ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Artificial Intelligence in Industry

DEALING WITH LONG-TERM CONSTRAINTS IN A HYBRID LEARNING AND OPTIMIZATION METHOD

CANDIDATE
Diego Chinellato

SUPERVISOR
Prof. Michele Lombardi

Academic year 2023-2024

Session 6th

To Giorgia.

Abstract

The increasing complexity of real-world decision-making problems, where long-term reasoning capabilities are required, is driving novel research on the integration of existing approaches. With this purpose, this thesis explores the synergies between Mathematical Optimization (MO) and Machine Learning (ML), focusing on integrating Reinforcement Learning (RL) with Constrained Optimization (CO) for complex decision-making problems. Our starting point is the UNIFY framework, an hybrid method comprising an offline ML-based phase as well as an online CO-based phase. Our core contribution is the extension of this framework to handle cumulative (long-term) constraints by re-casting the problem as a Constrained Markov Decision Process (CMDP) and employing Lagrangian relaxation methods for its solution. We conduct empirical evaluations on an Energy Management System (EMS) tasked with optimal power flow allocation under long-term sustainability constraints. The results reveal: (1) the critical role of virtual parameters modeling in the learning process; (2) the challenges associated with the choice of the online optimization problem; and (3) the importance of hyperparameter tuning in optimizing the framework's performance. Overall, the proposed approach can offer strategic foresight allowing for proactive preparation for future requirements, while also retaining good performances even when dealing with strict constraints.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, there has been a ever-increasing collaboration on interdisciplinary research between the Mathematical Optimization (MO) community, in particular that of Constrained Optimization (CO), and the Machine Learning (ML) community. While MO has long been established as a discipline focused on the formulation of mathematical models which are then employed to solve decision-making problems, ML has emerged more recently as a powerful framework for data-driven decision-making and pattern recognition. These two approaches excel in their respective domains, yet their convergence may yield a new realm of possibilities for solving complex problems that neither could efficiently tackle alone. For instance, ML methods tend to be scalable and capable of generalizing to unseen data, but they also often feature a black-box nature hindering explainability and their performances heavily depend on the quality (and quantity) of available data; on the other hand, MO methods usually have optimality guarantee on solutions found and they are often more interpretable than black-box ML models, but are sensitive to modelling assumptions and may be extremely computationally expensive, such as when dealing with non-convex or combinatorial problems. We believe this complementary nature of ML and MO methods to be a strong motivation behind working towards tighter integration of these fields.

This synergy between MO and ML, especially within the domain of Reinforcement Learning (RL) and (stochastic) Constrained Optimization, embodies the core focus of this thesis: we consider a hybrid learning and optimization framework and equip it with the ability of dealing with long-term constraints. The primary motivation behind this work arises from the increasing complexity of real-world problems, where an AI agent tasked with solving such complex problems would indeed need the capability to make decisions in an environment with constraints over a long horizon (i.e. long-term reasoning); these constraints could range from budgetary limitations in operations research to emission caps in environmental management systems, posing significant challenges to traditional problem-solving methodologies which may act in a myopic way.

The intersection of MO and ML, particularly through Constrained Reinforcement Learning (CRL), offers a promising avenue for addressing these challenges: CRL methods aim at learning optimal policies in environments where actions are not only evaluated based on rewards but must also adhere to specific constraints. This requirement mirrors real-world scenarios much closer than unconstrained environments, but it also introduces added complexity during the learning process.

Given the significance of long-term constraints in many practical applications, this thesis extends a recently proposed framework called UNIFY [58] to handle long-term constraints. UNIFY is an hybrid learning and optimization framework for complex decision-making problems which works by decomposing the problem into two stages, namely an offline ML model and an online CO model[58]. In this thesis, we extend this framework by slightly altering the general problem formulation and employing CRL methods to overcome the myopic behaviour of traditional CO methods.

All the code developed for this thesis can be found at the following GitHub repository: `https://github.com/diegochine/thesis`

The remaining of this thesis is structured as follows:

- **Chapter 2: Background and preliminaries.** This chapter deals with the mathematical preliminaries required for understanding the subsequent work. A general overview of the MO and ML fields is provided, with a focus on Reinforcement Learning (RL). Related works and constraints taxonomy are also provided in this chapter.

- **Chapter 3: Methods.** In this chapter a brief introduction to the UNIFY framework is provided, which is then reformulated to accommodate cumulative constraints. We also describe the RL algo that we employ to learn a constraint-satisfying policy, PPOLag.

- **Chapter 4: Empirical evaluation.** This is the empirical chapter: the description of the case study considered in our experiments (EMS) is provided first, followed by the experiments that have been carried out. The experimental setup as well as the evaluation methodology are also discussed in this chapter.

- **Chapter 6: Conclusions.** This chapter summarizes the thesis by synthesizing the findings and discussing their implications and the challenges encountered. We conclude by highlighting how the proposed framework contributes to the existing body of research and describing some possible avenues for future research.

Chapter 2

Background and preliminaries

In this chapter, a comprehensive theoretical overview of the foundational concepts required for understanding the hybrid approach proposed in this thesis is provided.

This chapter is structured as follows: the first section describes the general problem of Mathematical Optimization, as well as presenting various common optimization techniques and methods ranging from linear programming to stochastic optimization; we also describe the method of Lagrange Multipliers, a pivotal tool for solving constrained optimization problems. Following the overview on mathematical optimization, Section 2.2 delves into the realm of machine learning, specifically focusing on Reinforcement Learning (RL). Within this domain, we describe some of the most common RL algorithms. Then, by building upon the basic concepts of RL, we describe the Constrained RL problem, wherein agents must act in environments subject to constraints of different nature. We present some related works in Section 2.3. Given that a very large amount of different paradigms for integrating ML and CO have been developed (and are still being actively studied), we focus on the method that we find most closely related to our proposed framework, namely Decision-Focused Learning. Finally, in Section 2.4, we present a taxonomy of constraints, categorizing them based on their characteristics and implications for decision-making processes. This taxonomy serves as a foundational framework for understanding the diverse array of constraints encountered in real-world applications and guides the subsequent discussions on the integration of constraints into reinforcement learning frameworks.

2.1 Mathematical Optimization

Mathematical Optimization (also known as **Mathematical Programming**) is a foundational pillar of the Artificial Intelligence discipline that provides essential methods and techniques for modelling decision-making processes, and finds application in many different fields ranging from Machine Learning to financial applications. From a high-level perspective, it involves finding the best possible solution to a given problem from a set of alternatives.

Mathematically, an optimization problem involves the minimization (or maximization) of a function; a prototypical optimization problem can be expressed as:

$$\min_{x \in \mathbb{A}} f(x) \tag{2.1}$$

where $x = (x_1, ..., x_d)$ represents the vector of decision variables, $f : \mathbb{R}^d \to \mathbb{R}$ is the objective function and $\mathbb{A}$ is the search space, i.e. the set of admissible values that x may assume. Usually $\mathbb{A}$ is specified by means of equalities and inequalities involving constraint functions, i.e.:

$$\begin{aligned} \min_{x} \; & f(x) \\ \text{s.t. } & g_i(x) \leq 0, \; i = 1...n \\ & h_i(x) = 0, \; i = 1..m \end{aligned} \tag{2.2}$$

where $g_i(x)$ are the inequality constraint functions and $h_i(x)$ are the equality constraint functions. Then, the goal is to identify the optimal set of decision variables that minimizes (or maximizes) $f(x)$, that is: finding an element $x^* \in \mathbb{A}$ such that either $f(x^*) \leq f(x) \forall x \in \mathbb{A}$, for a minimization problem, or $f(x^*) \geq f(x) \forall x \in \mathbb{A}$, for a maximization problem. Note that by convention we consider only minimization problems, as minimizing a function $f(x)$ yields the same result as maximizing $-f(x)$, its reflection about the x-axis, since $f(x^*) \geq f(x) \iff -f(x^*) \leq -f(x)$; this means we can essentially treat minimization problems and maximization problems in the same way, so without loss of generality we only consider minimization problems from now on. Furthermore, the above definition of x^* is that of a **global minimum**; that is, x^* is the best possible solution out of the entire feasible set $\mathbb{A}$. From a practical point of view, however, for many classes of optimization problems finding the global optimum may be extremely complicated, for instance when the objective function is a highly nonlinear function; in such cases, we may instead obtain a **local optimum**, that is, a value $x^* \in \mathbb{A}$ such that $f(x^*) \leq f(x) \, \forall x \in \{x \in \mathbb{A}$ s.t. $||x - x^*|| \leq \delta\}$, $\delta > 0$. In other words, a local optimum is only better than solution in some local neighbourhood of itself, and may be as well as not be a good enough solution.

There are several dimensions that we may use to classify optimization problems, resulting in many different subfields; we briefly discuss them before moving to analyzing some particular classes of optimization problems. A main property to consider is convexity: a function f is convex if $\forall x, y$, $\forall \alpha \in [0, 1], f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y)$. A **Convex optimization** problem is one where the objective function as well as all constraint functions are convex, whereas a **nonconvex optimization** problem is one where the objective function or any of the constraint functions are not convex. Convex optimization is much easier w.r.t. nonconvex optimization: there are algorithms running in polynomial-time which can provide the optimal solution to many classes of convex problems [43], while the general problem of mathematical programming is NP-hard [42, 52, 45].

The type of the variables of the problem also heavily influences the applicable methods: if the variables are continuous, we are facing a **continuous optimization** problems, while **discrete optimization** deals with problems with discrete variables. Continuous optimization tends to be easier than discrete optimization, as the smoothness of the functions (objective and constraints) involved in continuous optimization allows to employ calculus methods which cannot be used to solve discrete problems.

The nature of the admissible set $\mathbb{A}$ also plays an important role: if no constraints are specified, that is if $\mathbb{A} = \mathbb{R}^d$, then the problem is an **unconstrained optimization** problem; conversely, if $\mathbb{A} \subset \mathbb{R}^d$, then we are dealing with **constrained optimization**.

Finally, **deterministic optimization** assumes that no stochasticity is involved in the model; however, in many practical applications there may be varying degrees of uncertainty, and such problems are called **stochastic optimization** problems.

Linear Programming

One of the most widely applied methods is that of **Linear Programming** (**LP**): a LP problem is one where the objective function as well as the constraint functions are linear functions, that is, a problem of the form:

$$\begin{aligned} &\min_x c^\intercal x = \sum_i c_i x_i \\ &\text{s.t. } \sum_i a_{ji} x_i \leq b_j,\ j = 1...n \\ &\quad\ \sum_i a_{ki} x_i = b_k,\ k = 1...m \\ &\quad\ \sum_i a_{li} x_i \geq b_l,\ l = 1...o \end{aligned} \tag{2.3}$$

Note that every LP problem can be rewritten in a compact, matricial form known as **Standard Form of an LPP** [48]:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} c^\intercal x \\ &\text{s.t. } Ax = b \\ &\quad x \geq 0 \end{aligned} \tag{2.4}$$

where $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$ with $n > m$. Also note that, since linear functions are both convex and concave, LP is a particular case of convex programming.

Several methods have been developed to solve LPPs. The **Simplex method**, first proposed in 1947 [16], works by noting that the feasible region (i.e. the set of values for which $Ax = b, x \geq 0$ holds) is a (possibly unbounded) convex polytope. Since it can be shown that, given an LPP in standard form, if the objective function has a minimum value inside the feasible region then this value must lie in (at least) one of the extreme points (vertexes) of the polytope, the simplex method then starts from one vertex and iteratively traverses the edge of the polytope until an optimal solution is found [12]. Although the Simplex method has a exponential worst-case time complexity, from a practical point of view it has proved to be efficient on a wide array of problems. Differently from the simplex method, **Interior Point methods** work by traversing the interior of the feasible region [51]. First proposed in [34], this class of methods has been widely studied and extended, with the current SOTA method in terms of theoretical time complexity being the one proposed in [30]; unlike the simplex methods, Interior Point methods have polynomial worst-case time complexity. Still, even though Interior Point methods are theoretically more efficient, empirical studies have shown comparable performances between the two approaches in most cases [28].

Quadratic Programming

Another widely used approach is that of **Quadratic Programming** (**QP**), which deals with the optimization of a quadratic objective function subject to linear constraints. The general QP problem can be formulated as follows:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} \frac{1}{2} x^\intercal Q x + c^\intercal x \\ &\text{s.t. } Ax \preceq b \end{aligned} \tag{2.5}$$

with $Q \in \mathbb{R}^{n \times n}$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$; $\preceq$ is the component-wise inequality, i.e. $\forall i \in \{1..m\},\ Ax_i \leq b_i$ with v_i being the i-th element of vector v.

QP problems are in general much harder to solve than LP, given that the objective function in QP may or may not be convex; the general problem is NP-hard [52]. However, if matrix Q is positive definite, then the problem is convex [35] and can be solved in polynomial time (see e.g. [35, 32]). For the general case, a wide variety of methods have been proposed, such as Augmented Lagrangian methods [18] and Interior Point methods [66].

Integer Programming

An optimization problem that requires the variables appearing in the problem to be integers is called an **Integer Programming** (**IP**) problem. If this type constraint is required only for a subset of all the variables, then we refer to the problem as a **Mixed Integer Programming** (**MIP**) problem. Usually, the linear case is considered (i.e. with linear objective function and linear constraints, escluding the integer constraint); in this case we call the problem **Integer Linear Programming** (**ILP**) in the IP case, or **Mixed Integer Linear Programming** (**MILP**) in the MIP case. A common formulation of an ILP problem is the following:

$$\begin{aligned} &\min_{x \in \mathbb{Z}^n} c^\intercal x \\ &\text{s.t. } Ax \preceq b \\ &x \geq 0 \end{aligned} \tag{2.6}$$

with $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$.

IP is known to be NP-complete: a simple variant of the problem involving no optimization (only satisfaction) and binary variables is actually part of Karp's 21 NP-complete problems [33]. Thus, a wide variety of approaches and methods for solving IP problems have been developed and studied.

The simplest approach for solving an ILP is to consider its *LP relaxation*, that is, to drop the integral constraint and solve the resulting LP problem using the simplex algorithm. While this simple approach does not work in the general case (given that the solution to the LP relaxation is not guaranteed to be the optimal solution to the original ILP problem, or even to be feasible at all), for a particular class of IP problems (those in which the matrix A is totally unimodular [69]) the solution of the LP relaxation returned by the simplex algorithm is guaranteed to be integral and this approach may be used.

Concerning the general case, both exact algorithms and heuristic methods have been developed for IP. One commonly used class of exact methods is that of **Branch and Bound** methods, first proposed in [36]. The core idea of B&B methods is to recursively split the feasible region in smaller regions (*branching*) and then optimize the objective function on these smaller regions; moreover, the method keeps track of bounds (*bounding*) that the optimum must satisfy (based on the search so far), in order to prune candidate solutions that for sure are not optimal solutions. This general schema has been extended and combined with other methods, for instance with the *Cutting Planes* method [41] resulting in the Branch&Cut method [44].

However, IP belonging to the NP-hard class implies that more difficult problem instances are intractable for exact algorithms - in the worst case, B&B methods have to explore the entire search space, which is exponential in the input size [72] - hence one must resort to heuristic methods. Many approaches have been studied, for example the application of tabu search [40], surrogate constraints heuristic [25] or even problem-specific heuristics such as the *k-opt* heuristic for the Travelling Salesman Problem [37, 38].

Stochastic optimization

The optimization models that have been discussed up to now are all examples of models for deterministic optimization, that is, with no randomness or uncertainty involved in the modelling or solving process. Many practical applications, however, are subject to varying degrees of uncertainty; the field of **Stochastic Optimization** (**SO**) deals with such stochastic problems. Over the years, the name of the field has become an umbrella term for many different communities dealing with different variants of the general problem, as stochasticity and uncertainty may indeed arise from different components of the optimization problem (objective function, constraints, input parameters, ...). We briefly describe two important approaches in this class. *Stochastic Programming* (which may be seen as the ”stochastic variant” of deterministic LP) [15] deals with problems of the following form:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \; & c^\intercal x + \mathbb{E}_{\xi}[Q(x,\xi)] \\ \text{s.t. } & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.7}$$

where $Q(x, \xi)$ is the solution of:

$$\begin{aligned} \min_{y \in \mathbb{R}^m} \; & q(\xi)^\intercal y \\ \text{s.t. } & T(\xi)x + W(\xi)y = h(\xi) \\ & y \geq 0 \end{aligned} \tag{2.8}$$

This formulation is known as “two-stage stochastic programming problem”, since in the first stage (Equation 2.7) we optimize the cost $c^\intercal x$ plus the expected value of the optimal cost of second stage decision before obtaining a realization of ξ; in the second stage (Equation 2.8), a realization of ξ is obtained and based on it the first stage decision is refined and optimized, now accounting for the actual uncertainty.

Differently from Stochastic Programming, *Robust Optimization* aims at finding the best solution x for the worst-case realization of some uncertain parameter w. The Robust Optimization problem is usually formulated as follows:

$$\min_{x} \max_{w \in \mathcal{W}} F(x, w) \tag{2.9}$$

where $\mathcal{W}$ is called the uncertainty set, i.e. the set of possible realization for the unknown paramater w [49].

A unified framework for many SO methods and subfields has recently been proposed in [49], which the reader may refer to for further reference on the topic.

