ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Artificial Intelligence in Industry

A COMPARATIVE ANALYSIS OF REINFORCEMENT LEARNING ALGORITHMS IN A HYBRID LEARNING AND OPTIMIZATION FRAMEWORK

CANDIDATE
Giorgia Campardo

SUPERVISOR
Prof. Michele Lombardi

Academic year 2022-2023

Session 6th

Abstract

The integration of Machine Learning (ML) and Constrained Optimization (CO) represents a promising avenue for enhancing decision-making capabilities in complex, uncertain environments. This thesis empirically evaluates the UNIFY framework, a novel approach that synergistically combines the predictive power of ML with the strategic prowess of CO. Focusing on the Energy Management System and Set Multi-cover with stochastic coverages problems, in this work we carry out a comparative analysis on the performance, efficiency, and scalability of four RL algorithms - A2C, PPO, TD3, and SAC - within the UNIFY framework. Empirical results reveal the strengths and limitations of these algorithms, highlighting SAC's superior sample efficiency and the benefits of training on multiple instances for improved model generalization. These findings underscore the potential of integrating ML and CO through RL, offering valuable insights for the development of advanced decision-making systems in various real-world applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A growing trend in recent years involves the integration and combination of methods from Machine Learning (ML) with techniques from Constrained Optimization (CO). Adopting such an hybrid approach turns out to be particularly attractive for tackling complex decision-making problems with inherent uncertainty: these problems often unfold over multiple steps and possess a partially known structure, making it difficult to naively apply traditional methods. For example, *explicit* knowledge may exist in the form of cost functions and constraints, which can be easily integrated in a CO-based system; but valuable *implicit* knowledge may also be obtained from historical data or simulations, and ML-based systems excel in this other case. Thus, it makes sense to work towards bridging and integrating methods from both classes as they show indeed different and complementary strengths and weaknesses.

In this thesis, we consider the recently-proposed UNIFY framework [55], an hybrid offline-online method that integrates ML methods, used in on the offline phase, and CO techniques, which are instead employed to address the online phase. In particular, we focus on Reinforcement Learning (RL) as the class of ML methods that we investigate, and carry out a comparative analysis of different RL algorithms within the UNIFY framework across diverse scenarios. These include the Energy Management System (EMS) problem

and the Set Multi-cover with Stochastic Coverages problem, which collectively embody the challenges of real-world optimization problems, such as constraint handling, sequential decision-making, and the management of uncertainty. Through a comprehensive empirical evaluation across various problem instances and under different training setups, we delve into the comparative efficiency, scalability, and generalization capabilities of these algorithms.

The empirical results unveil key insights into the behavior and performance of the considered RL algorithms. Notably, the algorithms exhibit differential efficiency with SAC generally outperforming others in terms of sample efficiency, particularly in the sequential decision-making setup of the EMS problem. Furthermore, training on multiple instances enhances the generalization capabilities of the RL agents, underscoring the value of diversity in training data for real-world applications. However, challenges such as action saturation in TD3 reveal the nuanced limitations that may arise in specific contexts, indicating the need for tailored approaches or algorithmic adjustments to achieve optimal performance.

These findings have profound implications for designing and implementing RL-based solutions within the UNIFY framework for complex optimization problems. The demonstrated effectiveness of integrating ML predictions with constrained optimization models through RL underscores the potential of this approach to address the dynamic and uncertain nature of real-world problems more effectively than traditional methods. The insights gained from the comparative analysis provide a foundation for future research directions, including algorithmic improvements for enhanced efficiency and scalability, employing diverse strategies for dynamic hyperparameter tuning, and the exploration of more complex problem instances that closely mirror real-world scenarios.

Overall, this thesis contributes to the evolving field of integrated ML and CO by evaluating the potential benefits and comparing the efficiency of Reinforcement Learning (RL) algorithms employed within the UNIFY framework,

paving the way for more sophisticated and effective decision-making tools.

The entire code developed for this thesis can be found in the following GitHub repository.

The remaining of this thesis is structured as follows:

- **Chapter 2: Theoretical Background.** This is the theoretical chapter which deals with all the mathematical background required for understanding the thesis. A general overview of the MO and ML fields is provided, with a focus on foundational Reinforcement Learning (RL) algorithms. Also, hybrid optimization approaches than can be related to UNIFY are discussed in this chapter.

- **Chapter 3: Experimental Setup** In this chapter, the empirical evaluation methodologies are discussed; the considered use cases (i.e. MSC and EMS problems) are also presented here, and the hyperparameter tuning process is described.

- **Chapter 4: Empirical Evaluation and Discussions.** In this chapter, results of the empirical evaluation are discussed and analyzed.

- **Chapter 5: Conclusions.** The concluding chapter provides a summary of the work done, the main contributions are highlighted and possible avenues for future works.

