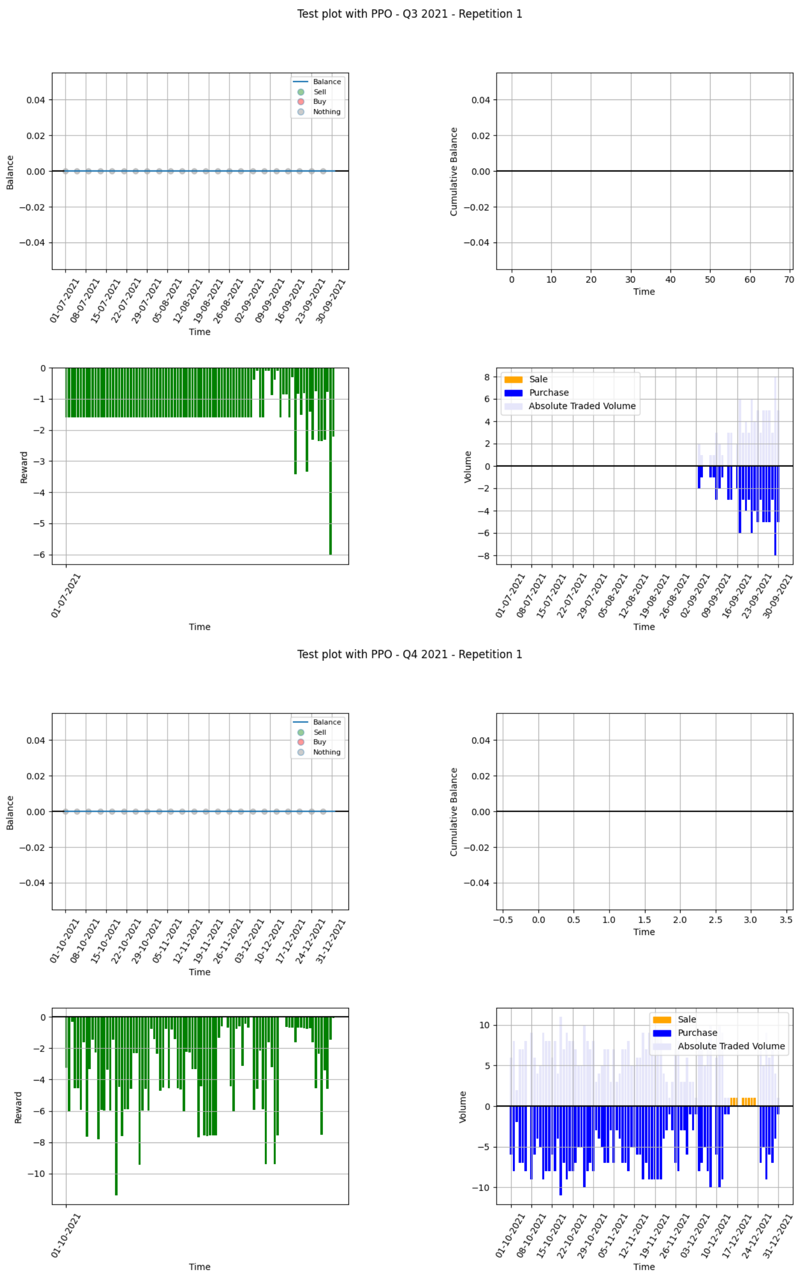


Figure 4.4. Q3 and Q4 rendered figures with 0.2€ fee - Trained Agent

balancing difficulties, exactly at the substantial price change highlighted by Figure 4.1. In conclusion, the agent managed to balance as much as 39.1% - 72 out of 184 - of the total episodes, showing in part that this type of rewarding assigned to him caused him to learn a more conservative trading strategy.