2.1.1 The method of Lagrange Multipliers

As we discussed earlier, Constrained Optimization (CO) involves optimizing a function subject to certain constraints. A general and thus widely used approach to address CO problems is the **method of Lagrange Multipliers** [5]; loosely speaking, this approach involves converting the constrained problem into an unconstrained one and finding saddle points of the newly defined objective function. Consider the following minimization problem:

$$\begin{aligned} &\min_{x} \ f(x) \\ &\text{s.t. } h(x) = 0, \end{aligned} \tag{2.10}$$

where for the sake of simplicity we consider a single equality constraint. The **Lagrangian function**, denoted as $\mathcal{L}(x, \lambda)$, is defined as:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda h(x) \tag{2.11}$$

where $\lambda \geq 0$ is the Lagrange multiplier associated with the constraint function $h(x)$. Then, the solution to the original constrained problem can be found by solving the following bilevel problem:

$$(x^*, \lambda^*) = \arg\max_{\lambda \geq 0} \min_x \mathcal{L}(x, \lambda) \tag{2.12}$$

Essentially, the Lagrange multiplier λ can be seen as a penalty term that is applied when constraints are violated; this violation is added to the original objective function (i.e. $f(x)$). For further reference on the method and its mathematical background, see e.g. [5].

2.2 Machine Learning

In this section we discuss about **Machine Learning** (**ML**), which refers to both a field of study within AI and to a large set of techniques and algorithms among the most commonly and ubiquitously used to develop systems for automatic decision-making without explicit programming. We start with a general overview of the three main approaches within ML, namely Supervised Learning (SL), Unsupervised Learning (UL) and Reinforcement Learning (RL).

Supervised Learning involves using learning algorithms to learn a model (i.e., a function) that can map input data to appropriate output labels; mathematically, let $\mathcal{X} \subseteq \mathbb{R}^k$ be a k-dimensional vector space, called *input space*, and $\mathcal{Y}$ be the output space, such that each $x \in \mathcal{X}$ is associated with a $y \in \mathcal{Y}$, then the goal of SL is to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that approximates the true, unknown mapping $g : \mathcal{X} \rightarrow \mathcal{Y}$ as closely as possible. Several types of SL exists; for instance, if the output space is a finite set of discrete values, then we are dealing with a classification task, whereas if the output space is the set of real numbers $\mathbb{R}$ we are dealing with a regression task. Common types of SL are classification, where the input is mapped to one (or more) out of several possible classes, and regression, where the input is mapped to a numerical value. SL can be considered the bedrock of ML, as SL techniques are commonly used within more complex systems; for example, many RL algorithms use SL-based subroutines within them. Indeed, in order for it to work SL requires data to be labelled, that is, for each input data point x we need to know its corresponding label y.

In constrast with SL, Unsupervised Learning involves methods that seek to extract inherent structures or patterns without explicit supervision, that is, in the absence of labeled data; clustering and dimensionality reduction are common applications of UL. The advantage of Unsupervised Learning lies in its capacity to unveil hidden relationships within data without guidance; nevertheless, its limitations arise when interpreting the discovered structures, often leaving room for subjectivity and requiring careful validation to extract meaningful insights.

Finally, Reinforcement Learning involves agents that learn how to behave in complex environments in a trial-and-error fashion, that is, by interacting with the environment and receiving feedbacks in the form of rewards or penalties. It draws inspiration from behavioral psychology and neuroscience, and it has been shown that the dopaminergic system in the human brain can be described using TD-learning, one of the main RL methods. RL shines in scenarios with sequential decision-making, as seen in robotics and game playing.

2.2.1 Reinforcement Learning

Reinforcement Learning (RL) comprises two main entities: the agent, which is the (artificial) system tasked with learning and decision-making, and the environment, which is the (real or virtual) world where the agent lives and with which it interacts; at each interaction step t, the agent receives an observation o_t of the current state s_t of the environment and it has to decide how to act, i.e. choose an action a_t, on the basis of such observation. The environment also emits another signal, called the **reward** signal r_t, a number that is perceived by the agent and that describes how good (or bad) the current state is. The goal of the agent is then to maximize its **expected return** (i.e. **cumulative reward**), which is the sum of rewards over the axis of time [63]. This goal is informally stated with the **reward hypothesis**: "*That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)*" [63, 57].

Formally, the above framework is usually represented via Markov Decision Processes (MDPs). A MDP is usually defined as a 5-tuple (S, A, R, P, ρ_0), where:

- $\mathcal{S}$ is the set of states;
- $\mathcal{A}$ is the set of actions;
- $\mathcal{R} \subset \mathbb{R}$ is the set of rewards;
- $\mathcal{P} : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ is the probability transition function or dynamics function;
- $\rho_0(s)$ is the starting state distribution, i.e. the probability that state s is the first state encountered by the agent.

There are many different flavours of MDPs depending on the nature of these individual components, indeed leading to different methods and strategies to solve them. First, if the observation o_t is a complete and exact representation of the environment's state s_t, then the environment is said to be *fully observable*; if o_t contains only partial information on s_t, then the environment is *partially observable*, leading to a **Partially Observable MDP** (**POMDP**). Furthermore, the action space $\mathcal{A}$ may be a *discrete action space* where only a finite (albeit possibly very large) number of moves are available to the agent; examples of such environments are board games like Chess and Go and the Atari57 benchmark. Differently, in a *continuous action space* the actions are real-valued vectors; for instance, most applications of RL in Robotics and control problems are usually modelled with continuous actions. Finally, the MDP may represent an *episodic task*, where the sequence of agent-environment interactions can be naturally split into different *episodes* (such as a game of chess), or it may represent a *continuing task* in which there is no concept of individual episodes (such as a robotic arm learning to operate) [63].

We now introduce some important definitions that are central to RL theory and methods. We start by introducing the core concept of **policy** $\pi(a|s)$, which is a mapping from states to a probability distribution over the action space that dictates how the agent acts; that is, $\pi(a|s)$ represents the probability of choosing action a when the agent finds itself in state s. From a practical point of view, *parametrized policies* are usually employed, that is, policies whose output depends on a set of parameters θ (such as the weights in a neural network) that can be optimized via some optimization method to obtain the desired behaviour; to emphasize this connection, policies are usually denoted as $\pi^{\theta}(a|s)$ (or $\pi(a|s,\theta)$). When a given policy is executed for a certain number of steps (say t) in an environment, a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, .., s_{t-1}, a_{t-1}, r_t)$ is obtained as a result, which is a sequence of states, actions and (instantaneous) rewards.

As earlier discussed, the goal of the agent in a MDP is to maximize the cumulative reward over a trajectory; this cumulative reward or **return** is usually denoted by $R(\tau)$. There are two main definitions of $R(\tau)$, depending on the nature of the MDP:

- **Finite-horizon undiscounted return** is the sum of rewards over a fixed number of steps:
$$R(\tau) = \sum_{t=0}^{T} r_t$$
- **Infinite-horizon discounted return** is the sum of all rewards ever received by the agent, discounted by a factor $\gamma \in (0, 1]$:
$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

Note that with $\gamma = 1$ there is no discounting applied, and also note that discount is essentially required in the infinite-horizon case, as otherwise the return may diverge. Finite-horizon undiscounted returns are well suited for episodic tasks, as T can naturally be the length of each episode, whereas infinite-horizon discounted returns are better used for continuing tasks.

Finally, we can introduce the performance measure $J_R(\pi)$, called **expected return**:

$$J_R^\pi = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)] \tag{2.13}$$

where $\tau \sim \pi$ represents a trajectory sampled by executing policy π in the environment; in other words, J_R^π is the expected value of the return of trajectories that result from the execution of policy π. The optimization problem associated with a MDP can then be formulated as the maximization of the performance measure, that is:

$$\pi^* = \arg\max_{\pi} \; J_R^\pi \tag{2.14}$$

where π^* is the optimal policy.

Before moving on to describing some fundamental RL algorithms, we first need to define the *value functions* - functions that estimate how good it is for the agent to be in a certain state; these definitions are based on [63]. The first one is the **state-value function** $V_\pi(s)$, which is the expected return obtained by starting from state s and following policy π:

$$V_\pi(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|S_t = s]$$

Similarly, we can define the **action-value function** (or *state-action value function*) $Q_\pi(s, a)$, which is the expected return obtained by starting from state s, executing action a and then following the policy π:

$$Q_\pi(s, a) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|S_t = s, A_t = a]$$

Finally, we can define the **advantage function**, which simply tells the *relative* *advantage* of an action, i.e. how much it is better than the other actions on average:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

Note that in the following sections, to indicate (learned) approximations of the above functions (e.g. via a neural network), we will use the hat notation (e.g. $\hat{V}_\pi(s)$ is the learned approximation of the state-value function of policy π).

VPG

One of the main classes of RL algorithms is that of **policy gradients (PG) algorithms**, where a policy is directly learned from environment interactions. Given a policy π parameterized by θ and a scalar performance measure function $J_R^{\pi_\theta}$, the core idea of PG methods is to update θ by performing stochastic gradient ascent on J w.r.t. θ:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J_R^{\pi_\theta} \tag{2.15}$$

However, in general it's difficult (if possible at all) to compute $\nabla_\theta J_R^{\pi_\theta}$, as it requires knowledge on the (gradient of the) distribution of states induced by the current policy π_θ, which is typically unknown [63]. Luckily, the **policy gradient theorem** establishes a useful proportionality relationship between the true gradient $\nabla_\theta J_R^{\pi_\theta}$ and an analytical expression that does not involve the derivative of the distribution of states [63]:

$$\begin{aligned} \nabla_\theta J_R^{\pi_\theta} &= \nabla_\theta \sum_{s\in S} \mu^{\pi_\theta}(s) \sum_{a\in A} Q_\pi(s,a)\pi_\theta(a|s) \\ &\propto \sum_{s\in S} \mu^{\pi_\theta}(s) \sum_{a\in A} Q_\pi(s,a)\nabla_\theta \pi_\theta(a|s) \end{aligned} \tag{2.16}$$

that is, we can perform stochastic gradient ascent on J by sampling from a sample gradient whose expectation is proportional to the actual gradient $\nabla_\theta J_R^{\pi_\theta}$.

Following the PG theorem, the **Vanilla Policy Gradient** (**VPG**) algorithm (also know as **REINFORCE** [68]) is the simplest instantiation of a PG algorithm; it works by defining the sample gradient as:

$$\nabla_\theta J_R^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}] \tag{2.17}$$

resulting in the following update rule:

$$\theta_{t+1} = \theta_t + \alpha R(\tau)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{2.18}$$

This can be further improved by including a *baseline* $b(s)$, an arbitrary function used to reduce the variance of the sampled return; the update rule becomes:

$$\theta_{t+1} = \theta_t + \alpha(R(\tau) - b(s))\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{2.19}$$

Actor Critic

Actor Critic (**AC**) algorithms can be seen as the combination of Temporal Difference learning [62] and PG methods, as AC methods learn approximations to both policy (i.e. actor) and value (i.e. critic) functions. In particular, the value function (state-value or action-value function depending on the algorithm) is learned via TD learning (i.e. bootstrapping), which is then used to inform the updates of the actor. For instance, one-step AC replaces the full return of REINFORCE with the one-step return and making used of the learned value function $\hat{V}_\pi$ as follows [63]:

$$\theta_{t+1} = \theta_t + \alpha(r_{t+1} + \gamma\hat{V}_\pi(s_{t+1}) - \hat{V}_\pi(s_t))\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{2.20}$$

PPO

A very simple idea to improve the training stability of PG-based algorithm is to limit how much the policy can change after a single update; that is, we want to avoid policy updates that would change the parameters too much, as it might lead to performance collapse. **Trust Region Policy Optimization** (**TRPO**) [54] does this by enforcing a KL divergence constraint on the policy update size at each step. While TRPO offers theoretical guarantees of monotonic improvement with each update, it does so via a complex second-order optimization problem, making it a complicated and computationally expensive algorithm. **Proximal Policy Optimization** (**PPO**) [56] is an AC method which addresses the same issue as TRPO (i.e. limiting how far the policy can move in parameter space) via a simpler first-order, and it has been shown empirically to perform at least as well as TRPO. Formally, the (constrained) problem that TRPO as well as PPO solve can be formalized as:

$$\begin{aligned} &\max_{\theta} \; J_R^{\pi_\theta} \\ &s.t. \; D(\pi_{\theta_t}, \pi_{\theta_{t+1}}) \leq \delta \end{aligned} \tag{2.21}$$

where D is some distance function (e.g. KL divergence). Now, let $r_t(\theta)$ be the probability ratio $r_t(\theta) = \frac{\pi_{\theta_t}(a_t|s_t)}{\pi_{\theta_{t-1}}(a_t|s_t)}$ between the old and new (updated) policy. PPO solves the above optimization problem by using the following surrogate objective:

$$\theta_{t+1} = \theta_t + \alpha \left(\min \left(r_t(\theta_t)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right) \tag{2.22}$$

where ϵ is an hyperparameter which approximately fixes the maximum allowable distance in parameter space between the old and new policy. $\hat{A}_t$ is an estimator of the advantage function at timestep t; to this end, the most commonly used estimator is the *Generalized Advantage Estimation* (GAE) [55].

2.2.2 Constrained Reinforcement Learning

Reinforcement Learning (RL) has shown remarkable success in training agents to make decisions in dynamic environments. However, deploying RL algorithms in real-world scenarios raises concerns about the safety of the learned policies. In this section, we delve into the concept of **Constrained Reinforcement Learning** (also know as **Safe Reinforcement Learning**), a crucial area that addresses the inherent risks associated with RL applications.

The motivation behind integrating safety measures into RL stems from the potential consequences of deploying unverified policies in environments where failures can have severe repercussions. Traditional RL algorithms lack explicit constraints to ensure that the learnt policies adhere to predefined safety specifications. Safe Reinforcement Learning seeks to mitigate these concerns by incorporating mechanisms that guarantee the satisfaction of safety constraints during the learning process.

Constrained Markov Decision Processes

To formalize safety in the RL context, we turn to the framework of **Constrained Markov Decision Processes** (**CMDP**s) [3]. Essentially, CMDPs extend the classical MDP formulation by introducing safety constraints via a set of cost functions and associated cost limits. A CMDP is formally defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{C}, \mathcal{D})$, where:

- $\mathcal{S}$ is the set of states;
- $\mathcal{A}$ is the set of actions;
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ is the probability transition function;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function;
- $\gamma \in (0, 1]$ is the discount factor;
- $\mathcal{C} = \{\mathcal{C}_i\}_{i=0}^{n}$, $\mathcal{C}_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the set of cost functions;

- $\mathcal{D} = \{d_i\}_{i=0}^{n},\ d_i \in \mathbb{R}$ is the set of cost limits, where each d_i is associated with cost function $\mathcal{C}_i$.

The optimization problem associated with a CDMP can then be formulated as:

$$\begin{aligned} &\max_{\theta}\ J_R^{\pi_\theta} \\ &s.t.\ J_{C_i}^{\pi_\theta} \leq d_i \end{aligned} \tag{2.23}$$

While CMDPs offer a formalism for incorporating safety constraints, several challenges persist in achieving safe RL: indeed, balancing the trade-off between learning optimal policies and satisfying safety constraints introduces additional complexity to the RL problem. The challenge lies in designing algorithms that navigate this trade-off effectively, ensuring the agent learns policies that not only maximize cumulative rewards but also adhere to safety specifications throughout the learning process.

2.3 Learning with optimization-driven costs

In the last couple years, interest has been increasing on the integration of ML techniques and CO methods, which led to the birth of many different subfields pursuing different approaches to tackle complex problem using both ML and CO. This is motivated by the fact that CO methods excel when *explicit knowledge* is available (e.g. cost function or constraints), as this can be used to build a mathematical model of the problem; on the other hand, ML methods can make use of *implicit knowledge* (e.g. historical data or simulators) [58].

Given these premises, many different methods and techniques have been developed, possibly addressing the same problem in quite different ways; for instance, learning algorithms may be used to help or substitute entirely the solving process of a CO solver, the modelling process, or both in an end-to-end fashion. A good survey on this wide set of techniques is provided by [19].

Given the availability of so many different methods, we focus on the method that most closely resembles the UNIFY framework [58] that is used as the basis of this thesis, namely **Decision Focused Learning** (**DFL**). First proposed in [20] and then later refined in [67], DFL is a framework that merges prediction and optimization into a cohesive end-to-end system, aiming at enhancing decision-making in combinatorial optimization problems; hence, it fundamentally differs from conventional methods, in which ML models are trained separately and independently from the optimization stage, by integrating learning (prediction) and optimization into a single, unified process. The primary challenge addressed is the misalignment of the loss function used in model training with the ultimate goal of optimizing decisions.

As an example, we consider the the Smart "Predict, then Optimize" (SPO) [22] framework, one of the main DFL approaches. The key component of this approach is the SPO loss, which measures the difference between the cost of the decision induced by the predicted parameters and the optimal cost under the true parameters; that is, it quantifies the decision error resulting from a particular prediction. The SPO framework recognizes that the same prediction error can lead to different decision errors depending on how the prediction affects the optimization outcome; therefore, the quality of a prediction is assessed based on its impact on the optimization decision, not just the accuracy of the prediction itself. [22]

Empirical results demonstrate the superiority of DFL over traditional two-stage approaches, where a predictive model is first trained for accuracy and then its predictions are used for the optimization phase: by directly aligning the model training objective with the goal of making optimal decisions, as opposed to optimizing for predictive accuracy alone, DFL allows the model to learn to prioritize predictions crucial for decision-making, even if it requires sacrificing overall prediction accuracy on irrelevant aspects [67, 22].