Chapter 2

Theoretical Background

In this chapter the theoretical background topics needed to effectively understand the UNIFY framework that lies at the core of this thesis will be briefly covered. We start with a general overview of the Mathematical Optimization (MO) field, with a focus on techniques for stochastic optimization (i.e. optimization under uncertainty) and on hybrid optimization approaches. We then discuss foundational concepts of Reinforcement Learning (RL) and describe foundational algorithms of this class. Finally, we describe the UNIFY framework and compare it with other approaches.

2.1 Mathematical Optimization

The process of decision making relies heavily on optimization, which involves selecting the most favourable option from a wide range of possibilities. The selection of the ”best” choice is guided by the desire to make an optimal decision. The quality of the available options is evaluated using an objective function or performance index. Optimization theory and methods are concerned with identifying the optimal solution based on the specified objective function. Mathematically speaking, an optimization problem involves finding the optimal solution within the set of feasible solutions. [10]

A formal definition of an optimization problem is the following:

$$\min_{x \in \mathbf{\Omega}} f(x) \tag{2.1}$$

The function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is a real-valued function known as the *objective function* or *cost function* that has to be minimized. The vector $x = [x_1, ..., x_n]^T \in \mathbb{R}^n$ of independent variables, often referred to as *decision variables*, belongs to the set $\mathbf{\Omega}$, a subset of $\mathbb{R}^n$, called the *constraint set* or *feasible set*. Whenever the decision variables are constrained to $\mathbf{\Omega} \subset \mathbb{R}^n$ the problem is a ***constrained*** *optimization problem*, on the other hand, if $\mathbf{\Omega} = \mathbb{R}^n$ it is referred to as an ***unconstrained*** *optimization problem*. The constraint $x \in \mathbf{\Omega}$ is called a *set constraint*, which takes the form $\mathbf{\Omega} = \{x : h(x) = 0,\ g(x) < 0\}$, where h and g are given functions. Such constraints are referred to as *functional constraints*. Beyond their basic structure, constrained optimization problems exhibit further diversity based on the type of constraints they impose. These constraints can manifest as linear, nonlinear, or convex relationships, while the functions themselves can be either smooth and differentiable or discontinuous and non-differentiable. Furthermore, the nature of the objective function itself adds another layer of complexity: when objective functions are convex, optimization problems fall into the category of **convex optimization**, characterized by well-understood properties and efficient solution techniques. Conversely, non-convex objective functions, which can have multiple local minima or even be discontinuous, present significant challenges and require specialized algorithms for effective solution. Within the family of **non-convex optimization problems**, one key characteristic to consider is whether the domain of the decision variables is continuous or discrete; problems belonging to the former category are **continuous optimization problems** while problems belonging to the latter are **discrete optimization problems**.

Conventionally, problems are formulated in terms of minimization for consistency and to simplify the development of optimization theory and algorithms. None the less, there are also optimization problems that require maximization of the objective function and they can be represented equivalently in the minimization form above because maximizing $f(x)$ is equivalent to minimizing $-f(x)$ ($\max f(x) = \min -f(x)$). Therefore, we can confine our attention to minimization problems without loss of generality.

The discussion of different problem types so far has implicitly assumed complete knowledge of the objective function and all constraints, which is a defining characteristic of **deterministic optimization problems**.

A deterministic optimization problem can be classified into several categories depending on the forms of the objective function, the type of constraints or the domain of the decision variables and different techniques can be applied in order to solve them based on the nature of their characteristics [36].

- whenever any of the objective function or the type of constraints are nonlinear functions, problems like 2.1 are mixed-integer nonlinear programs (MINLP);
- on the other hand, whenever the objective function and the type of constraints are all linear functions, problems 2.1 become mixed-integer linear programs (MILP);
- whenever any of the objective function or the type of constraints are nonlinear functions and there is no discrete decision variable, problems 2.1 belong to the category of nonlinear programs (NLP);
- on the other hand, whenever the objective function and the type of constraints are all linear functions and there is no discrete decision variable, problems 2.1 are linear programs (LP).

While providing powerful tools for finding optimal solutions to problems with well-defined parameters, deterministic optimization may fall short when

faced with real-world scenarios characterized by uncertainty and variability. In many practical applications, such as those in finance, engineering, and machine learning, the presence of random fluctuations, incomplete information, or inherent variability complicates the optimization process. Deterministic optimization methods may struggle to effectively address these challenges, as they rely on precise knowledge of all parameters and assumptions of deterministic behavior. Moreover, deterministic methods can reach their computational limits when dealing with complex optimization problems with many variables and intricate constraints, thus, resulting in the necessity of powerful tools such as probabilistic sampling and random exploration in order to tackle large-scale problems where deterministic approaches might struggle to find feasible solutions within reasonable time.