4.2.2 Evaluation without fee cost

Let us now analyze a random agent that this time uses a reward signal that no longer includes any fee cost; this will allow the agent to no longer receive large penalties due to opening a position, giving the agent being trained in the future the ability to take on more "risky" strategies. In fact, recall that the fee was introduced every time a buy or sell order was posted and an actual match was encountered on the LOB, all with the purpose of curbing the agent from posting orders where they were not needed, introducing precisely this additional penalty. Now, this argument applies to a trained agent, but not to the random agent we are now going to analyze; however, it is good to remind the reader what the considerations are from which we start so that the choices made can be fully understood. Speaking of the agent performing random actions as in the previous subsection, Figure 4.5 shows the rendering result at the end of the test period. First of all it immediately jumps out from the Q3 balance graph that there are no balance alterations of any kind. This is not because the agent took no action, as can be seen from the red and green dots, but rather because in no episode did the agent manage to close the position with a total overall volume traded of 0. In fact, we notice this from the graph of imbalances at the bottom right: the light blue lines in the background actually indicate that volumes were traded, but in no case did the agent get to 0: this meant that there were no actual gains or losses. Recall here that in a real environment this would in any case correspond with economic losses, but in our case, because of the way we set up the "rules of the game," we decided to consider gains and losses only as balanced episodes, trying to direct the agent first to learn this dynamic. The matter changes for the Q4 graph where we can see as many as 7 changes in the total balance (one very small one); as one can quickly guess one has managed to close positions as many as 7 times following the percentage of balancing that amounted to 3.8% remains very low. However, this does not mean having had gains, quite the contrary: as the top right graph indicates an initial phase where the random agent seems to be gaining reaching a maximum peak of 259.24€ at the 103rd episode is followed by a second phase of loss that will lead to a final total negative balance of 344.45€ in loss. This clearly shows that trading with random strategies is unsuccessful. Lastly, if we look at the reward graph, as in the previous subsection we note negative and constant values with average -6.05, with frequent downward peaks; in addition to these there are rare exceptions with slightly positive values between 0.2 and 0.3: we can think that in correspondence of these the agent managed to obtain small intermediate gains, which, however, did not lead him to positive results.
Let us now turn to the analysis of the trained agent. Again, as in the case of the 0.2€ fee, we performed a fine-tuning process with 300 trials over a period of only one quarter. The best set, based on the balancing percentage which is 25.71%, is represented by Table 4.2.

Once this set was selected we trained the model for a total of 7 total runs (547×7). Again, longer training would probably have been more optimal, however, this choice is given the computational power available. Figure 4.6 shows the graphs of the training stage. First, from the graph of volumes traded we can observe that initially we have a tendency in the initial episodes to unbalance, after which the agent learns not to do so and therefore we observe a reduction in volumes traded. However, this trend is not