2.4 Constraints taxonomy

As described earlier, a CMDP extends the standard MDP definition by adding a set of cost functions $C = \{C_i\}_{i=0}^n$ and modifying the objective into finding a policy wich maximizes expected return while also being feasible, i.e. satisfying the constraints induced by C. Several classes of constraints exists: for instance, **instantaneous constraints** are constraints that the actions selected by the policy must satisfy at each time step [39], such as avoiding to reach certain states or to select some particular actions. Several approaches exist to deal with such constraints, such as applying a Lagrangian relaxation [8], analytically solving an action correction by adding a so-called Safety Layer to the policy [14] or even adopting a human-in-the-loop training and testing regime to guarantee zero constraint violation, as other methods only asymptotically guarantee constraint satisfaction [53]. Differently from instantaneous constraints, the class of constraints that we will focus on is that of **cumulative constraints**, i.e. constraints that apply to multiple time steps. Generally speaking, there are two main kinds of cumulative constraints: **expected constraints** involve an expectation of a cost function, whereas **probabilistic constraints** are based on the probability of a cost function. Expected constraints include discounted constraints and mean valued constraints [3, 39]. **Discounted constraints** are of the form:

$$J_{C_i}^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\sum_{t=0}^{\infty} \gamma^t C_i(s_t, a_t, s_{t+1})] \leq \epsilon_i \tag{2.24}$$

while **mean valued constraints** are of the form:

$$J_{C_i}^{\pi_\theta} = \frac{1}{T} \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\sum_{t=0}^{T-1} C_i(s_t, a_t, s_{t+1})] \leq \epsilon_i \tag{2.25}$$

where $\tau = (s_0, a_0, s_1, a_1, ...)$ is a trajectory induced by the policy π_θ, T is the total number of steps in the trajectory and ϵ_i is the cost limit (bound) associated with cost function C_i. For the sake of simplicity and without loss of generality, from now on we will only consider mean valued constraints for the expected constraints class, as algorithms that can deal with one type of constraint usually can work with the other type as well.

Probabilistic constraints instead assume the following form:

$$J_{C_i}^{\pi_\theta} = P(\sum_t C_i(s_t, a_t, s_{t+1}) \geq \eta) \leq \epsilon_i \tag{2.26}$$

where η is the cumulative cost limit for each trajectory and $\epsilon_i \in (0, 1)$ is the probability limit [23, 39].

Chapter 3

Methods

This chapter is the core of this thesis, as we describe the extension to the UNIFY framework that is being proposed to handle cumulative (i.e. long-term) constraints. This chapter is structured as folllows: we start by providing a brief introduction to the UNIFY framework in Section 3.1, which is then reformulated to accommodate cumulative constraints in Section 3.2; we then describe in Section 3.3 the RL algorithms that we employ to learn a constraint-satisfying policy for the offline problem, PPOLag and CPPO.

3.1 The UNIFY framework

The UNIFY framework, proposed in [58], considers the problem of learning a *constrained policy* for complex decision-making problems. Let $x \in X$ be the state vector (the observable information) and let $z \in C(x)$ represent the decision, which must lie in the feasible set $C(x)$. A (parametrized) constrained policy is a function: $\pi^\theta(x) \mapsto z \in C(x)$, that is, a function of the observable information mapping into feasible decisions. Then, the goal is to find parameters θ that minimize an objective function on a target distribution; the optimization problem can be formulated as:

$$\arg\min_{\theta \in \Theta} \mathop{\mathbb{E}}_{(x,x_+)\sim p} [f(x, x_+, \pi^\theta(x))] \tag{3.1}$$

where x is the observation, x_+ is the realization of the uncertainty, $f(x, x_+, z)$ is the objective function and p the target distribution. The issue here is that the above problem is generally pretty hard to solve; thus, the authors propose a clever decomposition of the policy in two separated components, namely a ML model and a CO problem. The policy is then reformulated as:

$$\begin{aligned} \pi^\theta(x) &= g(x, h(x,\theta)) \\ g(x,y) &= \operatorname*{arg\,min}_{z \in \tilde{C}(x,y)} \tilde{f}(x,y,z) \end{aligned} \tag{3.2}$$

where $h(x, \theta)$ is the ML model (e.g. a RL agent) and $g(x, y)$ is a function representing the optimal solution of the CO problem, parametrized on the ML model output y; this vector is called the *virtual parameter vector*. Based on this decomposition, the authors propose two different formulations of UNIFY, one for single-stage problems and the other for sequential decision-making problems. We only report the sequential formulation, as it is the one we'll actually use:

$$\begin{aligned} \operatorname*{arg\,min}_{\theta \in \Theta} \mathop{\mathbb{E}}_{\tau \sim \rho}[\sum_{k=1}^{eoh} \gamma^k f(x_{k+1}, x_k, z_k)] &= \operatorname*{arg\,max}_{\theta \in \Theta} \mathop{\mathbb{E}}_{\tau \sim \rho}[J_R^{\pi_\theta}] \\ z_k = g(x_k, y_k) &= \operatorname*{arg\,min}_{z_k \in \tilde{C}(x_k, y_k)} \tilde{f}(x_k, y_k, z_k) \\ y_k &= h(x_k, \theta) \end{aligned} \tag{3.3}$$

where $\tau = (x_0, z_0, x_1, z_1, ...)$ is a trajectory sampled according to probability distribution ρ. For further reference on the UNIFY framework, the readers are referred to [58].

3.2 UNIFY formulation with cumulative constraints

In order to instantiate the UNIFY framework to deal with cumulative constraints, recall the sequential definition of UNIFY presented in Equation 3.3.

It would be natural to try enforcing such constraints on z, as the bilevel optimization problem already involves constrained optimization on z via the virtual feasible set $\tilde{C}$; however, in multi-stage stochastic optimization the future outcomes of uncertainty is unknown. Incorporating long-term thinking capabilities required to handle cumulative constraints in traditional CO solvers could be extremely expensive and complex due to their myopic decision-making behaviour, while it would be much easier to enforce instantaneous constraints. Instead, we can formally view the problem as a CMDP, so that Equation 3.3 becomes:

$$\begin{aligned} \arg\min_{\theta \in \Theta} = \mathop{\mathbb{E}}_{\tau \sim \rho} [\sum_{k=1}^{eoh} & \gamma^k f(x_{k+1}, x_k, z_k)] \\ & z_k = g(x_k, y_k) = \arg\min_{z \in \tilde{C}(x_k, y_k)} \tilde{f}(x_k, y_k, z_k) \\ & y_k = h(x_k, \theta) \\ s.t. \quad & J^{\theta}_{C_i} \leq \epsilon_i \end{aligned} \tag{3.4}$$

We can employ Lagrangian relaxation to convert the offline constrained problem to an unconstrained one, an approach first introduced in [3]. Such Lagrangian approach has been chosen over a number of other methods (see e.g. [39]) both for its simplicity and strong theoretical foundations, as well as for being one of very few methods that can deal with both cumulative and probabilistic constraints.

We start by defining the Lagrangian function:

$$L(\theta, \lambda_i) = J^{\theta}_R - \sum_i \lambda_i (J^{\theta}_{C_i} - \epsilon_i) \tag{3.5}$$

where each λ_i is the Lagrange multiplier associated with each constraint and

$J_R^\theta = -\sum_{k=1}^{eoh} \gamma^k f(x_{k+1}, x_k, z_k)$. Now, we can rewrite 3.4 as:

$$\begin{aligned}
\arg\min_{\lambda_i \geq 0} \arg\min_{\theta \in \Theta} \mathop{\mathbb{E}}_{\tau \sim \rho}[-L(\theta, \lambda_i)] &= \mathop{\mathbb{E}}_{\tau \sim \rho}[\sum_{k=1}^{eoh} \gamma^k f(x_{k+1}, x_k, z_k) + \sum_i \lambda_i (J_{C_i}^\theta - \epsilon_i)] \\
z_k = g(x_k, y_k) &= \arg\min_{z \in \tilde{C}(x_k, y_k)} \tilde{f}(x_k, y_k, z_k) \\
y_k &= h(x_k, \theta)
\end{aligned} \tag{3.6}$$

3.3 Solving CMDPs

The Lagrangian method and the UNIFY extension described in previous sections can be employed to tackle CMDPs: this approach was first introduced in [3], where the constrained problem associated with a CMDP is reduced to an unconstrained problem via Lagrangian relaxation. Given a CMDP, its associated unconstrained problem can be formulated as:

$$\min_{\lambda \geq 0} \max_\theta \mathcal{L}(\theta, \lambda) = \min_{\lambda \geq 0} \max_\theta [J_R^{\pi_\theta} - \lambda J_C^{\pi_\theta}] \tag{3.7}$$

where $\mathcal{L}(\theta, \lambda)$ is the Lagrangian function and $\lambda = [\lambda_0, ..., \lambda_n]$, $\lambda_i \geq 0$ is the vector of Lagrange multipliers; this problem is solved by gradient ascent on θ and descent on λ. For the sake of simplicity and without loss of generality, we consider the case with a single Lagrangian multiplier (i.e. $\lambda = [\lambda_0]$), while in principle one might introduce a different multiplier for each constraint.

3.3.1 PPOLag

We consider PPO as the underlying RL agent along with the extended training objective given by the lagrangian relaxation, an approach first introduced in [50] in which the resulting agent is named **PPOLag**. In order to use PPO to solve the above unconstrained problem, we first need to update PPO's surrogate objective to take into account the cost function(s):

$$\begin{aligned} &\max_{\theta} \; [J_R^{\pi_\theta} - \lambda J_C^{\pi_\theta}] \\ &s.t. \; D(\pi_{\theta_t}, \pi_{\theta_{t+1}}) \leq \delta \end{aligned} \tag{3.8}$$

that is, we penalize the agent for violating constraints based on the current value of λ. Then we can update λ by solving:

$$\min_{\lambda \geq 0} \, [J_R^{\pi_\theta} - \lambda J_C^{\pi_\theta}] \tag{3.9}$$

From a practical point of view, the update rule for λ usually is the following, employing subgradient descent:

$$\lambda_{t+1} = \left(\lambda_t + \eta_\lambda \left(J_C^\pi - d\right)\right)_+ \tag{3.10}$$

where $(\cdot)_+$ is a projection on $\mathbb{R}^+$, in order to satisfy the nonnegativity constraint $\lambda \geq 0$. Equivalently, we can define a real-valued vector $\hat{\lambda}$ from which to obtain the actual Lagrangian multiplier λ as $\lambda = (\hat{\lambda})_+$, resulting in a similar update rule:

$$\hat{\lambda_{t+1}} = (\hat{\lambda_t})_+ + \eta_\lambda \left(J_C^\pi - d\right) \tag{3.11}$$

3.3.2 CPPO

Previous studies have shown that gradient-based Lagrangian methods for CRL do converge to optimal, feasible policies under mild assumptions [64]; in fact, it has been proved that the Constrained RL problem has zero duality gap, i.e. it features strong duality [47].

Still, these methods suffer from several drawbacks, such as slow convergence requiring many training iterations and wild oscillations of the cost during training, leading to intermediate policies that are either over-satisfying the constraints (i.e. obtaining a much lower cost that the one required) or just unsafe.

In [61], the authors interpret the overall Constrained RL problem as a dynamical system, in which the behaviour of classical Lagrangian method can be classified as *integral control*; they then consider the simplest mechanisms that could be integrated in this scheme, namely *proportional control* and *derivative control* [31, 61]. This results in a PID control update rule for RL agent solving CMDPs; this is then integrated on top of PPO, resulting in **Constraint-controller PPO** (**CPPO**). The usage of a PID controller to update the penalty coefficient is expected to improve performances by increasing the responsiveness of the agent w.r.t. variations in the cost, as the proportional component can dampen oscillations, whereas the derivative component can anticipate constraint violation and prevent cost overshoot [61].

3.3.3 Practical considerations

We conclude this chapter by highlighting some practical issues and considerations of using a RL agent as ML model within an hybrid learning and optimization framework such as UNIFY. Note that the following considerations apply to both PPOLag and CPPO.

Previous work have often used a single value function approximator (i.e. a single critic), which is trained in the usual supervised way but using $r + \lambda c$ as reward (i.e. target) [64, 47]; in other words, by using this combined reward the single critic employed is tasked with learning a state representation that is expressive both in terms of return-related characteristics as well as its safety-related aspects. In general, these features may be intertwined in a complex manner, and learning a single, good representation for both aspects may be a challenging task; moreover, λ may change rapidly (even more when using the PID-based controller), thus leading to large variation of the target and adding further training instability. Thus, we employ two separated function approximators, one for the usual RL return and the other for the safety cost;

this approach has also been used in more recent works (see e.g. [61, 70]).

Furthermore, we employ the re-scaled objective presented in [61] to train the actor network, as a large value of λ may indeed result in a large update for θ. The resulting objective is the following:

$$\arg\max_{\theta} J_R - \lambda J_C = \arg\max_{\theta} \frac{1}{1+\lambda}(J_R - \lambda J_C) \tag{3.12}$$

Finally, one main assumption of convergence proofs is the requirement that the Lagrangian update is carried out on a slower time-scale w.r.t. policy (and, eventually, value function) updates [64, 7, 47], meaning that these methods are actually solving the dual problem.

Chapter 4

Empirical evaluation

In this chapter, we instantiate the UNIFY framework in order to deal with long-term (i.e. cumulative) constraints. Note that the hybrid offline/online optimization method proposed in [17] and adopted in [58] assumes *exogenous uncertainty*; conversely, a cumulative constraint can be viewed as a constraint on a state variable subject to *endogenous uncertainty*, thus extending the applicability of the method. An extensive empirical evaluation has been carried out to determine strengths and weaknesses of the method.

This chapter is structured as follows: we start by describing the basic EMS problem as well as its variant equipped with long-term (i.e. expected) constraints that we will use as case study for the empirical evaluation in Section 4.1. Section 4.2 described the empirical setup and evaluation methodology adopted, while the remaining of the chapter focuses on empirical results: Section 4.3 compares the proposed method against a simple baseline; Section 4.4 discusses and analyzes the modelling process of the virtual parameter set; Section 4.5 compares different methods for the online component of the problem; finally, Section 4.6 involves trying to improve the general performances of the method.

4.1 Study case: Energy Management System

The problem we take into consideration is an **Energy Management System** (**EMS**) whose task is to allocate minimum cost power flows from various Distributed Energy Resources (DER) [2]. We follow and extend the hybrid offline/online approach first described in [17] and then also employed in [58], where a virtual cost parameter representing the storage subsystem is introduced, with the goal of overcoming the myopic behaviour of the online solver.

4.1.1 EMS with expected constraints

The variant of the EMS problem we consider involves expected constraints on the storage capacity. In particular, we consider the following constraint:

$$J_C^\theta = \frac{1}{T}\mathbb{E}[\sum_{t=0}^{T} cap(t)] > \rho \tag{4.1}$$

where $cap(t) \in [0,1]$ is the storage capacity at timestep t and $\rho \in [0,1]$; that is, we want the storage to be on average at least ρ percent full. Similarly, by focusing on unoccupied storage rather that occupied storage, we can invert the sign of the inequality, thereby simplifying its management:

$$J_C^\theta = \frac{1}{T}\mathbb{E}[\sum_{t=0}^{T} cap_{max} - cap(t)] \leq d \tag{4.2}$$

where cap_{max} is the maximum capacity of the storage and $d = 1 - \rho$ is the limit.

4.1.2 Online problem: LP model

The myopic online step follows a typical approach based on a LP model [2, 58]:

DV	**Power Flow**	**Description**	**Bounds**
x_0	Input to storage	Energy stored into the battery.	$[0, 200]$
x_1	Output from storage	Energy retrieved from the battery.	$[0, 200]$
x_2	Diesel power	Energy produced using the traditional generator.	$[0, 1200]$
x_3	Energy bought	Energy bought from the grid (i.e. market).	$[0, \infty]$
x_4	Energy sold	Energy sold to the grid (i.e. market).	$[0, 600]$

Table 4.1: Description of each (online) decision variable (i.e. power flows).

$$\begin{aligned} &\min_x \sum_{g \in G} c_g x_g \\ &\text{s.t. } \tilde{L} = \sum_{g \in G} x_g \\ &\quad 0 \leq \tau + \eta(x_0 - x_1) \leq \Gamma \\ &\quad \underline{x}_g \leq x_g \leq \overline{x}_g \end{aligned} \tag{4.3}$$

where x_g are the decision variables (i.e. power flows) between nodes with associated costs c_g. G represents the generic flows set (i.e. $G = \{0, .., 4\}$ for our purposes). All decision variables have finite domains, with $\underline{x}_g, \overline{x}_g$ being respectively the lower and upper bound of each variable. We assume that the (input) flow to the storage system is associated with index 0 and the (output) flow from the storage system is associated with index 1, so that τ, η, Γ are respectively the current charge, battery efficiency and upper bound; we set $\eta = 1$, $\Gamma = 1000$ in our experiments. Finally, $\tilde{L}$ represents the users' demand for the current step. A brief description of each decision variable is given in Table 4.1.

Please note that the model described in Equation 4.3 is the *virtual* model which makes use of the predicted virtual parameters, whereas the *true model* (i.e. the model we are optimizing in the EMS problem) only differs in the objective function, where c_0 and c_1 are not present.

4.1.3 Offline problem: RL agent

The offline problem relies on training a RL agent to predict the virtual costs associated with the storage systems, i.e. c_0, c_1. Note that for the sake of clarity we will later refer to c_0 as c_{virt}^{in} and to c_1 as c_{virt}^{out}, to emphasize the role of each virtual cost in the objective function. We employ the same environment variants introduced in [59] and also adopted by [58], focusing in particular on the sequential formulation. For the sake of completeness, we report here the reward function employed:

$$R(s_t, c_t, s_{t+1}) = -\sum_{g \in G \setminus \{0,1\}} c_g^t x_g^t \tag{4.4}$$

Indeed, due to the presence of a cumulative constraint in the problem formulation, there is an additional component to the environment, namely the *cost function*. The most straightforward cost function that can be employed is the following:

$$C(s_t, c_t, s_{t+1}) = \begin{cases} 0 & \text{if } t < T \\ \frac{1}{T}\sum_{t=0}^{T} cap_{max} - cap(t) & \text{if } t = T \end{cases} \tag{4.5}$$

This is a *sparse* cost function that has value equal to 0 at every timestep, except the last one where it is computed as the average free storage during the episode. We hypothesize (and later show empirically) that such a function does not fare well, as it introduces a *credit assignment problem*: essentially this means that the agent has to understand not only what actions optimize the function at each step, but also how the actions taken during the episode have affected the final outcome [63].