2.1.1 Optimization under uncertainty

Given the inherent unpredictability of real-world problems, **stochastic optimization** techniques are specifically designed to handle situations where some elements of the optimization problem are subject to randomness or uncertainty. By incorporating probabilistic elements into the problem formulation or optimization process, stochastic optimization methods offer a more realistic and robust framework to find optimal solutions in the presence of uncertainty [57]. Uncertainty is considered:

- **exogenous**: when external factors outside the decision-making process cause the uncertainty (e.g., weather conditions);
- **endogenous**: when the uncertainty arises within the system itself and future realizations can be affected by previous decisions (e.g., treatment effectiveness in recovering from illness).

Stochastic optimization problems are typically tackled through either offline or online methodologies. Offline techniques focus on generating a resilient solution by preemptively accounting for future uncertainty; however,

they often come with high computational costs. On the other hand, online algorithms make decisions after uncertainty is disclosed responding to data sequentially without any knowledge of the future; they offer faster solutions that make no assumptions about what might happen in the future, producing myopic policies that potentially compromise the solutions' quality due to limited time for computation.

This section delves into two prominent approaches for handling uncertainty in optimization: robust optimization and stochastic programming. Each approach tackles uncertainty differently, offering diverse tools to find optimal solutions under various conditions. Finally, an overview of hybrid optimization techniques is presented.

Robust Optimization

Robust optimization tackles uncertainty by considering a set of possible scenarios or **uncertainty sets**. By seeking solutions that remain feasible under all scenarios within the uncertainty set, robust optimization provides a conservative, yet practical approach to handle uncertainty relying on worst-case analysis. [21] Problems of this category are formulated as follows:

$$\min_{x \in \mathbf{\Omega}} \max_{u \in \mathcal{U}} f(x, u) \tag{2.2}$$

where the set $\mathcal{U}$ is known as the uncertainty set, which might affect both the *feasibility* or the *optimality* of the solution. In the former case, the optimization process seeks to optimize the objective over the set of solutions that are feasible for all the possible realization in the uncertainty set. The latter focuses on obtaining a solution that performs well for any realization taken by the unknown coefficients, usually by optimizing the worst-case objective.

Early research in robust optimization primarily focused on one-time decision making, where all decision variables are determined at once. However, problems with uncertainty revealed over time could be tackled by repeatedly

solving the multi-stage problem as new information arise and only implementing the decisions relevant to the current stage. As the field developed, a key research focus emerged: integrating this time-varying information efficiently into the optimization model itself, leading to a more dynamic approach. Due to the inherent complexities of multi-stage robust optimization, many theoretical works have focused on two-stage models [2][3]. Early robust optimization simplified problem complexity by treating uncertainties as fixed parameters within a defined set. This approach offered computationally efficient solutions similar to deterministic approaches, avoiding the curse of dimensionality that plagues stochastic and dynamic programming. Afterwards, researchers focused on bridging the gap between robust and stochastic optimization: quantifying robust solutions' performance under real-world, stochastic conditions and re-examining robust optimization within the context of uncertain probability distributions.

Stochastic Programming

The field of stochastic programming evolved from deterministic linear programming, with the introduction of random variables [45]. Stochastic programming aims to find an optimal solution for optimization problems where uncertainty is present in the form of random variables, modeled using probability distributions, affecting the objective function or constraints.

Stochastic programming finds applications across various fields due to its ability to handle decision-making under uncertainty: it is applied to production planning, supply chain management and risk management, as well as to portfolio optimization, risk budgeting, energy sources management and many more [24]. The generality of stochastic programming problem lends itself to be applied to a variety of problems in different fields.

In the following two common formulation of stochastic programming are presented, which have different objectives, yet, in more complex scenarios, a combination of both may be better suited to the problem at hand [52].

Chance-Constrained Formulation: this formulation ensures that the constraints are satisfied with a certain desired **level of confidence** (α). The problem with uncertain constraints which depend on a vector of random variables ξ with distribution P can be formulated as:

$$\begin{aligned} &\min_{x \in \mathbf{\Omega}} f(x) \\ &\text{s.t. } P(g_{11}(x,\xi) \leq 0, ..., g_{1k_1}(x,\xi) \leq 0) \geq \alpha_1, \\ &\quad \vdots \\ &\quad P(g_{m1}(x,\xi) \leq 0, ..., g_{mk_m}(x,\xi) \leq 0) \geq \alpha_m, \end{aligned} \tag{2.3}$$

where $\alpha = (\alpha_1, ..., \alpha_m)$ are the given confidence levels $\alpha_j \in (0, 1)$. The formulation covers the joint ($k_1 > 1$ and $m = 1$) as well as the separate ($k_j = 1$ and $m > 1$) chance-constrained problems as special cases [7].

Expected Value Minimization: this approach aims to minimize the expected value of the objective function across various possible realizations of the random variables.

$$\min_{x \in \mathbf{\Omega}} \mathbb{E}[f(x, \xi)] \tag{2.4}$$

where ξ is a vector of random variables.