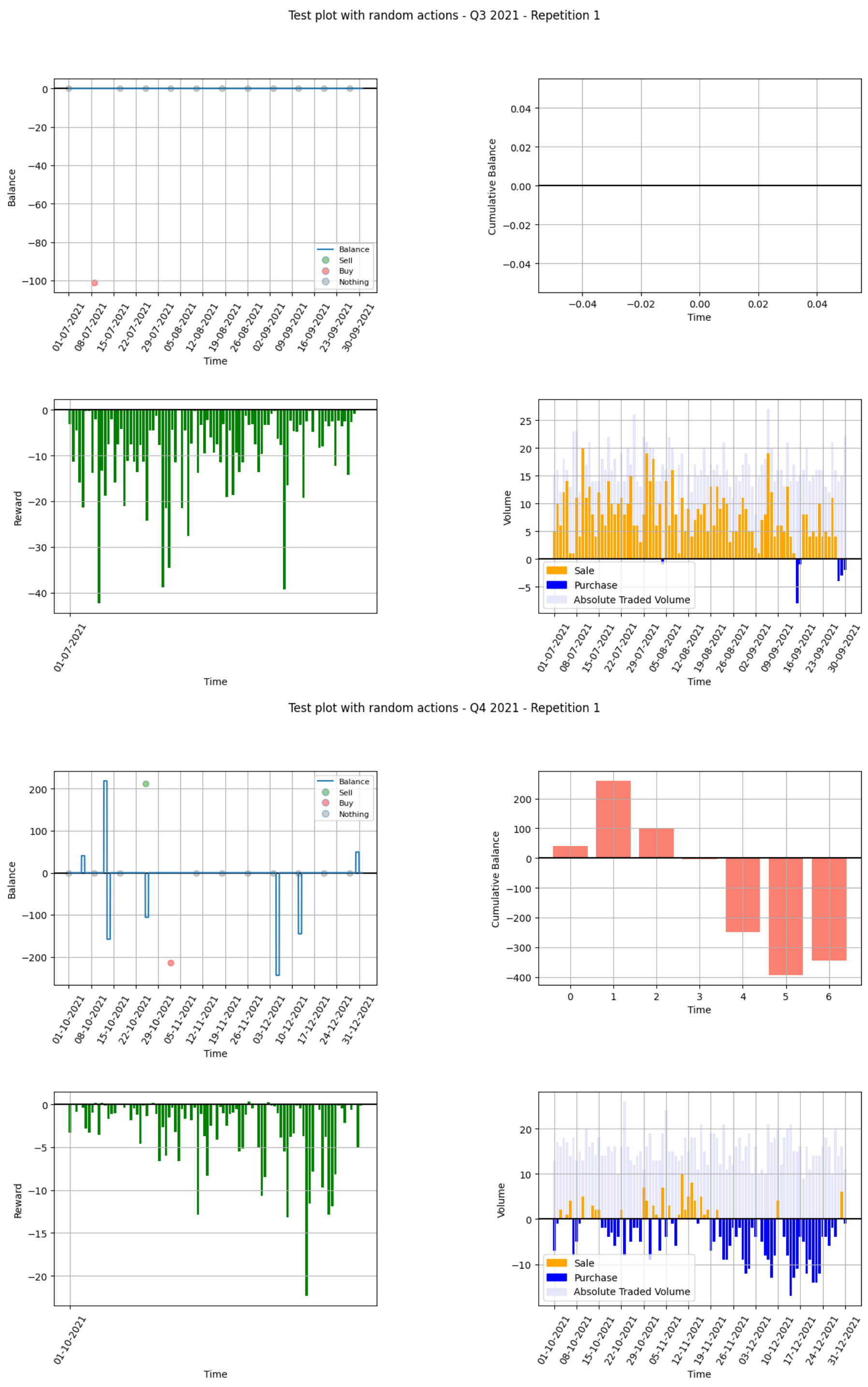


Figure 4.5. Q3 and Q4 rendered figures without fee

Table 4.2.

Hyperparameter	Value
Batch size	1440
γ	0.995
Learning rate	0.0003
Entropy Coefficient	0.00001
ϵ	0.4
Value Function Coefficient	0.2

progressive until a plateau is reached; rather, as the trend line shows, there is a period true the second half of the bar graph in which the agent seems to unbalance again during the episodes. We explain this behavior through the fact that the training in the RL is extremely susceptible to the choice of hyperparameters, and specifically, the ones we selected seem to make the agent learn fairly quickly, but nevertheless subject it sometimes to noise that slightly destabilizes it. In any case however even after these periods the agent returns to balance, which is exactly what we want it to learn in the first place. From the central graph of volumes traded in absolute value we observe a very important fact: the agent trades a lot of volumes and therefore places a lot of orders. Whereas before, with the fee at 0.2 he learned not to play, here he adopts a less conservative strategy and therefore tries to place many orders: the quality of this behavior can only be judged when the model is evaluated. However, it is good to know that on average it trades about 17-18 MW of electricity and has a balancing percentage of 27.57%. From the last graph below we can draw the same conclusions as in the previous subsection, in fact we cannot observe anything except that longer training theoretically should allow the cumulative balance to reach a plateau and after a certain period begin to rise very slowly. It is now time to test the model and observe the results. The first evidence is that where the agent encounters prices never seen during training it prefers not to trade and therefore freezes: this is from the second half of Q3 and for most of Q4; in this regard we find a few exceptions in the bottom right graph of Q4, but still not affecting the balance. The graph that certainly jumps out most to everyone's eye is that of the balance; at first there is a small loss after which there is a gain in the positive which attests to the overall balance at +7€. Although the amount is small it is a first result and it is interesting to observe how once it is reached it is not altered. So overall we find a positive final balance, with a high balance percentage 53.2% and an average reward of -5.6. Surely this is the best result achieved but it would, however, be more interesting still to be able to test it if the price outliers due to the pandemic and the outbreak of war were included in the training set.