A possible solution may be to introduce a *dense* cost function such as the following:

$$C(s_t, c_t, s_{t+1}) = \frac{1}{T}(cap_{max} - cap(t)) \tag{4.6}$$

Note that the latter function is equivalent to the former only if the discount factor γ is equal to one, which is our case since we are considering an episodic environment [63].

4.2 Experimental setup and methodology

In this section we briefly discuss the experimental setup and evaluation methodology, describing how empirical results have been collected and providing some relevant implementation-level details.

The dataset employed is the same as [58] which in turn is based on a Public Dataset[1]: it contains forecasts for load demand and photovoltaic production of 10000 different days (i.e. *instances*), with aggregated profiles with a timestamp of 15 minutes (i.e. 96 profiles for each day). Furthermore, based on data from the Italian national energy market management corporation[2] (GME) and from the Italian Ministry of Economic Development[3], electricity demand hourly prices and diesel price have been derived; note that the latter is assumed to be constant for the time horizon (one day) as done in previous works [58, 2]. For more details on the data, see [58].

This dataset is split into training, validation and testing; the training instances may vary (as we experimented with training on a single instance like in previous works [58] as well as training on multiple instances), while the validation set (10 instances) and testing set (100 instances) are the same for all experiments (to avoid instance-induced bias when comparing different trials). To obtain the realizations of the uncertain variables (load demand and photovoltaic production), we loosely employ the same method of [58] (see Appendix B), although with a slight variation: for the training instances, once an episode is over (i.e. the agent has seen one particular realization of that instance) a new realization is computed, to allow for a better exploration of the (stochastic) state space. To ensure comparability across different experiments

[1] www.enwl.co.uk/lvns
[2] http://www.mercatoelettrico.org/En/Default.aspx
[3] http://dgsaie.mise.gov.it/

for each validation and testing instance one realization has been computed beforehand and has been kept the same for all experiments.

Moving on to the metrics, throughout the subsequent experiments we collect three main metrics:

- **Return** J_R^θ: the (average) return obtained by π, normalized according to the optimal cost obtained by a look-ahead oracle;
- **Cost** J_C^θ: the (average) cost obtained by π;
- **Constraint score**: a proxy metric, computed as the distance between the average cost and the cost limit, i.e. $|J_C^\theta - d|$; this metric has been introduced because, differently from the optimal cost (which depends on a particular cost limit), the optimal value of this score for a safe policy is ideally zero, with higher values indicating unsafer policies.

These metrics are collected once per training iteration on the training and validation set, and again at the end of the whole training on either the validation set (for hyperparameters tuning experiments) or the testing set (for the main experiments); this end-of-training evaluation is performed using both the *final* policy (i.e. the policy obtained at the end of the training) as well as using the *best* policy, which is the policy that minimized the constraint score on the validation set during training. Unless specified, in the following sections we consider scores obtained by the best policy as measured by the constraint score on the validation set. Note that this may be seen as a form of early stopping (although we still train for the predetermined number of iterations, to later compare the results of the two). Evaluation metrics on the validation and testing set are collected using both a *deterministic* policy (by directly using the mean outputted by the policy network) and the *stochastic* policy trained by PPO (note in this case several rollouts are performed to account for the stochasticity of the policy, whereas only a single rollout is required for the deterministic case).

On the implementation side, the library employed is TorchRL, a data-driven decision-making library for PyTorch [10, 46].

Finally, RL is known to be sensible to the choice of random seed [13, 26]; moreover, evaluation regimes of most of research in RL ignores the statistical uncertainty that results from evaluating a particular method only on a small, finite set of training runs [1]. Hence, we follow the evaluation regime described in [1], in which the authors suggest to employ Confidence Intervals (CIs) and in particular stratified bootstrap CIs to evaluate the uncertainty in aggregated performances. We consider a minimum of 5 different random seeds (for the most computationally-heavy set of experiments) up to 20 different seeds; differently, for hyperparameter tuning-related runs the random seed are not predetermined and different from testing seeds, so as to avoid overfitting as suggested in [21].

4.3 Unconstrained baseline: PPO vs PPOLag

We start by comparing two different RL algorithms for the offline component of the model on the EMS problem with expected constraints: an unconstrained baseline (PPO) that solves the problem as a MDP (i.e. it disregards the cost and only maximizes the return), and a constrained one (PPOLag) that maximizes the return while also learning a constraint-satisfying policy by solving a CMDP. We consider four different cost limits (from "easiest" to "hardest": 800, 500, 300, 200).

Figures 4.1, 4.2 and 4.3 show the results of this experiment. As expected, PPO does not care about the cost as it only optimizes the return, while PPOLag is capable of learning a safe policy. Figures 4.1 and 4.3 clearly show this: while PPOLag is capable of learning policies that attain the specified cost (or very close to it, considering the strictest limits i.e. 200 and 300), whereas PPO just always obtains maximum cost. On the other hand, Figure 4.2 shows that PPO can attain an higher overall return w.r.t. PPOLag. Moreover, considering

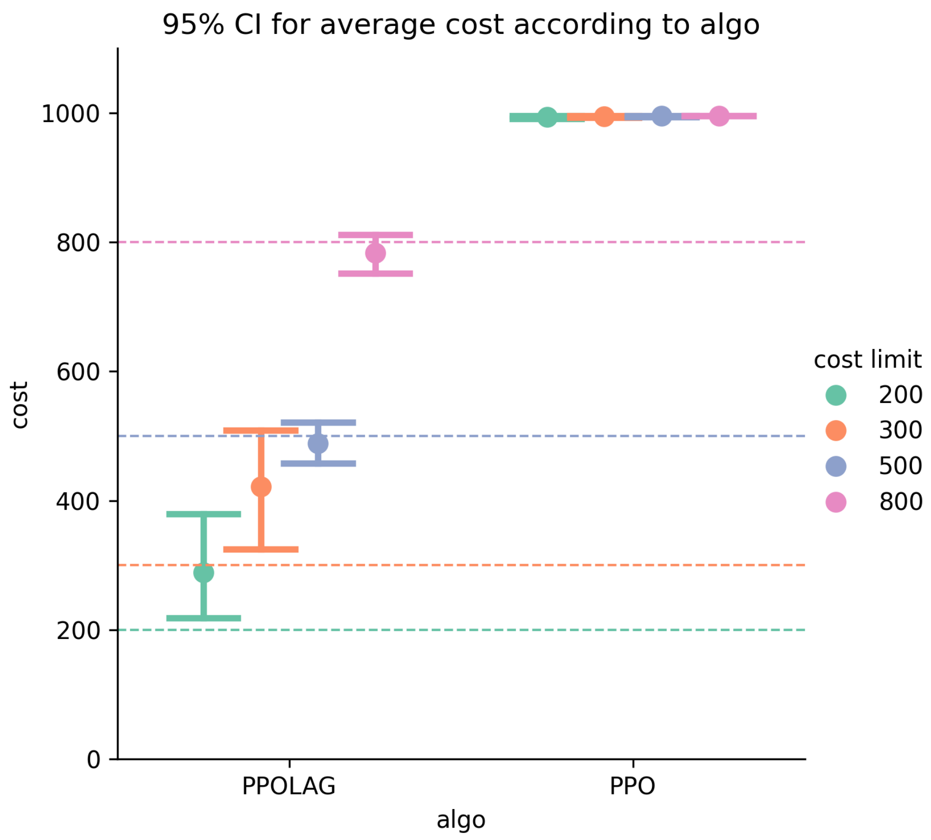

Figure 4.1: Average cost obtained by PPO and PPOLag on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit, with the horizontal bars indicating the specified costs. Stricter cost limits are more difficult to attain.

PPOLag we can notice a linear (direct proportionality) relationship between the specified cost and the obtained return: this is due to the fact that with stricter cost limits the agent must focus more on charging the battery system to satisfy the constraint and less on choosing optimal actions (in terms of return), thus leading to lower overall return.

4.4 On the choice of the virtual parameters

In [17, 58], the authors consider a single virtual cost as output of the RL policy, which we represent as c_{virt}^{in}, associated with the input flow to the storage

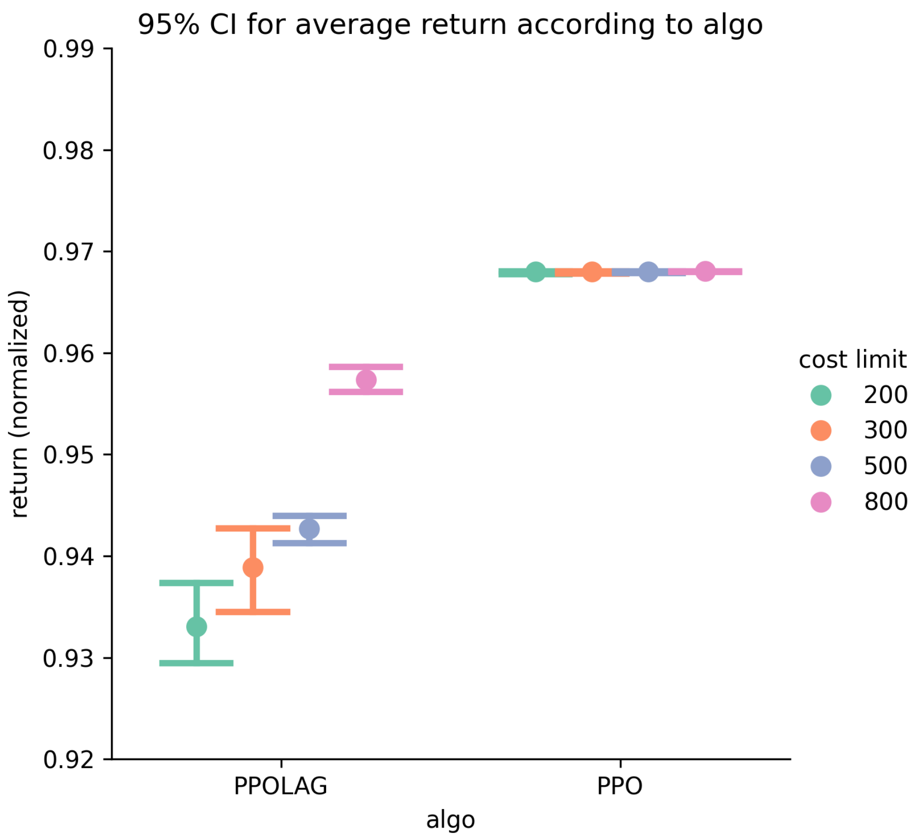

Figure 4.2: Average normalized return (the higher the better) obtained by PPO and PPOLag on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit.

system; we name this parametrization[4] CVirtIn. Following the online problem definition from Equation 4.3, c_{virt}^{in} is c_0, whereas c_1 is absent (i.e. it can be considered to be equal to 0 at every timestep). While such approach should work in principle, we noticed empirically that by employing such a simple parametrization the resulting policies were unsafe (i.e. incapable of satisfying constraints).

At first a quick round of hyperparameter tuning was carried out, to rule out the possibility that some particular hyperparameter setting was impeding convergence; this is because RL is known to be very sensitive to the choice of hyperparameters [29, 71], a phenomenon that is further aggravated by the

[4]The term parametrization may be to some extent considered improper; what we mean in this context by parametrization is the way the virtual parameter set is modelled and structured.

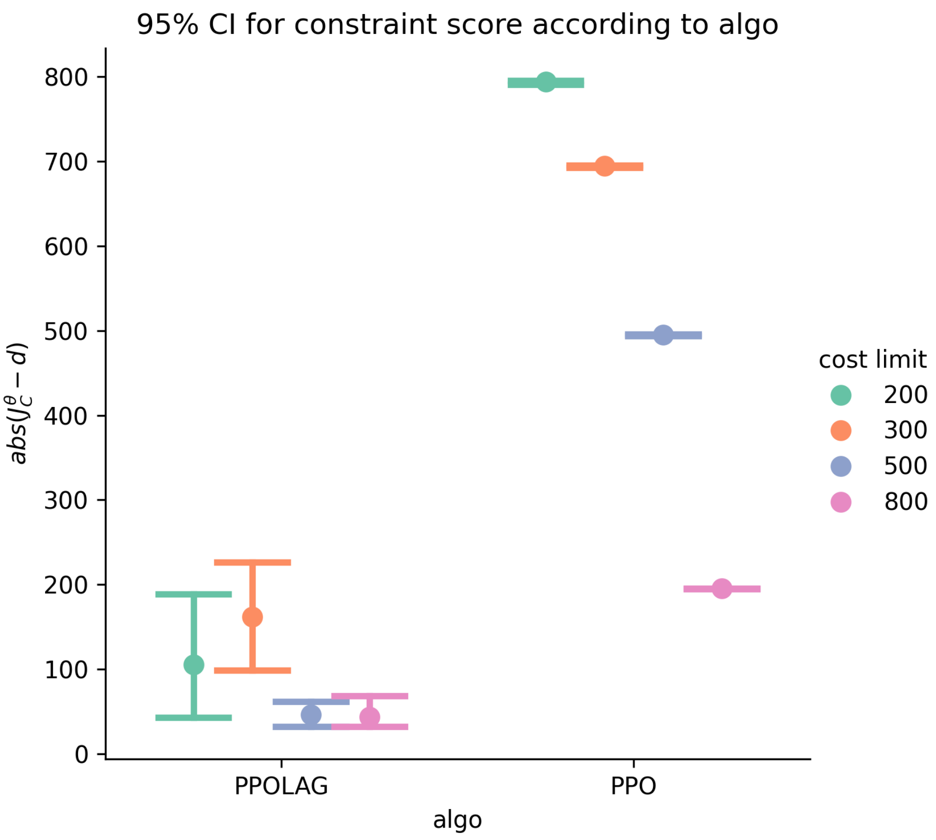

Figure 4.3: Average constraint score (the lower the better) obtained by PPO and PPOLag on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit.

usage of a Lagrangian method [39]. However, hyperparameter tuning didn't help in this context. To understand the actual issue, consider Figure 4.7, which shows the power flows resulting from executing a fully trained PPOLag agent on a testing instance, with the CVIRT-IN parametrization. As the agent has no control on the storage output flow (recall the agent's output i.e. c_{virt}^{in} only affects the online model's decisions concerning the storage input flow), the online model tends to myopically sell the available energy when there's a surplus and buying it when needed, rather than using the storage system to store excess energy so as to avoid buying it later at a possibly higher price.

Thus, it is clear that a different parametrization is required for the agent

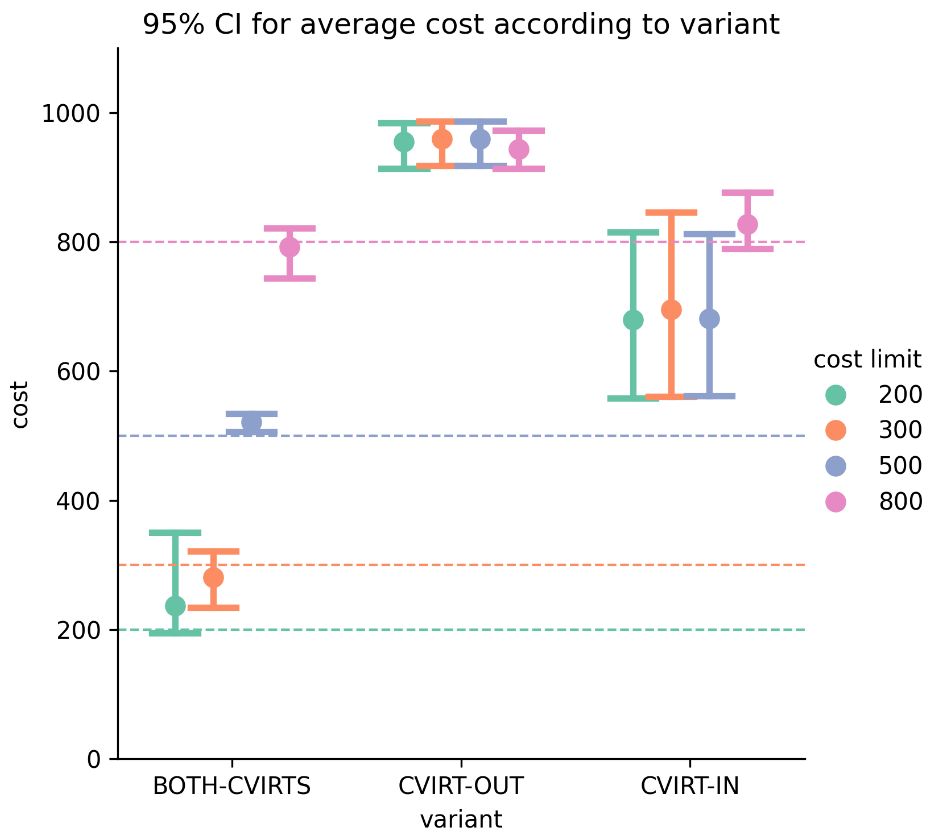

Figure 4.4: Average cost obtained by each parametrization on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit, with the horizontal bars indicating the specified costs. The BOTH-CVIRTS variant is the only one capable of producing policies that learn to satisfy the required constraint.

to properly influence the online model's decision concerning the storage system so as to satisfy the imposed constraint. This different parametrization essentially consists on a second virtual cost, represented as c_{virt}^{out} and associated with the (output) flow from the storage systems; we call this parametrization BOTH-CVIRTS. Figure 4.8 shows the power flows resulting from executing a fully trained PPOLag agent on a testing instance, employing the BOTH-CVIRTS parametrization. By comparing these power flows with the power flows of the CVIRT-IN parametrization (Figure 4.7), several differences can be noted: first, on average less energy is retrieved from the storage, leading to a much higher storage capacity that complies with the required cost limit (in this instance it