Dantzig's work in 1955 [12] marked a significant milestone in stochastic programming: he introduced the **two-stage stochastic programming problem**, which has become a cornerstone concept in the field. The equation 2.4 can be divided in two stages where the first-stage decision are based on data available at that time and should not depend on future observations, yielding the following formulation:

$$\min_{x \in \mathbf{\Omega}} f(x) + \mathbb{E}[\mathbf{Q}(x, \xi)] \tag{2.5}$$

where $\mathbf{Q}(x, \xi)$ is the optimal value of the second-stage problem:

$$\begin{aligned} &\min_{y \in \mathbb{R}^m} q(y, \xi) \\ &\text{s.t. } T(\xi)x + W(\xi)y = h(\xi) \end{aligned} \tag{2.6}$$

where $x \in \mathbb{R}^n$ is the first-stage decision vector, $\mathbf{\Omega} \subseteq \mathbb{R}^n$ set of feasible solutions, $y \in \mathbb{R}^m$ is the second-stage decision vector and $\xi(q, T, W, h)$ contains the data of the second-stage problem to be solved after a realization of ξ becomes available [53, 5].

Multi-stage Stochastic Programming Despite its seemingly static nature of the two-stage stochastic programming model (2.5), it incorporates a subtle dynamic element in its formulation: the sequential nature of function $\mathbf{Q}$, which is evaluated only after x is being instantiated, is an essential element of decision making under uncertainty. Moreover, the model 2.5 assumes that the uncertainties are realized only once after the first-stage decisions are made. However, many real-world situations involve a series of decisions based on evolving information. **Multistage stochastic programming** addresses the sequential realization of uncertainties by allowing them to unfold gradually over discrete time periods called *stages*. Each stage incorporates the uncertainties revealed at that specific point in time. The model 2.5 may be perceived as the first stage of a more extensive multi-stage formulation whenever $\mathbf{Q}$ is defined recursively.

Consider an N-stage problem: let the boundary conditions be given by $\mathbf{Q}_{N+1} \equiv 0$, and let ξ_0 denote a degenerate random variable reflecting the deterministic information available prior to decisions of stage 1. For $t = 1, ..., N$, let ξ^t denote the history prior to stage t [i.e. $\xi^t = (\xi_0, ..., \xi_{(t-1)})$]. Note that the decision variables at stage t depend on the history of the data process. Hence these variables are functions of random variables, and will be denoted $x_t(\xi^t)$. The entire history of decisions until stage t will then be represented as a superscripted vector $x_t(\xi^t) = (x_1(\xi^1), x_2(\xi^2), ..., x_t(\xi^t))$, or simply x^t. For

$t = 2, ..., N$, define the value functions [52]:

$$\begin{aligned} \mathbf{Q}_t(x^{t-1}, \xi^t) = \min_{x_t \in \mathbf{\Omega}_t(x^{t-1}, \xi^t)} & f_t(x_t; x^{t-1}, \xi^t) + \mathbb{E}[\mathbf{Q}_{t+1}(x^t, \tilde{\xi}^{t+1} | \xi^t)] \\ \text{s.t. } & T_t(\xi^t) x_{t-1}(\xi^{t-1}) + W_t x_t(\xi^t) = h_t(\xi^t) \end{aligned} \tag{2.7}$$

2.1.2 Hybrid Optimization Approaches

In numerous real-world scenarios, a significant amount of information about the stochastic variables is known before the uncertainty is disclosed. This drives the desire to develop hybrid offline/online approaches that leverage the strengths of both methods to enhance solution quality and computational efficiency. This section provides a brief overview of various hybrid optimization approaches, offering a broad perspective of the field.

Many multi-stage optimization problems under uncertainty involve two distinct phases: a strategic offline phase and an operational online phase. While the offline phase allows for extensive planning (with minimal time constraints), the online phase is often subject to strict time limitations, requiring decisions to be made quickly [13]. Given an n-stage stochastic optimization problem modelled as a Markov Decision Process [47], the use of Dynamic Programming as a solution method can be seen as a hybrid offline/online optimization approach. The policy and its corresponding value-function are iteratively improved offline, simulating executions, and then the resulting policy can be efficiently executed online. In recent years, researchers have proposed hybrid stochastic/robust models (e.g. Zhao and Guan [70]) to address the limitations of purely stochastic and robust approaches. These models aim to leverage the strengths of both methodologies for better decision-making under uncertainty in complex domains.

The ever-growing volume of data available in last decades presents a significant advantage for tackling stochastic optimization problems: this vast amount of information allows us to gain insights to inform and enhance the decision-making processes involved in optimization problems. In particular,

machine learning algorithms have emerged as powerful tools for extracting valuable knowledge and uncovering hidden patterns from vast and intricate datasets [44]. The area of machine learning has three major sub-fields:

- **Supervised learning** focuses on training models using labeled data, where each data point has a corresponding label or target value. The goal is to learn a mapping function from the input data to the corresponding labels.
- **Unsupervised learning** deals with unlabeled data, where the data points do not have predefined labels. The goal is to uncover hidden patterns and structures within the data.
- **Reinforcement learning** involves training an agent to interact with an environment through trial and error. The agent receives rewards or penalties for its actions, and the goal is to learn an optimal policy to maximize the long-term reward.