4.3 Overall comparison of results

Let us now summarize the results obtained and compare the models with each other

As you can see from the Table 4.3 the one with the worst performance is the random

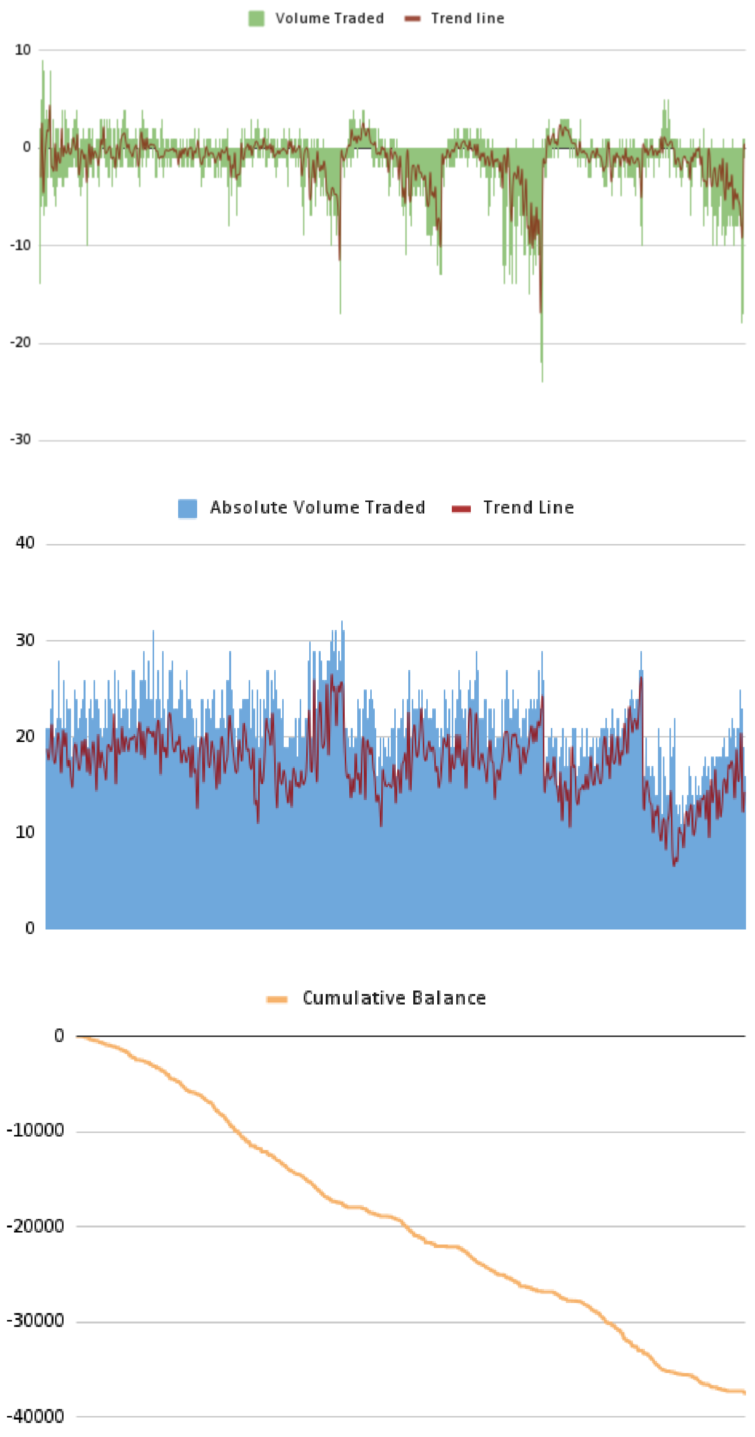


Figure 4.6. Training graphs with hyperparameters of the Table 4.2 without fee