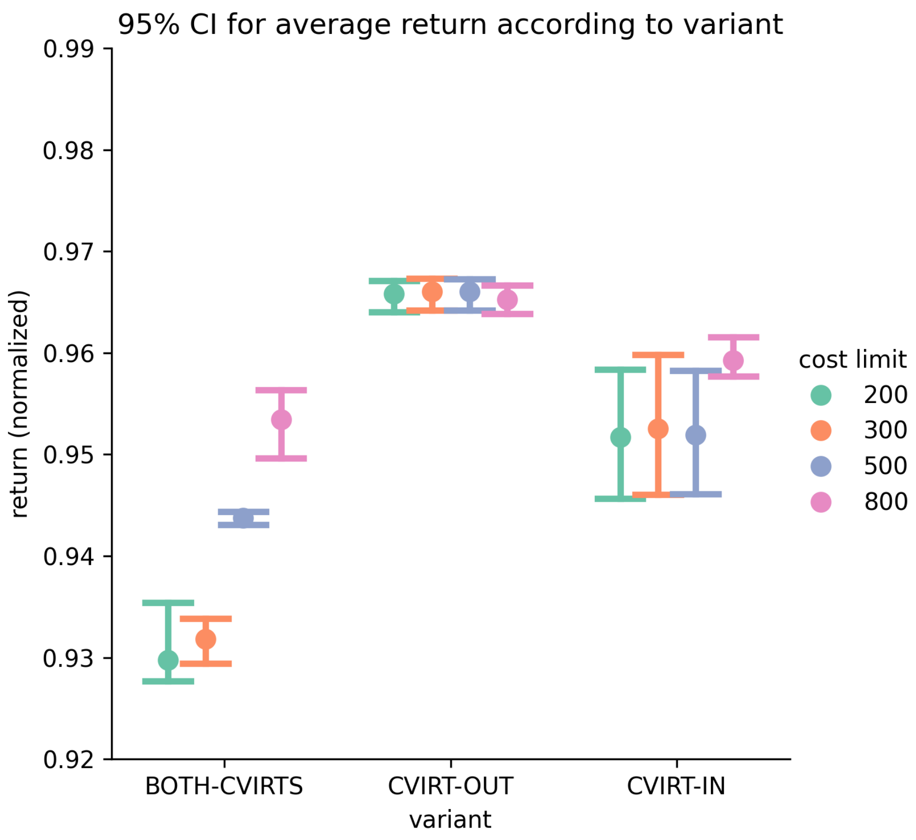

Figure 4.5: Average normalized return (the higher the better) obtained by each parametrization on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit.

is equal to 300, i.e. the battery should be 70% full on average). Second, less energy is sold to the grid, given it is now needed to keep the storage system charged. Third, less energy is bought from the grid, as the model can now actually use the energy that is stored in the battery. Finally, the generated power flows are almost identical, suggesting it is still needed to produce energy using traditional generators during peak hours.

As an ablation study, we also consider a trivial variant involving only c_{virt}^{out}, i.e. the agent outputs a virtual cost that is applied on the storage output flow only; we call this parametrization CVirt-In, and again we report the exemplar power flows of this parametrization in Figure 4.9. As expected, this is the worst performing parametrization out of the three considered: as it cannot influence in any way the online model's decision on the input energy to the storage, the

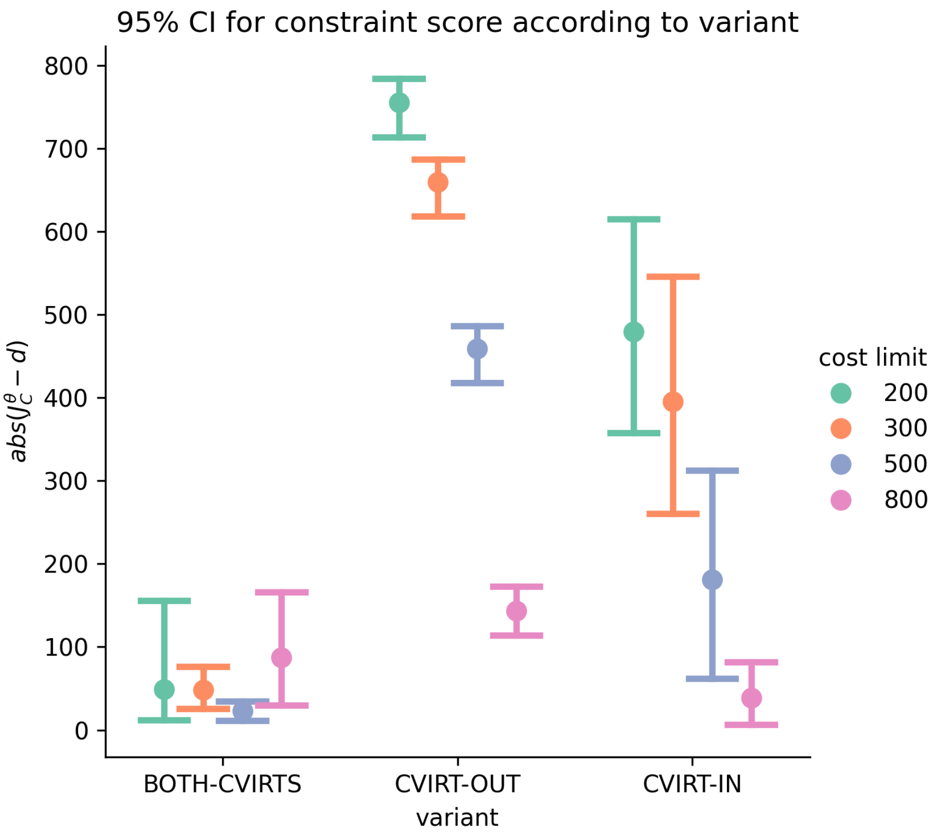

Figure 4.6: Average constraint score (the lower the better) obtained by each parametrization on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit.

agent almost immediately empties the storage without using it for the rest of the day.

Figures 4.4, 4.5 and 4.6 show the metrics collected on the test set for each variant. Clearly, the Both-Cvirts parametrization is the only one capable of consistently producing policies that can adhere to the specified requirements, as can be seen in Figures 4.4 and 4.6.

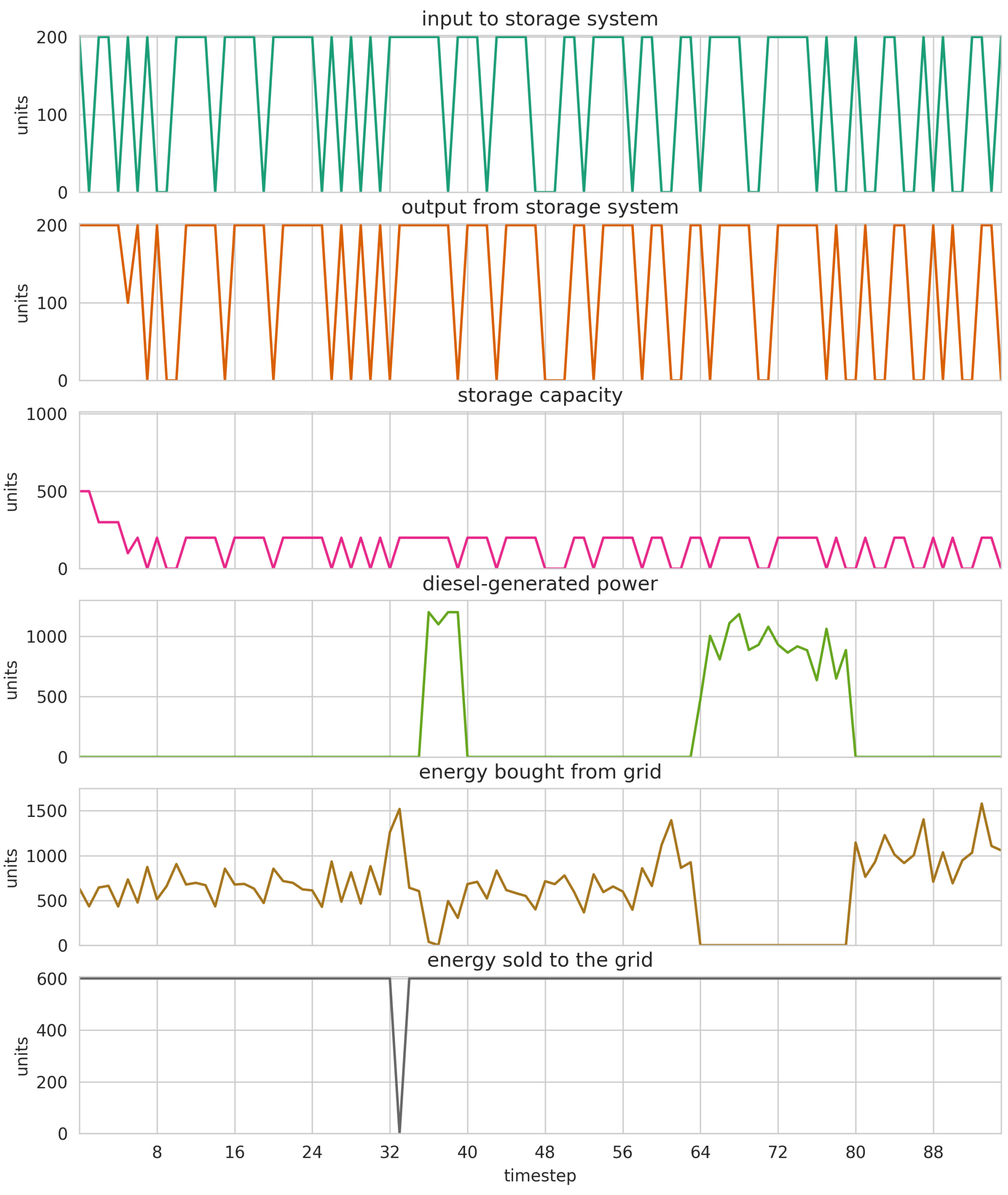

Figure 4.7: Power flows of a trained agent employing the CVirt-In parametrization (test instance, $d = 300$). As the agent has no control on the storage output, the online model tends to myopically sell the energy and then buying it when needed, rather than using the storage system to store excess energy to avoid buying it later at a possibly higher price.

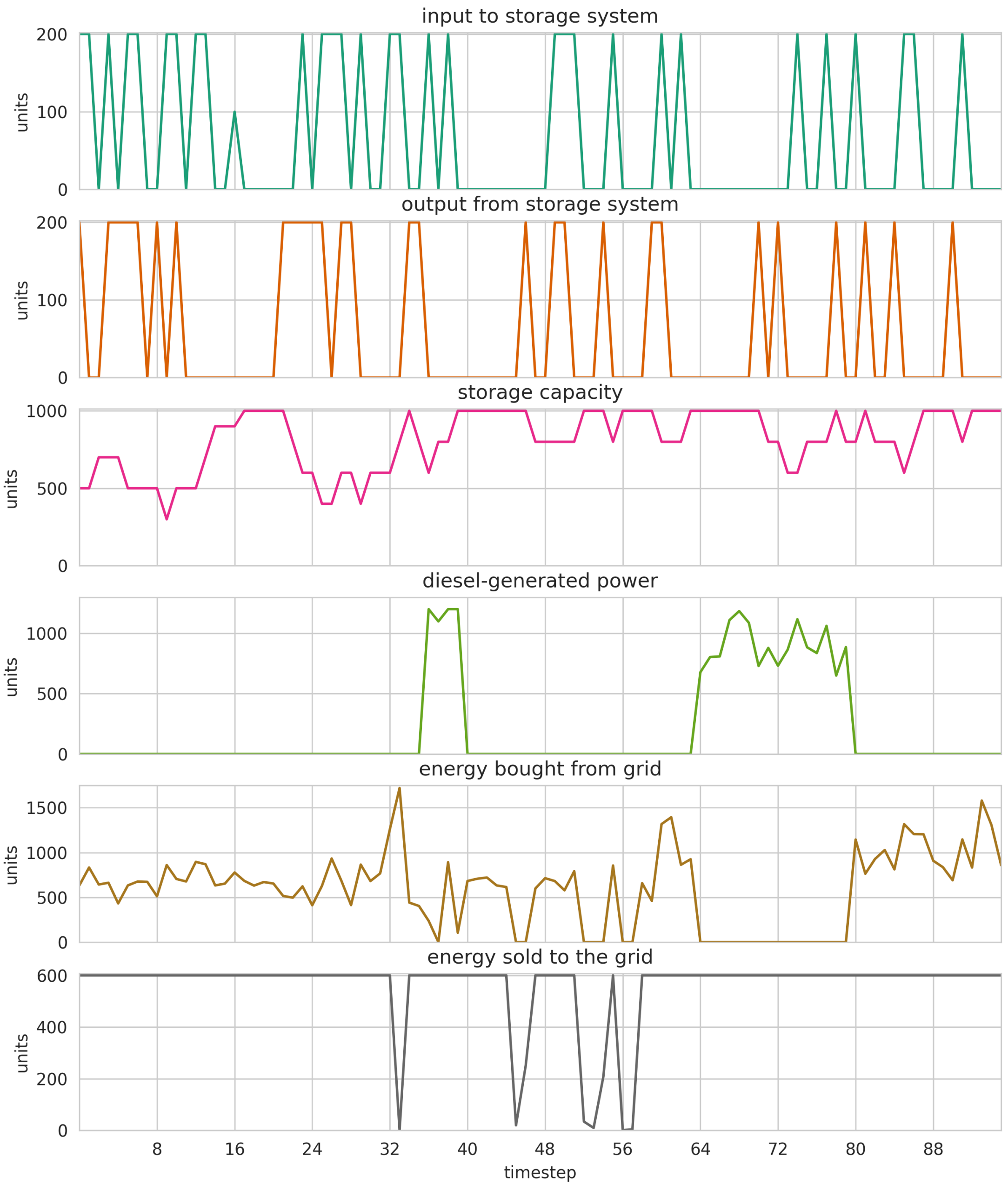

Figure 4.8: Power flows of a trained agent employing the Both-Cvirts parametrization (test instance, $d = 300$). Having given (indirect) control on the storage output to the RL agent, the online model prefers to keep the storage system at a higher capacity and only retrieve energy from it when necessary; this results both in less energy sold to the grid as well as less energy bought from the grid during peak hours.

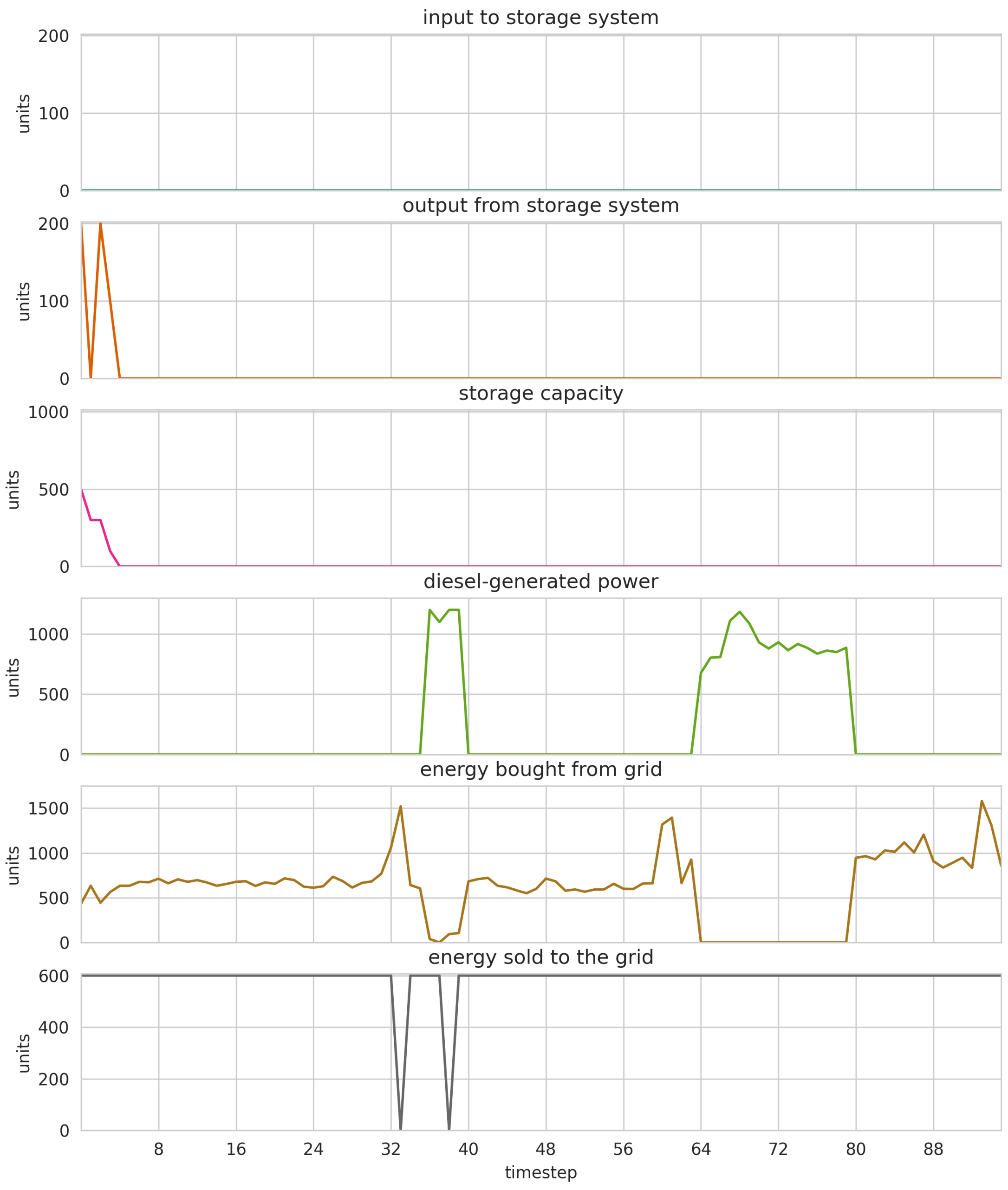

Figure 4.9: Power flows of a trained agent employing the CVirt-In parametrization (test instance, $d = 300$). As it cannot influence in any way the online model's decision on the input energy to the storage, the agent immediately empties the storage without using it for the rest of the day.