Integrating the insights gleaned from machine learning models can significantly enhance the decision-making capabilities of optimization models in several ways. Firstly, by leveraging insights gleaned from data, a more accurate modeling of uncertainties can be achieved. This empowers the optimization process to account for potential future variations with greater precision, leading to more robust and reliable outcomes. While stochastic programming assumes perfect knowledge of the probability distribution of uncertain parameters, such detailed information is often unavailable in real-world situations. Relying solely on an assumed probability distribution in conventional stochastic programming, as it may not reflect real-world distributions, can lead to sub-optimal or even detrimental out-of-sample results [56].

Recognizing the limitations of traditional stochastic programming, **Distributionally Robust Optimization** (DRO) [23] emerges as a new data-driven approach. It tackles uncertainty by considering the ”worst-case” scenario

within a set of possible probability distributions, referred to as *ambiguity set*, rather than relying on a single assumed distribution. Unlike stochastic programming, DRO leverages statistical inference and big data analytics to construct the *ambiguity set* containing various potential distributions based on available uncertainty data. This allows DRO to hedge against potential errors in the assumed distribution, effectively incorporate real-world uncertainty data into the decision-making process and having various applications in power systems, such as unit commitment problems [68, 9, 17, 69], and optimal power flow [62, 27].

Secondly, the integration of machine learning unlocks the power of predictive analytics. By analyzing historical data and identifying trends, the model can anticipate future events and proactively guide the decision-making process. This proactive approach allows for capitalizing on opportunities and mitigating potential risks, leading to improved overall performance. A common name that identifies this paradigm is **predict-then-optimize**, where the optimization model is used to generate decisions, based on key parameters predicted by a machine learning algorithm. A growing trend in analytic solutions for real-world challenges is the combination of prediction and optimization, see Chan, Farias, Bambos, and Escobar [8], Deo, Rajaram, Rath, Karmarkar, and Goetz [16], Gallien, Mersereau, Garro, Mora, and Vidal [22], Mehrotra, Dawande, Gavirneni, Demirci, and Tayur [40], and den Hertog and Postek [15] as examples. Most machine learning tools are not designed to consider how their predictions will be used in subsequent optimization tasks. Recent advancements led to the development of machine learning tools that take into account how their predictions will be used in optimization problems: the general framework called **Smart "Predict, then Optimize"** (SPO)[18] explicitly uses the structure of the nominal optimization problem in order to evaluate the quality of the prediction. In the *SPO* framework, the prediction model is optimized in order to generate predictions that aim to minimize decision error, not prediction error: the loss is the true cost of the decision induced by the

predicted parameters minus the optimal cost under the true parameter after its realization. This approach has numerous applications in different real-world scenarios, including:

- **Vehicle routing** problems, where the cost of each edge of a graph needs to be predicted before making a routing decision.
- **Inventory management**, where the production demands are the key input into the optimization model.
- **Portfolio optimization**, where the returns of potential investments can be estimated from data.

Finally, machine learning models possess the remarkable ability to continuously learn and adapt as they are exposed to new data. This allows the optimization model to dynamically adjust its strategies in response to evolving conditions, ensuring its continued effectiveness in the face of changing circumstances. This sequential ecision-making process closely aligns with *Reinforcement Learning* (RL), which will be covered in detail in Section 2.2. The interplay between RL and stochastic optimization has been extensively explored by [46], emphasizing their shared goal of finding optimal decision-making strategies over time in dynamic and uncertain environments.

In essence, machine learning models act as powerful informants for the optimization process, enabling it to operate in a dynamic and data-driven manner. This synergy between optimization and machine learning paves the way for more informed, adaptable, and effective decision-making in an ever-evolving world.

2.2 Reinforcement Learning

Learning through interaction is a fundamental principle in many theories of learning and intelligence. This includes infants, who learn by interacting with

their environment through exploration and experimentation. This goal-directed learning through interaction is also the core concept behind **Reinforcement Learning**. It is simultaneously a problem, a class of solution methods that work well on such problems, and the field that studies these problems and their solution methods. Three key characteristics distinguish reinforcement learning problems:

- *closed-loop system*: the learner's actions directly influence future inputs and experiences.
- *trial and error*: the learner discovers the most rewarding actions through experimentation, not explicit instruction.
- *long-term effects*: actions can impact not only immediate rewards but also future situations and rewards.

These characteristics differentiate RL from other learning approaches and highlight its unique approach to learning through interaction and exploration [59].