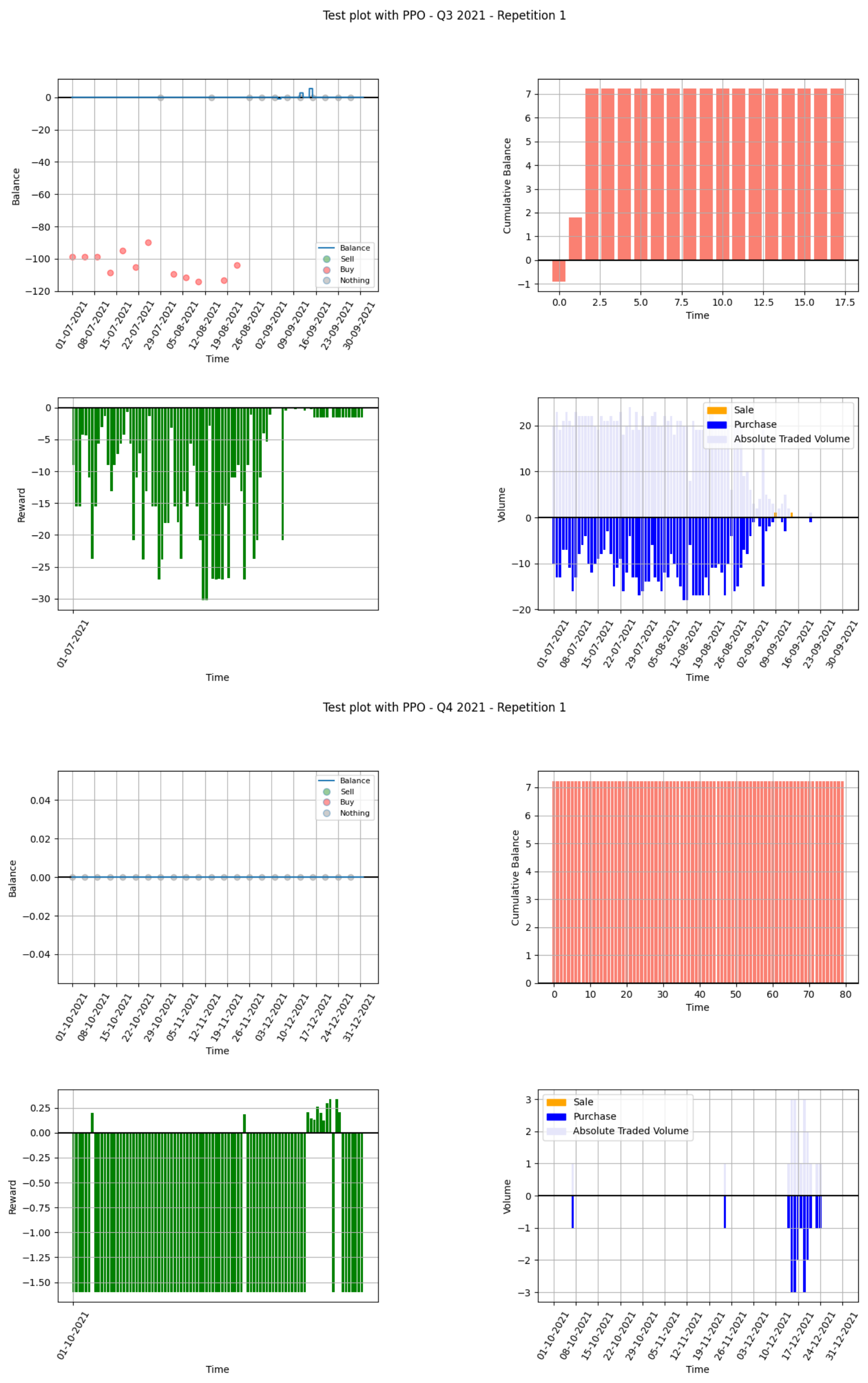


Figure 4.7. Q3 and Q4 rendered figures without fee - Trained Agent

Rewarding	0.2€ fee		No fee	
Agent	Random	Trained	Random	Trained
Balancing percentage	2.7%	39.1%	3.8%	53.3%
Total net profit	-509.03€	—	-344.35€	7.23€
Mean Reward	-9.09	-2.63	-6.05	-5.6