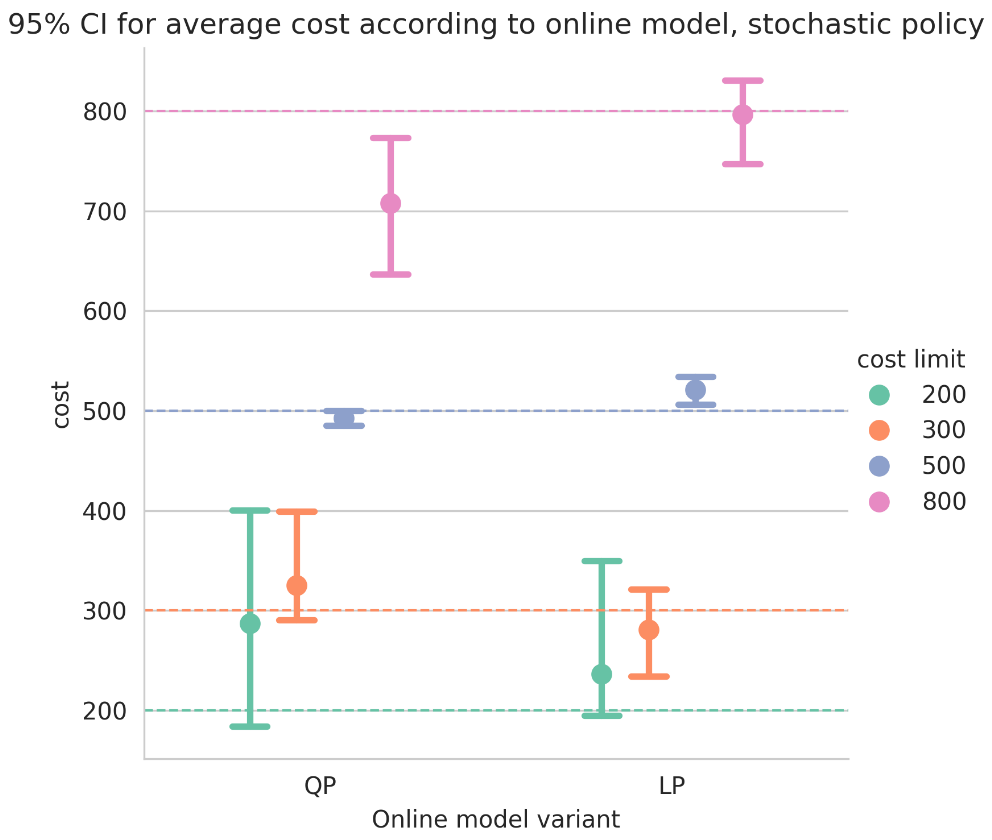

Figure 4.10: Average cost obtained by each online model variant on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit.

4.5 On the choice of the online optimization problem

The online optimization problem described in 4.3 and adopted in previous works [17, 58] is an LP model assumed to be in standard form (i.e. all $\underline{x}_g$ are equal to 0), as the nonnegativity constraint makes the feasible region of the optimization problem convex, which in turns makes the problem well-behaved and allowing for efficient algorithms to find the optimal solution. If negative values were allowed, the feasible region would be a non-convex set, leading to complications during the optimization phase and potentially multiple optimal solutions [65].

The issue with using a LP model in this particular context is that bound

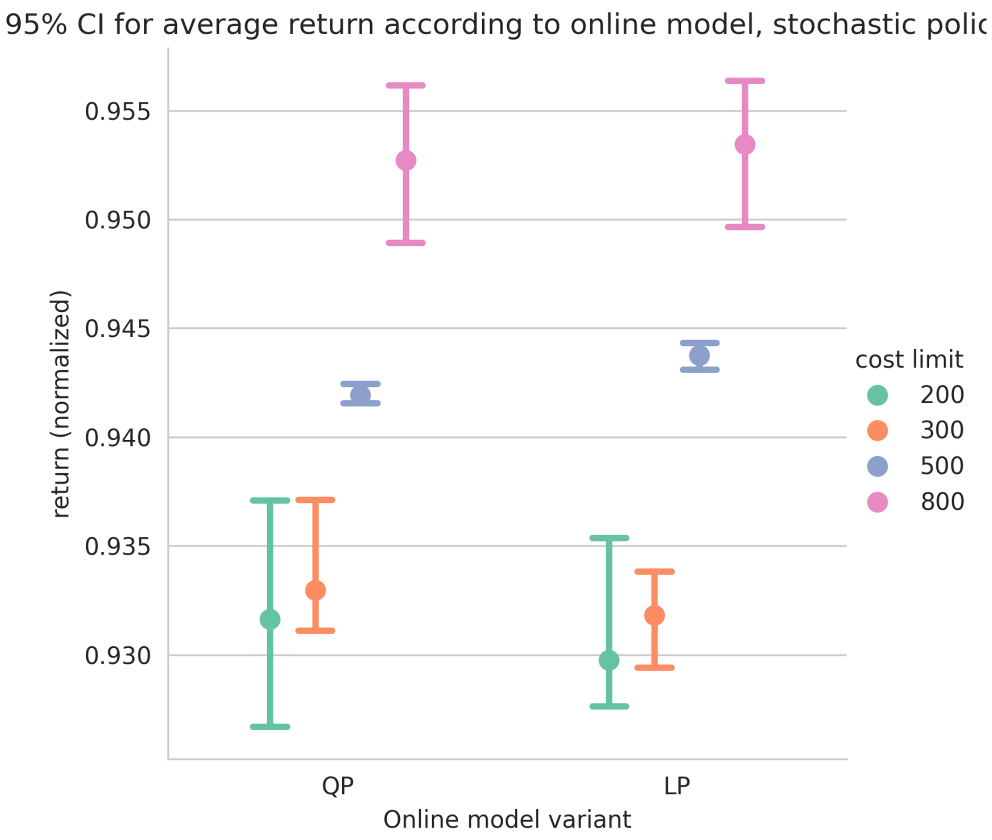

Figure 4.11: Average normalized return (the higher the better) obtained by each online model variant on 100 test instances, 95% CI. Runs are grouped according to the specified cost limit.

constraints involving the storage system become binding constraints, i.e., constraints that hold with equality in the optimal solution [9] due to the nature of the objective function. This basically means there are two possibilities: if $c^g_{virt} \geq 0$ then the corresponding flow x_g will be pushed to its lower bound (i.e. 0); conversely, if $c^g_{virt} < 0$ then the corresponding flow x_g will be pushed to its upper bound (unless of course the storage is full in the case of the input storage, or it is empty in the case of the output flow).In other words, the storage input and output flows essentially become binary decision variables.

Hence, we experimented with a slightly different online model (an offline RL agent) to see whether we could improve on this behaviour. The idea is to have as output of the RL policy the storage flows themselves; however the online model must retain the capability of altering them, as in general (especially at the beginning of training, when the policy is mostly random) there's a very high chance that it will output non-feasible flows. To this end, we consider a Quadratic Programming (QP) model, which can be seen as a simple variation of Equation 4.3 employing a quadratic objective function; in particular, the offline agent influences the storage flows not via costs within a linear objective function but by directly proposing the values of such flows. The online model can be formulated as follows:

$$\begin{aligned} \min_{x} \quad & \sum_{g \in G \setminus \{0,1\}} c_g x_g + (c_0 - x_0)^2 + (c_1 - x_1)^2 \\ \text{s.t. } & \tilde{L} = \sum_{g \in G} x_g \\ & 0 \leq \tau + \eta(x_0 - x_1) \leq \Gamma \\ & \underline{x}_g \leq x_g \leq \overline{x}_g \end{aligned} \tag{4.7}$$

where $c_0,\ c_1$ are given by the RL policy (for the sake of clarity we maintain the same notation as before, but note that here $c_0,\ c_1$ are actual flows and not virtual costs). Essentially, the online model optimizes the same objective function as before, with the only difference being the storage flows that are not a linear component but rather a quadratic term (a squared difference w.r.t. the RL policy output). Also note that the objective function is still convex, making the problem easier to solve as efficients algorithms are available for solving convex QP problems [43].

Figure 4.12 shows the power flows of a trained agent employing the QP model on a testing instance. As expected, the storage flows are no longer behaving as binary variables; however, by looking at Figures 4.10 and 4.11 that shows CIs for average cost and return, it can be noticed that there appears to be no evident benefit in using the QP model rather than the simpler LP one as the QP model performs the same or worse both in terms of cost and return. We also noticed that the training phase of the RL agent employing the quadratic online model to be much more unstable than when using the LP model, with some runs possibly diverging if some heuristic tricks had not been in place (e.g. stopping PPO's updates to the actor network when an estimator of the KL divergence between the old and new policy exceeds a certain threshold, which is a common implementation trick of PPO [27]). We also report that due to computational limitations the hyperparameter tuning was carried out only using the LP model, hence it is possible that performances of the QP model may improve after tuning. In any case, it is definitely easier for a RL agent to optimize the return in an environment with liner rather than quadratic dynamics, as from the point of view of the agent the online model is part of the nonstationarity of the environment.

4.6 Improving performances

We now turn our attention to a pivotal aspect that is indeed critical for practical deployment, that is, the optimization of the algorithmic performance. We first investigate the impact of different cost limits on the performance of the resulting agent; then we move on to describing the extensive hyperparameter tuning process that has been carried out.

4.6.1 Effects of varying the cost limit

We start by analyzing how different cost limits impact on the general performance of the method. To this end, we consider a (simulation of) an environment with 2 days episodes: in particular, the specified cost limit acts also as the value of the storage charge at the beginning of the episode, thus simulating a (trained) agent that on the day before left the storage system with the required charge level (i.e. the cost limit)[5].

We considered 8 different cost limits, ranging from 900 to 50, and for each cost limit we train agents on 10 different seeds. Figure 4.13 shows 95% CIs

[5]We had to resort to such a simulation, rather than actual 2 days episodes, due to computational limitations.

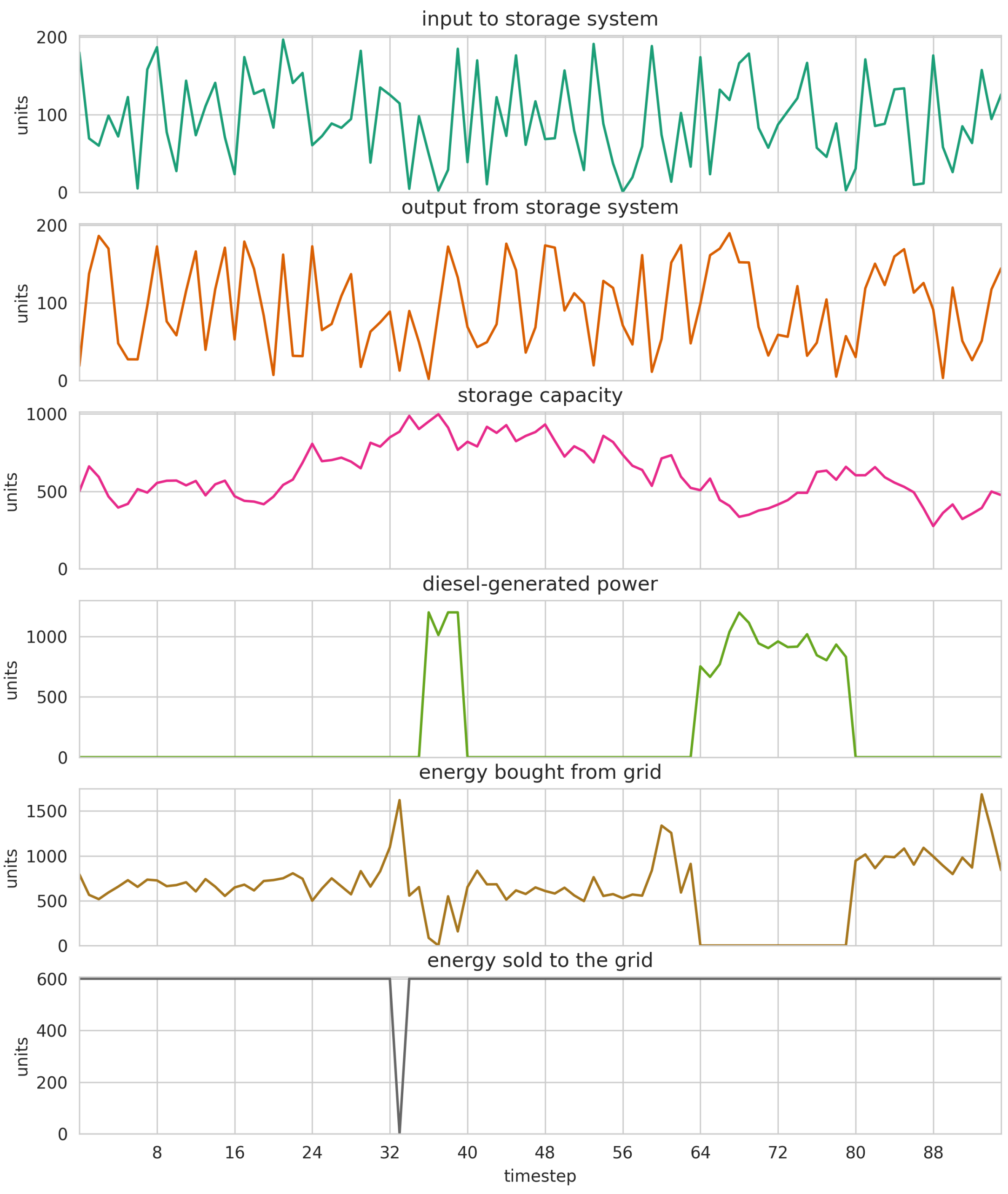

Figure 4.12: Power flows of a trained agent employing the Quadratic Programming online model (test instance, $d = 300$). With this approach, the storage flows no longer act as binary variables.

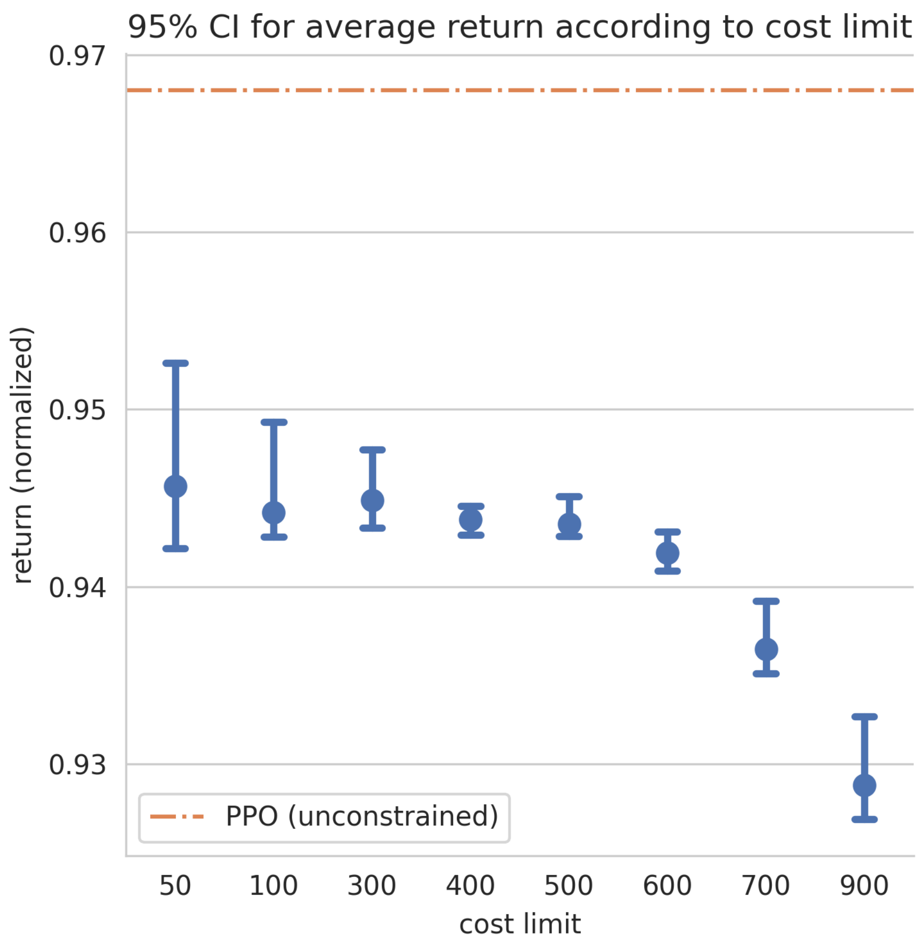

Figure 4.13: 2 days episode (simulation), average return (normalized) according to cost limit, 95% CI. Higher cost limits appear to performs worse than lower cost limits.

of the average return according to the specified cost limit, with the orange line representing the return of the unconstrained baseline (PPO). A clear trend can be noticed: higher cost limits (i.e. keeping the storage system at a lower charge, such as 10% or 30% full) performs worse than stricter cost limits, which requires the RL policy to keep the storage system at a higher level on average. Moreover it can be seen that the stricter the cost limit, the more uncertain the estimate is (i.e. wider CI); indeed stricter cost limits such as 50 or 100 may lead to additional difficulty in the Lagrangian-based optimization process, potentially adding instability. In any case, all the constrained agents appear to fall short of the unconstrained baseline (PPO): clearly the optimization problem that needs to be solved when training an unconstrained agent is simpler w.r.t. a constrained one, as the former needs to optimize for the return only whereas the latter is essentially optimizing two different objectives, and this struggle can be seen to be particularly intense with stricter (i.e. more difficult) cost limits. Thus, it cannot be ruled out that that either with further computational budget (i.e. longer training) or with additional fine-tuning a constrained agent may reach the performances of an unconstrained one, while retaining the capability of adhering to desired long-term constraints.