In the reinforcement learning framework, the *agent* represents the learner or decision-maker who interacts continually with the *environment*, which provides new situations and generates rewards, special numerical values that the agent tries to maximize over time. A complete specification of the environment defines a specific RL task.

More specifically, the interaction between an agent and the environment happens over discrete time steps ($t = 0, 1, 2, 3, ...$): at each time step (t), the agent observes the environment's state, $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states; based on the state, the agent selects an action $A_t \in \mathcal{A}(S_t)$ from the set of available actions in state S_t, denoted by $\mathcal{A}(S_t)$; one time step later the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and finds itself in a new state, S_{t+1} (refer to Figure 2.1 for a visual representation of this interaction loop). The sequence of all states, actions and rewards generated is called *trajectory*(τ) which looks like $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$

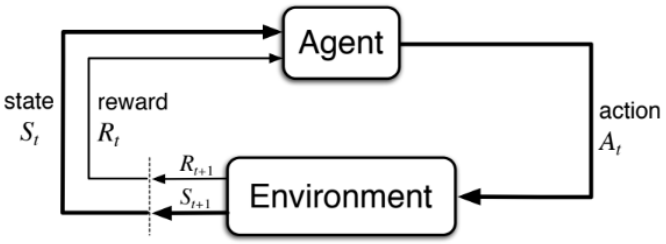

Figure 2.1: The agent–environment interaction in reinforcement learning. From Sutton and Barto [59]

The framework that is usually employed to formally describe RL dynamics is that of Markov Decision Processes (MDPs). A MDP is usually defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \rho_0)$, where $\rho_0(s)$ is the starting state distribution (i.e. the probability that state s is the first state encountered by the agent) and the function $P : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ defines the dynamics of the system: for particular values of $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the immediately preceding state and action S_{t-1} and A_{t-1}(also known as the *Markov Property*). A formal definition of P follows:

$$\begin{gathered} P(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \\ \forall s, s' \in \mathcal{S}, \forall r \in \mathcal{R}, \forall a \in \mathcal{A}(s) \end{gathered} \tag{2.8}$$

This function is the starting point for anything else one might want to compute about the environment's dynamics, such as state-transition probabilities, but, most importantly, the expected rewards for state-action pairs ($r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$):

$$r(s, a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} P(s', r|s, a) \tag{2.9}$$

[59].

The tasks the agent undertakes can usually be classified into two types:

- **Episodic tasks** are characterized by *episodes*, identified by a clear beginning and end: each episode ends in a terminal state after T steps,

followed by a reset to a standard starting state or to a sample of a distribution of starting states. The following episode is completely independent from the previous one.

- **Continuing tasks**, on the other hand, do not have a defined endpoint. These tasks are ongoing, and the agent aims to continuously collect rewards over an indefinite period.

In reinforcement learning, the ultimate objective of the agent is to act in a way that maximizes its rewards over time. However, the notion of "time" introduces complexity because rewards received in the immediate future are often considered more valuable than those received in the distant future. In the case of continuing tasks, the distinction between immediate and distant future rewards becomes critical: without a terminal state to provide natural breaks in the decision-making process, the agent must meticulously balance the value of immediate rewards against those that it might receive in the distant future.

It is possible to overcome this challenge through the introduction of a *discount factor* denoted as $\gamma \in (0, 1]$, whose purpose is to weight rewards received at varying time distances, essentially acting as a trade off between immediate and future rewards.

Given this, the agent's goal can be encapsulated by an objective function that seeks to maximize the **expected return**: the discounted sum of future rewards. This is mathematically represented as:

$$\mathcal{G}_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.10}$$

where $\mathcal{G}_t$ represents the total discounted reward from time step t on. R_{t+k+1} is the reward received after $k+1$ time steps from time t, and γ^k effectively discounts the reward's value. This notation includes the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

The agent, through its interactions with the environment, seeks a policy,

π, which governs how it selects actions based on the current state. This policy is aimed at maximizing not just the immediate reward, but the cumulative discounted rewards it expects to receive over the future. Formally, the optimal policy, denoted as π^*, is the one that maximizes the expected sum of discounted rewards from any given state, s, as:

$$\begin{aligned} \pi^* &= \arg\max_{\pi} J(\pi) \\ &= \arg\max_{\pi} \mathbb{E}_{\pi}[\mathcal{G}_t] \\ &= \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T} \gamma^t r(S_t, A_t) | S_0 = s, \pi\right] \end{aligned} \tag{2.11}$$

Here, $\mathbb{E}_{\pi}[\cdot]$ denotes the expectation, which considers that actions are selected according to policy.