Table 4.3. Performance comparison table

agent with the 0.2€ fee, as we expected. The random actions in fact do not allow the agent to focus on the positions to be closed, and for those few that he managed to balance at the end of the episodes he made bad trades closing with a very negative final balance. This, however, is not very useful except as a benchmark for trained agents. In fact, the comparison with the agent coached with the same rewarding system is totally in favor of the latter, with almost 15 times as many balanced episodes. However, this agent does not have very good total performance because he actually learns not to trade to avoid receiving too negative rewards given by the fee. Then we move on to the next random agent with a rewarding system without a fee, right away we notice that here too the percentage of balancing is very low and that the final cumulative balance is clearly negative, again as we expected. Let us now focus on the trained agent. The balance percentage is the best of our 4 examples and is at a remarkable value, with more than half of the episodes balanced. Also, although small, the final balance is positive, indicating that we were on the right track. In the next section we will discuss what could have been done to improve the performance of the model, using this last result as a basis.

Chapter 5

Conclusions

5.1 Future Outlook

This work allowed us to exploit the advantages of the reinforcement learning paradigm to be able to perform a rather complicated task: trading. In this sense, the primary desire was to demonstrate that an agent trained under the right assumptions could be capable of trading in the continuous intraday electricity markets.
For issues due to the difficulty of the task itself and the emulation of a real environment, such as price fluctuations due to contextual data, we had to restrict our scope to a more ordinary and therefore less 'real' situation. This choice led us to obtain the results seen in the previous section. But in order to obtain results that are more indicative and linked to real markets, first of all we could try to take into account a more complex environment by including in the space of the observations data backed by better performing regression models, which try to constitute a LOB that is less artificial than the one constructed by us, but more truthful. Since the space of observations in our Gym is easily manipulated, in addition to inserting more realistic LOB data, one could also try to insert a forecast on the energy production of the renewable plants, so as to have forecasts on subsequent costs. A further step would be to allow our Gym to be able to post orders with volumes that have respectively continuous values and not fixed at 1 as in our case. Removing this limitation would ideally allow our agent to develop more advanced and certainly complex strategies.
An additional point to be made is that our agent was evaluated over a 6-month test period from July 1, 2021 to December 31, 2021, however, it might be very interesting to evaluate it, after improving it by removing the simplifications currently present, over longer test periods that are more plagued by outliers and sudden price fluctuations due to contextual events such as wars, economic bubbles, and so on.
A further idea that could be developed further is to change rewarding. In section 4.2 of this chapter, we observed how the introduction or deletion of a cost due to a fee essentially induces the agent to behave differently by either not placing orders on the market with a more conservative strategy or by increasing the number of orders and consequently increasing the risk of loss and gain. It would be interesting to try evaluating a model with a rewarding function given at the end of each episode and no longer progressive step

by step; or to completely change the logic, perhaps by inserting a maximum budget that can be spent by the agent to see which strategies can be successful. Furthermore, one could try to structure a reward entirely based on success in terms of profit, for example with a +1 in case of a gain or -1 in case of a loss at the end of the episode or even at the end of an entire quarter. This feature leaves room for experimentation that could lead to a smarter, more complex and more strategic model.