4.6.2 Hyperparameter Tuning

We now turn our focus to a common ML practice to improve the performance of algorithms, namely Hyperparameter Tuning (HT), which is an important issue of practical ML-based systems and is particularly problematic in RL and in Lagrangian-based gradient approaches [29, 71, 39].

Several rounds of HT had been carried out. The main HT process involved a random search, a well-know method for HT first proposed in [4], over a large set of hyperparameters and corresponding values. Table 4.2 shows the considered hyperparameters, each with the corresponding set of values employed in the random search. The search overall produced over 700 independent runs, which we will use for the analysis presented in the remainder of this section.

Given the very large set of hyperparameters considered, it is not reasonable if possible at all to fine-tune each single hyperparameter. A first step in our analysis involved determining the most important hyperparameters. One possible approach for doing so is by considering a regression task in which the input is a vector containing hyperparameters' values and the target is an evaluation metric (e.g. constraint score in our context). We can train a Random Forest-like regressor (we employed Extra-trees [24]) and use it to compute feature (i.e. hyperparameters) importance scores. In particular, we compute

Config name	**Description**	Values
environment.instances.train	Which instance(s) to use for training.	1, 10, 100 instances
actor.net_spec.depth	Depth of the actor MLP.	1, 2, 3
actor.net_spec.num_cells	Width of the actor MLP.	8, 16, 32, 64
agent.activation	Activation function used by the networks.	tanh, relu
agent.actor_lr	Learning rate for actor network.	0.02, 0.01, 0.0075, 0.005, 0.003
agent.actor_weight_decay	L2 norm applied to the actor network's weights.	0.0, 0.005, 0.01, 0.05
agent.critic_lr	Learning rate for critic network.	0.02, 0.01, 0.0075, 0.005, 0.003
agent.lag_lr	Learning rate for Lagrangian multiplier.	5.0, 2.0, 1.0, 0.5, 0.1
agent.lagrange.params.initial_value	Initial value of the Lagrangian multiplier.	50, 25, 15, 5, 1
agent.lagrange.positive_violation	Whether to train on positive costs only.	True, False
agent.loss_module.entropy_bonus	Whether to add an entropy term to the actor loss.	True, False
agent.loss_module_lag.cost_scale	Cost scaling factor.	1.0, 0.1, 0.01
agent.loss_module_lag.reward_scale	Reward scaling factor.	1.0, 0.1, 0.01
agent.loss_module_lag.target_kl	KL divergence threshold for early stopping actor updates.	0.5, 0.25, 0.1, 0.05, 0.025
agent.orthogonal_init	Whether to employ orthogonal initialization.	True, False
agent.schedule	Whether to schedule the LRs.	True, False
critic.net_spec.depth	Depth of the critic MLP.	1, 2, 3
critic.net_spec.num_cells	Width of the critic MLP.	8, 16, 32, 64
environment.params.cost_fn_type	Type of cost function.	dense, sparse
training.batch_size	Batch size used when training.	160, 320, 640, 1280
training.frames_per_batch	Amount of transitions collected at each rollout.	3200, 6400, 9600, 12800
training.num_envs	Number of parallel environments used during training.	8, 16, 32
training.num_epochs	PPO number of epochs for the inner optimization loop.	1, 5, 10

Table 4.2: Description of each tuned hyperparameter and considered values.

permutation feature importance: essentially, the computation process involves randomly shuffling the values of features (one at a time) and observe how much this degrades model performances [11]. Figure 4.14 shows such permutation importance scores for the 8 most important ones. Two hyperparameters appear of paramount importance, namely the (coefficient of) L2 regularization applied to the actor network and the type of cost function employed. Less important, yet relevant parameters appear to be the amount of training instances, the activation function of the networks, reward and cost scaling factor and depth of both critic and actor MLP.

We will use primarily the constraint score as performance metric. We start by comparing performances of the two cost functions described in Section 4.1.3, namely the dense and sparse cost function. Figure 4.15 shows 95% CI of the average constraint score on the validation set according to the cost function employed. As expected, the dense cost function appears to be significantly better, as it does not introduce a credit assignment issue in the offline RL problem.

Now, to better untangle interaction between different hyperparameters, we

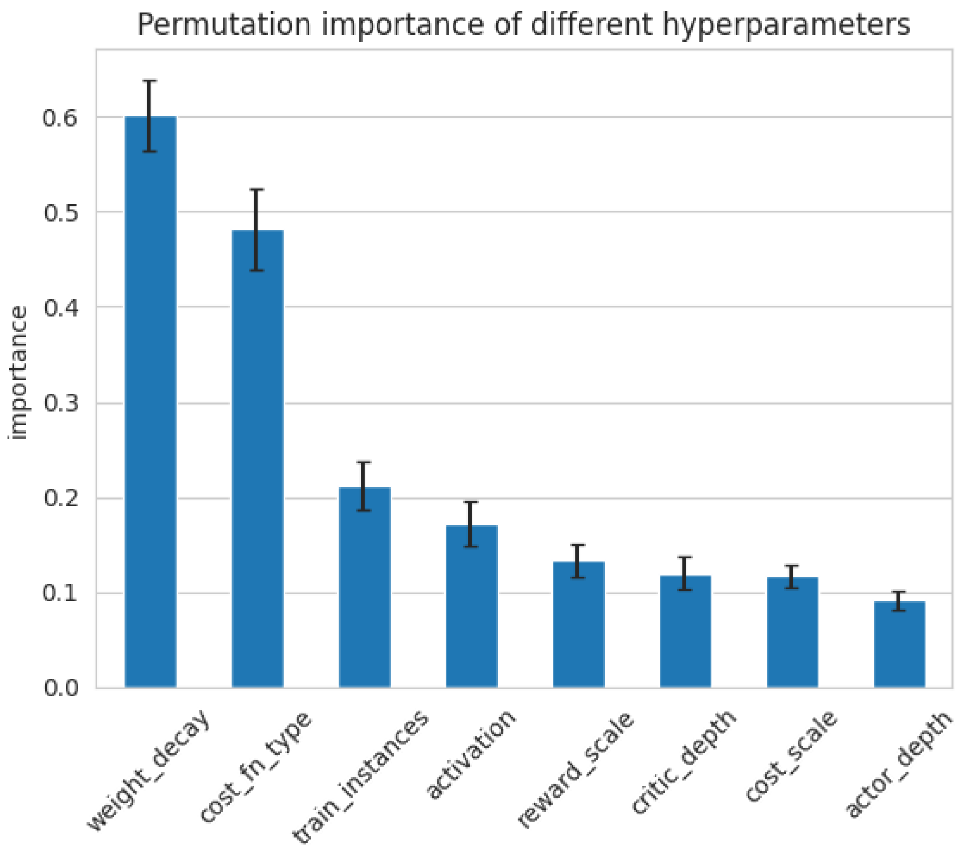

Figure 4.14: Permutation importance scores of the 8 most important hyperparameters. Bars show standard deviation. L2 regularization and the cost function employed appear to be particularly important.

will split the runs according to the cost function employed and compute CIs independently for the two groups. Figures 4.16 and 4.17 show how different hyperparameter values for actor L2 regularization and number of training instances appear to affect the performances in term of constraint score, grouped according to the cost function employed.

Concerning the L2 regularization (Figure 4.16), we can notice different behaviours between the dense and sparse cases. When the dense cost function is employed, L2 regularization seems to enhanche performance, although its coefficient requires fine-tuning. For instance, in our experiment, the highest value considered resulted in poorer outcomes compared to not using L2 regularization at all. Conversely, in the sparse case L2 regularization does not

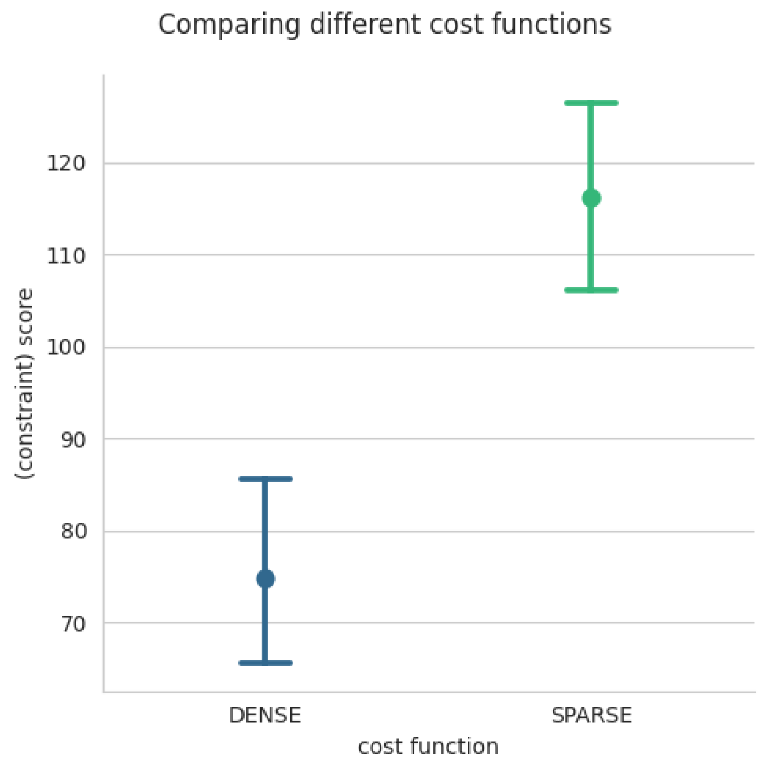

Figure 4.15: 95% CIs for constraint score (the lower the better, validation set) for different cost functions. The dense cost function appears to be significantly better, which was expected as it does not introduce a credit assignment issue in the offline RL problem.

appear be useful. Still, the reason we considered L2 regularization for the actor network in the first place was to help mitigating an issue that commonly happened in many training runs, namely the divergence of the actor's output (i.e. the virtual costs). Indeed, as the virtual cost are then employed in a LP model, several different optimal values may exist, leading to instability in the gradient-based RL optimization process [6]; this may explain why L2 regularization seems to be helping. It's not unclear though why the same reasoning does not apply in the sparse case.

Moving on to the amount of training episodes (Figure 4.17), we can see once more different behaviours between the dense and sparse case. In particular, with the dense cost function training with a larger amount of instances performs better than training with a single or few instances, whereas with the

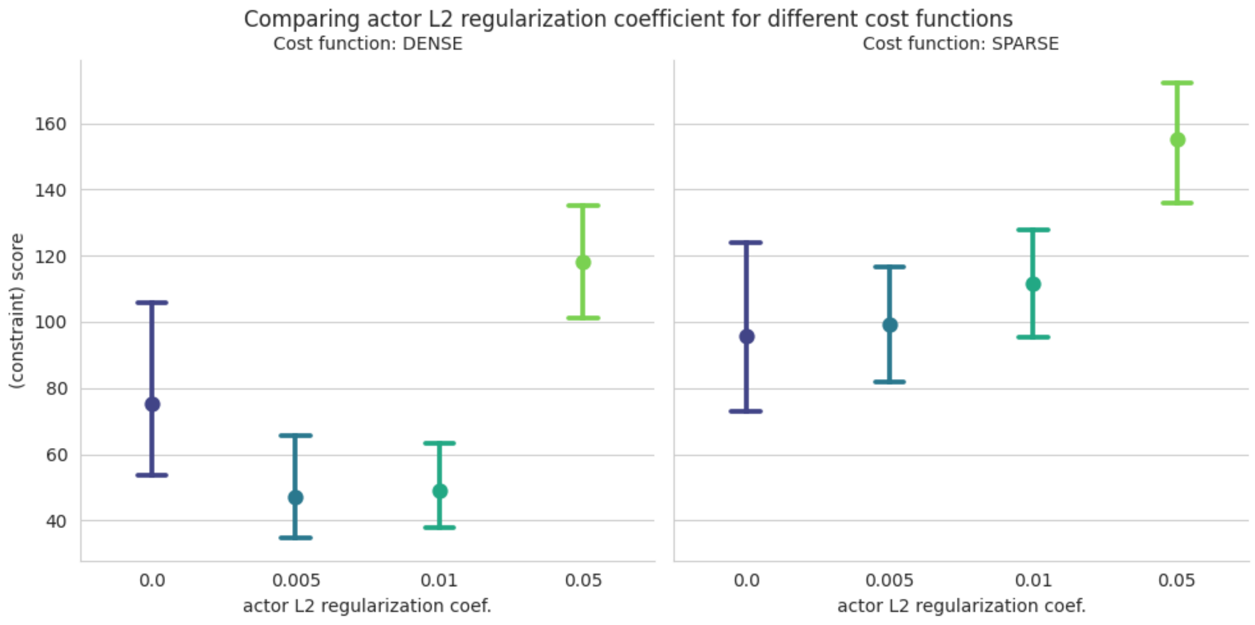

Figure 4.16: 95% CIs for constraint score (the lower the better, validation set) based on actor L2 regularization and type of cost function employed. The left plot considers the dense cost function, while the right plot considers the sparse cost function. The dense cost function appears to be overall better than the sparse cost function. Also in the dense case, L2 regularization seems to help in improving performances, although too much regularization is detrimental. In the sparse case, L2 regularization seems to have an opposite effect, leading to worse performances.

sparse cost function a single instance performs better than multiple instances.

Naive Lagrange vs PID controller

Finally, we move on to comparing two different methods to manage the Lagrangian update, namely a "naive" Lagrangian gradient-based method adopted in PPOLag [50] and the PID-based method adopted in CPPO [61].

Figures 4.18 and 4.19 show 95% CIs of average cost and average return on the validation set according to the Lagrangian method employed. Overall, we can see little difference in performances between the two methods; CIs of the PID-based method are slightly narrower than the naive gradient-based method, suggesting the more advanced PID controller effectively helps in terms of convergence. However, we noticed no substantial improvements in terms of *time*

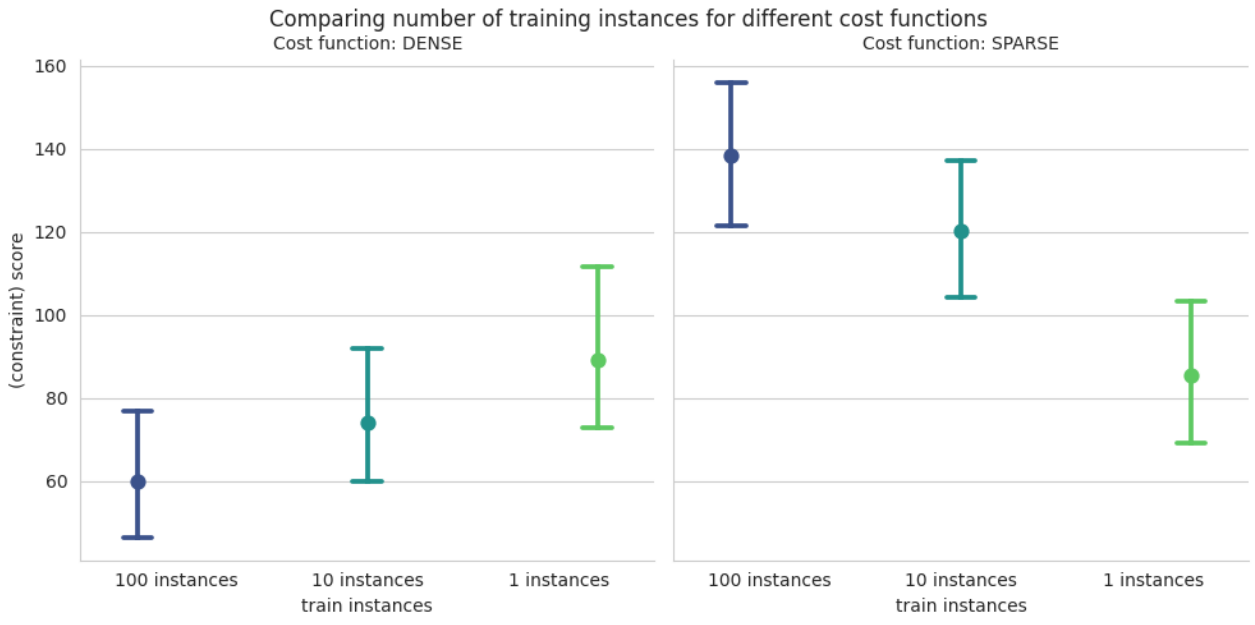

Figure 4.17: 95% CIs for constraint score (the lower the better, validation set) based on number of training instances and type of cost function employed. The left plot shows data of runs employing the dense cost function, while the right plot considers the sparse cost function. A curious behaviour can be noticed: in the dense case, more instances lead to better performances, while in the sparse case the opposite is true, with a single-instance training faring better than 10 or 100 instances. Still, overall performances appear to be better when employing the dense cost function.

to reach to convergence, which for us was a primary interest in adopting a non-gradient based method which suffers from slow convergence time. Moreover, the naive method is particularly susceptible to its hyperparameters (Lagrangian multiplier learning rate and initial value), an issue that still persists in the PID-based method, and it is possibly amplified by the larger number of hyperparameters (3 learning rates for each PID component, with their own intrinsic tuning difficulty [60], plus possibily other hyperparameters [61]).