At the core of most reinforcement learning algorithms lies the concept of value functions [59]. These functions, defined for states or state-action pairs, estimate the expected future benefit the agent can anticipate from being in a particular state (or taking a specific action in that state). This "benefit" is often referred to as the expected return, which considers the sequence of rewards the agent might receive in the future. The **state value function**, denoted $V^{\pi}(s)$, quantifies the expected long-term return an agent can achieve starting from state s and following policy π thereafter: $V^{\pi}(s) = \mathbb{E}_{\pi}[\mathcal{G}_t | S_t = s]$. It has its optimal counterpart denoted as: $V^*(s) = \max_{\pi} V^{\pi}(s)$, that follows the optimal policy π^*. Similarly, we define the value of taking action a in state s under a policy π, namely **action-value function**, denoted $Q^{\pi}(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π: $Q^{\pi}(s, a) = \mathbb{E}_{\pi}[\mathcal{G}_t | S_t = s, A_t = a]$, with the optimal counterpart denoted as: $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$, acting according to the optimal policy π^*.

In the following sections, approximation of value functions and policies will be carried out through the use of **neural networks** since they are widely used for nonlinear function approximation [11]. The sets of parameters will be

denoted by letters such θ, ψ or ϕ, and then written as a subscript on the policy or value functions symbols: $\pi_\theta, V_\psi, Q_\phi$.

Action-Value Methods

The Bellman equation captures a fundamental property of the optimal action-value function $Q^*(s,a)$: if $Q^*(s',a')$ was known for all possible next states s' and actions a', then the optimal strategy would be to select the action a' maximising the expected value of $r + \gamma Q^*(s',a')$:

$$Q^*(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \tag{2.12}$$

Reinforcement learning algorithms leverage this principle to iteratively estimate the action-value function by applying the Bellman equation as an iterative update $Q_{i+1}(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q_i(s',a')|s,a]$ that converges to the optimal action-value function as $i \to \infty$ [59].

Training algorithm for deep Q-networks. In practice, it is common to use a function approximator to estimate the action-value function: let $Q_\phi(s,a)$ be an approximate action-value function with network parameters ϕ, referred to as Q-network $Q_\phi(s,a) \approx Q^*(s,a)$ [42]. Q-learning algorithm [63] is *model-free*, meaning it doesn't feature, nor learn, a model of the environment (i.e. a function that predicts state transitions and rewards), and *off-policy*: it learns about the greedy policy $a = \arg\max_{a'} Q_\phi(s,a')$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often implemented by an ϵ-greedy policy that follows the greedy policy with probability $1-\epsilon$ and selects a random action with probability ϵ. Off-policy learning benefits from the use of experience replay [38]: the agent stores experiences as tuples (s_t, a_t, r_t, s_{t+1}) at each time-step t in a replay buffer $\mathcal{D}$. During learning, Q-learning updates are calculated on samples (or minibatches) of experience $(s,a,r,s') \sim \mathcal{D}$, drawn uniformly

at random from the pool of stored samples. The Q-network can be trained by minimizing a sequence of loss functions $L_i(\phi_i)$ that changes at each iteration i:

$$L_i(\phi_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q_{\bar{\phi}_i}(s', a') - Q_{\phi_i}(s, a)\right)^2\right] \tag{2.13}$$

where ϕ_i are the parameters of the Q-network at iteration i and $\bar{\phi}_i$ are the network parameters used to compute the target at iteration i. The target network parameters $\bar{\phi}_i$ are only updated with the Q-network parameters ϕ_i every C steps (or at every step through Polyak's averaging [37]) and are held fixed between individual updates [43].

Policy Gradient Methods

Having explored action-value methods, which rely on calculating action-values, this section introduces policy gradient methods. These methods directly learn a parameterized policy, denoted as π_θ, which can select actions without requiring a value function. While a value function can contribute to learning the policy's parameters, it's not essential for action selection [59]. The notation $\pi_\theta(a|s)$ denotes the probability that, at time t with parameters θ, the action a is chosen given the environment is in state s.

Policy gradient methods focus on learning the optimal policy parameters leveraging the gradient of a performance measure, $J_\pi(\theta)$, which is a scalar value indicating how well the agent performs under policy π_θ. The ultimate goal is to maximise this performance measure, achieved through gradient ascent updates approximated as follows: $\theta_{t+1} = \theta_t + \alpha^\theta \nabla \hat{J}_\pi(\theta_t)$, where $\nabla \hat{J}_\pi(\theta_t) \in \mathbb{R}^d$ represents a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to the argument $\theta \in \mathbb{R}^d$.

For the sake of simplicity and without loss of generality, we will focus on the episodic case; similar derivations are provided for the continuing case (see Sutton and Barto [59]). In the episodic case, the performance measure is

defined as $J_\pi(\theta) \doteq V^{\pi_\theta}(s_0)$ where V^{π_θ} is the true value function for policy π_θ. The *policy gradient theorem* provides an analytic expression for the gradient of performance with respect to the policy parameter that does not involve the derivative of the state distribution:

$$\begin{aligned}
\nabla J_\pi(\theta) =& \nabla V^{\pi_\theta}(s_0) \\
\propto& \sum_s \mu(s) \sum_a Q^\pi(s,a) \nabla \pi_\theta(a|s) \\
=& \mathbb{E}_\pi \left[Q^\pi(S_t, A_t) \nabla \pi_\theta(A_t|S_t) \right] \qquad \text{replaced } a, s \text{ with } A_t \sim \pi, S_t \sim \pi \\
=& \mathbb{E}_\pi \left[\mathcal{G}_t \frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \right]
\end{aligned}$$

where μ is the on-policy distribution under π.