5.2 Final Considerations

The aim of this thesis was to use the reinforcement learning paradigm to create an agent capable of devising more or less complex trading strategies that outperform trivial strategies. To do this, the entire process was developed step by step, first by constructing an environment as close to reality as possible, and then by training an agent under certain considerations.
In order we used the Geometric Brownian Motion to generate simulations of price fluctuations from the average electricity prices of the Italian market in 2020-2021. Next we looked at the problems involved in reconstructing the Limit Order Book, a fundamental tool that regulates supply-side and demand-side transactions in the market; we gave an accurate description of how it works and what assumptions we made to construct one, again, similar to reality. We then saw what Extreme Gradient Boosting is and how it is used, a regression technique capable of providing us with real energy price forecasts with certain margins of error and accuracy.
The most copious part of the entire treatise is dedicated to the Gym environment and agent learning through Proximal Policy Optimization: the entire parts are drafted in as much detail as possible in order to provide every useful tool for understanding the subject and every nuance on this specific task, given its great potential and future developments. We have seen how these learning mechanisms work and, during development, we do not deny it, we sometimes stumbled over its implementation given the complexity of the task. At the end, we analysed the results and compared the various strategies.
Therefore, we have seen and discussed the potential of reinforcement learning on this specific task, leaving room for further investigation in this regard, once the restrictions we applied are removed.

Bibliography

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[2] Alexisbcook. XGBoost. *Kaggle*, 4 2023.

[3] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 3 2016.

[4] Rüdiger Kiesel and Florentina Paraschiv. Econometric analysis of 15-minute intraday electricity prices. *Energy Economics*, 64:77–90, 5 2017.

[5] Christopher Koch and Lion Hirth. Short-term electricity trading for system balancing: An empirical analysis of the role of intraday trading in balancing Germany's electricity system. *Renewable and Sustainable Energy Reviews*, 113:109275, 10 2019.

[6] Malte Lehna, Björn Hoppmann, René Heinrich, and Christoph Scholz. A reinforcement learning approach for the continuous electricity market of germany: Trading from the perspective of a wind park operator. *CoRR*, abs/2111.13609, 2021.

[7] Henry Martin and Scott Otterson. German Intraday Electricity Market Analysis and Modeling Based on the Limit Order Book. *International Conference on the European Energy Market*, 6 2018.

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 6 2016.

[9] Michał Narajewski and Florian Ziel. Econometric modelling and forecasting of intraday electricity prices. *Journal of Commodity Markets*, 19:100107, 12 2018.

[10] Michał Narajewski and Florian Ziel. Ensemble forecasting for intraday electricity prices: Simulating trajectories. *Applied Energy*, 279:115801, 12 2020.

[11] Gymnasium Documentation.

[12] Plato. Ion. http://classics.mit.edu/Plato/ion.html.

[13] Plato. The republic. http://classics.mit.edu/Plato/republic.html. Books II, III, and X.

[14] Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations - Stable Baselines3 2.0.0a13 documentation.

[15] What is XGBOOST? | Data Science and Machine Learning.

[16] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[18] Zijian Shi, Yu Chen, and John Cartlidge. The LOB Recreation Model: Predicting the Limit Order Book from TAQ History Using an Ordinary Differential Equation Recurrent Neural Network. *Proceedings of the ... AAAI Conference on Artificial Intelligence*, 35(1):548–556, 5 2021.

[19] Viktor Stojkoski, Trifce Sandev, Lasko Basnarkov, Ljupco Kocarev, and Ralf Metzler. Generalised Geometric Brownian Motion: Theory and Applications to Option Pricing. *Entropy*, 22(12):1432, 10 2020.

[20] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks*, 16(1):285–286, 1 2005.

[21] Bartosz Uniejewski, Grzegorz Marcjasz, and Rafał Weron. Understanding intraday electricity markets: Variable selection and very short-term price forecasting using LASSO. *International Journal of Forecasting*, 35(4):1533–1547, 10 2019.

[22] Wei Wang and Nanpeng Yu. A Machine Learning Framework for Algorithmic Trading with Virtual Bids in Electricity Markets. 8 2019.

[23] Zihao Zhang, Stefan Zohren, and Stephen J. Roberts. DeepLOB: Deep Convolutional Neural Networks for Limit Order Books. *IEEE Transactions on Signal Processing*, 67(11):3001–3012, 6 2019.