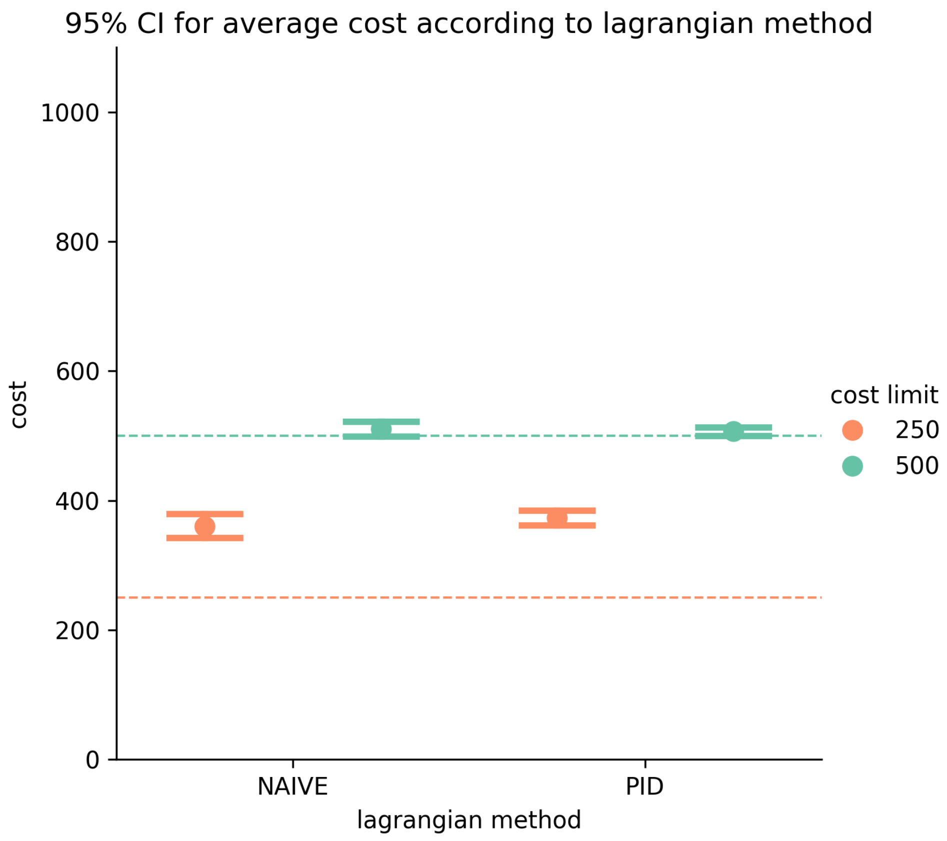

Figure 4.18: 95% CIs for average cost (validation set) based on Lagrangian method employed.

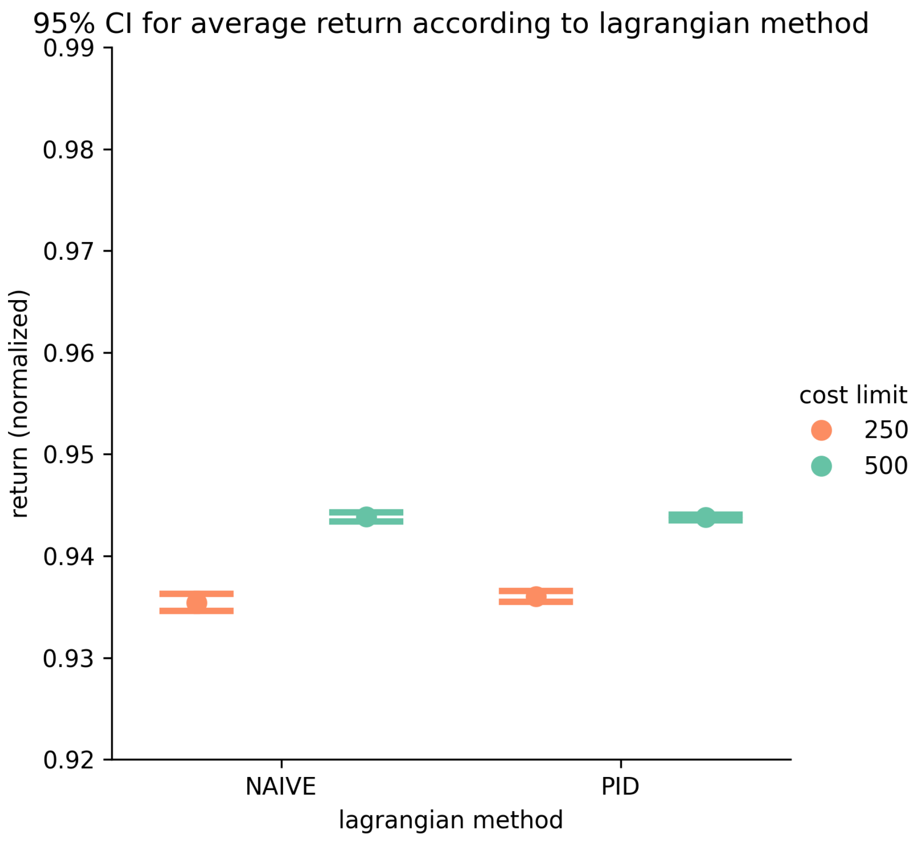

Figure 4.19: 95% CIs for average return (the lower the better, validation set) based on Lagrangian method employed.

Chapter 5

Conclusions

This thesis has presented a step forward in the interdisciplinary convergence of ML and CO, particularly focusing on the domain of CO problems within the context of RL. Our work in extending the UNIFY framework underscores the potential of hybrid learning and optimization frameworks in addressing complex decision-making problems, exemplified through the challenging case of an EMS with long-term constraints. This chapter discusses the main challenges encountered, future directions, and any additional insight that emerged from this work.

5.1 Main challenges and insights

One of the first challenges pertained to the modelling of virtual parameter vector (recall that it is the ML model output which act as (an additional) set of parameters for the CO model): this parametrization choice significantly influenced the learning process's effectiveness, in particular in terms of being able to learn to satisfy long-term constraints. This is indeed because the parametrization choice played a crucial role in determining how the RL agent could influence the online model's decisions, regarding storage management in the EMS case study. The findings suggest that the choice of virtual parameters cannot be arbitrary and requires careful consideration to reflect the problem's structure and constraints accurately.

Another issue was associated with the choice of the online optimization problem. Aside from the traditional LP online model we also experimented with a QP online model; however, the QP model did not yield any improvements in performance but rather introduced additional instability in the RL training process. This highlights the complexity of integrating offline RL agents with online optimization problems, especially when the optimization models introduce nonlinear dynamics that the RL agent must learn to navigate effectively.

The empirical evaluation and hyperparameter tuning process provided valuable insights into the factors that influence the framework's performance. The importance of cost functions in shaping learning outcomes underscores the need for mechanisms that can mitigate issues like credit assignment in RL with cumulative constraints. Furthermore, the analysis accentuates the role of hyperparameters, indicating potential areas where automated ML techniques like Bayesian optimization could streamline the tuning process, leading to enhanced performance with reduced manual intervention and time complexity. Indeed one main challenge lied in the fact that getting the RL agents to learn properly in the first place required a good deal of time, both due to the general RL requirement of long training runs (due to the low sample complexity of RL methods), the sensitivity to hyperparameters which required an accurate tuning and the need for several implementation-level details [13, 26, 27, 29, 71]. Thus, modelling and solving the offline phase of an hybrid framework such as UNIFY via RL methods definitely hampers the general applicability and scalability of the framework as a whole.

Still, several benefits arise from adopting long-term constraints in a hybrid framework such as UNIFY: it offers strategic foresight in scenarios such as EMS, allowing for proactive preparation for future demands and limitations. This approach not only enhances the system's resilience and sustainability by ensuring that today's decisions align with future objectives, but it also been shown to be capable of retaining good performance levels even under stringent requirements (e.g. a cost of 50, which implies a 95% battery capacity on average). Thus, the integration of long-term constraints stands as a cornerstone for advancing towards more anticipatory and sustainable decision-making in complex environments.

The framework's ability to accommodate stringent cost thresholds while sustaining commendable performance levels highlights its potential to revolutionize how complex problems are approached. Our findings advocate for a broader application of this methodology, suggesting that its benefits transcend the EMS context, presenting a versatile tool for tackling diverse complex, constraint-driven problems across various domains.

5.2 Future Works

Several avenues for future research have been revealed through this work. First, further exploring alternative approaches to virtual parameters modelling could enhance the RL agent's ability to influence the decisions made by the online model. Second, extending the UNIFY framework to a broader range of application domains would test its generalizability and adaptability, as each domain might present unique challenges, for instance concerning how constraints are defined as well as, of course, the nature of the decision-making problem. Such investigations could lead to a more robust and versatile framework capable of handling a wide array of complex decision-making problems. Finally, incorporating more sophisticated, powerful or even simply different RL algorithms and optimization models could definitely offer new ways to improve the framework's performance. Indeed, in our experiment we focused on PPO as the underlying RL algorithm, which is a model-free, on-policy, policy gradient-based approach; it could be that a different approach (e.g. model-based, off-policy, ...) performs better than PPO, which is known to be somewhat sample inefficient at times [26].

In conclusion, this thesis contributes to the ongoing dialogue between the fields of ML and CO, specifically through the lens of Reinforcement Learning. By addressing the inclusion of long-term constraints within the UNIFY framework, we pave the way for inventive solutions to complex decision-making problems, highlighting the potential for interdisciplinary approaches that leverage the strengths of both ML and CO.

Bibliography

[1] R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, and M. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.

[2] D. Aloini, E. Crisostomi, M. Raugi, and R. Rizzo. Optimal power scheduling in a virtual power plant. In *2011 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies*, pages 1–7. IEEE, 2011.

[3] E. Altman. *Constrained Markov decision processes*. Routledge, 1999.

[4] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[5] D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[6] D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena scientific Belmont, MA, 1997.

[7] S. Bhatnagar and K. Lakshmanan. An online actor–critic algorithm with function approximation for constrained markov decision processes. *Journal of Optimization Theory and Applications*, 153:688–708, 2012.

[8] S. Bohez, A. Abdolmaleki, M. Neunert, J. Buchli, N. Heess, and R. Hadsell. Value constrained model-free continuous control. *arXiv preprint arXiv:1902.04623*, 2019.

[9] J. C. G. Boot. On trivial and binding constraints in programming problems. *Management Science*, 8(4):419–441, 1962.

[10] A. Bou, M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, G. D. Fabritiis, and V. Moens. Torchrl: a data-driven decision-making library for pytorch, 2023. arXiv: `2306.00577 [cs.LG]`.

[11] L. Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[12] E. K. Chong, W.-S. Lu, and S. H. Zak. *An Introduction to Optimization: With Applications to Machine Learning*. John Wiley & Sons, 2023.

[13] C. Colas, O. Sigaud, and P.-Y. Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

[14] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*, 2018.

[15] G. B. Dantzig. Linear programming under uncertainty. *Management science*, 1(3-4):197–206, 1955.

[16] G. B. Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. 1990.

[17] A. De Filippo, M. Lombardi, and M. Milano. The blind men and the elephant: integrated offline/online optimization under uncertainty. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 4840–4846, 2021.

[18] F. Delbos and J. C. Gilbert. Global linear convergence of an augmented lagrangian algorithm for solving convex quadratic optimization problems. *Journal of Convex Analysis*, 12(1):25, 2005.

[19] F. Detassis. Methods for integrating machine learning and constrained optimization, 2022.

[20] P. Donti, B. Amos, and J. Z. Kolter. Task-based end-to-end model learning in stochastic optimization. *Advances in neural information processing systems*, 30, 2017.

[21] T. Eimer, M. Lindauer, and R. Raileanu. Hyperparameters in reinforcement learning and how to tune them. *arXiv preprint arXiv:2306.01324*, 2023.

[22] A. N. Elmachtoub and P. Grigas. Smart "predict, then optimize". *Management Science*, 68(1):9–26, 2022.

[23] P. Geibel. Reinforcement learning for mdps with constraints. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*, pages 646–653. Springer, 2006.

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

[25] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision sciences*, 8(1):156–166, 1977.

[26] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32 of number 1, 2018.

[27] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang. The 37 implementation details of proximal policy optimization. *The ICLR Blog Track 2023*, 2022.

[28] T. Illés and T. Terlaky. Pivot versus interior point methods: pros and cons. *European Journal of Operational Research*, 140(2):170–190, 2002.

[29] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

[30] S. Jiang, Z. Song, O. Weinstein, and H. Zhang. Faster dynamic matrix inverse for faster lps. *arXiv preprint arXiv:2004.07470*, 2020.

[31] M. A. Johnson and M. H. Moradi. *PID control*. Springer, 2005.

[32] S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 147–159, Berkeley, California, USA. Association for Computing Machinery, 1986.

[33] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[34] L. G. Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, volume 244 of number 5, pages 1093–1096. Russian Academy of Sciences, 1979.

[35] M. K. Kozlov, S. P. Tarasov, and L. G. Khachiyan. Polynomial solvability of convex quadratic programming. In *Doklady Akademii Nauk*, volume 248 of number 5, pages 1049–1051. Russian Academy of Sciences, 1979.

[36] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[37] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

[38] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[39] Y. Liu, A. Halev, and X. Liu. Policy learning with constraints in model-free reinforcement learning: a survey. In *The 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

[40] A. Lokketangen and F. Glover. Solving zero-one mixed integer programming problems using tabu search. *European journal of operational research*, 106(2-3):624–658, 1998.

[41] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1-3):397–446, 2002.

[42] K. G. Murty and S. N. Kabadi. Some NP-complete problems in quadratic and nonlinear programming. Technical report, 1985.

[43] Y. Nesterov and A. Nemirovskii. *Interior-point polynomial algorithms in convex programming*. SIAM, 1994.

[44] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[45] P. M. Pardalos and S. A. Vavasis. Quadratic programming with one negative eigenvalue is np-hard. *Journal of Global optimization*, 1(1):15–22, 1991.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: an imperative style, high-performance deep learning library, 2019. arXiv: `1912.01703 [cs.LG]`.

[47] S. Paternain, L. Chamon, M. Calvo-Fullana, and A. Ribeiro. Constrained reinforcement learning has zero duality gap. *Advances in Neural Information Processing Systems*, 32, 2019.

[48] P. Pedregal. *Introduction to optimization*, volume 46. Springer, 2004.

[49] W. B. Powell. A unified framework for stochastic optimization. *European Journal of Operational Research*, 275(3):795–821, 2019.

[50] A. Ray, J. Achiam, and D. Amodei. Benchmarking safe exploration in deep reinforcement learning. *arXiv preprint arXiv:1910.01708*, 7(1):2, 2019.

[51] C. Roos, T. Terlaky, and J.-P. Vial. Interior point methods for linear optimization, 2005.

[52] S. Sahni. Computationally related problems. *SIAM Journal on computing*, 3(4):262–279, 1974.

[53] W. Saunders, G. Sastry, A. Stuhlmueller, and O. Evans. Trial without error: towards safe reinforcement learning via human intervention. *arXiv preprint arXiv:1707.05173*, 2017.

[54] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[55] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[56] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[57] D. Silver, S. Singh, D. Precup, and R. S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021.

[58] M. Silvestri, A. De Filippo, M. Lombardi, and M. Milano. Unify: a unified policy designing framework for solving constrained optimization problems with machine learning. *arXiv preprint arXiv:2210.14030*, 2022.

[59] M. Silvestri, A. De Filippo, F. Ruggeri, and M. Lombardi. Hybrid offline/online optimization for energy management via reinforcement learning. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 358–373. Springer, 2022.

[60] O. A. Somefun, K. Akingbade, and F. Dahunsi. The dilemma of pid tuning. *Annual Reviews in Control*, 52:65–74, 2021.

[61] A. Stooke, J. Achiam, and P. Abbeel. Responsive safety in reinforcement learning by pid lagrangian methods. In *International Conference on Machine Learning*, pages 9133–9143. PMLR, 2020.

[62] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[63] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[64] C. Tessler, D. J. Mankowitz, and S. Mannor. Reward constrained policy optimization. *arXiv preprint arXiv:1805.11074*, 2018.

[65] R. J. Vanderbei et al. *Linear programming*. Springer, 2020.

[66] R. J. Vanderbei. Loqo: an interior point code for quadratic programming. *Optimization methods and software*, 11(1-4):451–484, 1999.

[67] B. Wilder, B. Dilkina, and M. Tambe. Melding the data-decisions pipeline: decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33 of number 01, pages 1658–1665, 2019.

[68] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[69] L. A. Wolsey. *Integer programming*. John Wiley & Sons, 2020.

[70] Q. Yang, T. D. Simão, S. H. Tindemans, and M. T. Spaan. Wcsac: worst-case soft actor critic for safety-constrained reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35 of number 12, pages 10639–10646, 2021.

[71] B. Zhang, R. Rajan, L. Pineda, N. Lambert, A. Biedenkapp, K. Chua, F. Hutter, and R. Calandra. On the importance of hyperparameter optimization for model-based reinforcement learning, 2021. arXiv: `2102.13651 [cs.LG]`.

[72] W. Zhang. *Branch-and-bound search algorithms and their computational complexity*. University of Southern California, Information Sciences Institute, 1996.

Acknowledgements

I’m very grateful to Mattia and Allegra for their extensive support in carrying out this thesis project. I would also like to thank my advisor Prof. Michele Lombardi. Last but not least, I am extremely grateful to Giorgia for everything she did and all the help she has given me. I wouldn’t still be here, if it wasn’t for her. Thank you from the bottom of my heart.