This derivation yields the **REINFORCE** update [65] :

$$\theta_{t+1} \doteq \theta_t + \alpha^\theta \mathcal{G}_t \frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{2.14}$$

This can be generalized to include a comparison of the action value to an arbitrary **baseline** $b(s)$, any function that does not vary with a:

$$\begin{aligned}
\theta_{t+1} &\doteq \theta_t + \alpha^\theta (\mathcal{G}_t - b(S_t)) \frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \\
&= \theta_t + \alpha^\theta (\mathcal{G}_t - b(S_t)) \nabla \log \pi_\theta(A_t|S_t)
\end{aligned} \tag{2.15}$$

One natural choice for the baseline is an estimate of the state value $\hat{V}^\pi_\psi(S_t)$, which reduces variance but introduces bias [25]. The simplest method for learning $\hat{V}_\psi$ is to minimize a mean-squared-error objective similarly to 2.13 by updating neural network parameters ψ as follows:

$$\psi_{t+1} \doteq \psi_t + \alpha^\psi (\mathcal{G}_t - V^\pi_\psi(S_t)) \nabla V^\pi_\psi(S_t) \tag{2.16}$$

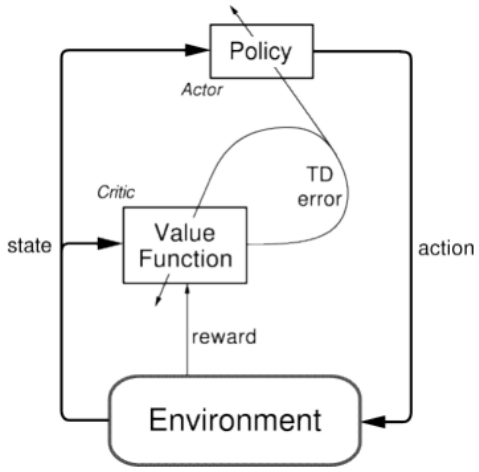

Figure 2.2: The actor-critic architecture. From Sutton and Barto [59]

Actor-Critic Methods

Methods that concurrently learn approximations of both the policy and value function are categorized as **actor-critic methods**. In these methods, the ”actor” refers to the learned policy, while the ”critic” represents the learned value function. Although REINFORCE with baseline involves learning both a policy and state-value functions, it is not classified as an actor-critic method. The key difference lies in how the value function is utilized: in REINFORCE with baseline, the state-value function acts solely as a baseline for comparison, not as a critic directly influencing policy updates. On the other hand, typical actor-critic methods leverage TD (temporal-difference) learning [58], which inherently involves bootstrapping. This learned value function then serves as a critic, providing feedback for refining the actor (policy). See Figure 2.2.

The one-step actor-critic method replaces the full return of REINFORCE (2.15) with the one-step return using a learned state-value function as baseline as follows:

$$\begin{aligned}\theta_{t+1} &\doteq \theta_t + \alpha(R_{t+1} + \gamma \hat{V}^{\pi}_{\psi}(S_{t+1}) - \hat{V}^{\pi}_{\psi}(S_t)) \frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \\ &= \theta_t + \alpha \delta_t \nabla \log \pi_\theta(A_t|S_t)\end{aligned} \tag{2.17}$$

2.2.1 A2C

It was first proposed in Mnih et.al. [41], where the authors actually describe an asynchronous algorithm called **A3C**; lately empirical studies have shown that good performances can be obtained by its synchronous version (A2C) [67]. **Advantage Actor-Critic** (A2C) is the simplest actor-critic algorithm: it maintains a policy $\pi_\theta(a_t|s_t)$ and an estimate of the value function $V^\pi_\psi(S_t)$. The update performed by the algorithm can be seen as a generalization of 2.17:

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla A^\pi(S_t, A_t) \log \pi_\theta(A_t|S_t) \tag{2.18}$$

where $A^\pi(S_t, A_t)$ is the **advantage function**, generally defined as the relative advantage of an action compared to the average value of all possible actions in a given state: $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. This function yields almost the lowest possible variance, though in practice, it is not known and must be estimated [50]. The estimate can be as simple as the TD(0) $\delta_t = R_{t+1} + \gamma \hat{V}^\pi_\psi(S_{t+1}) - \hat{V}^\pi_\psi(S_t)$ employed in one-step actor-critic. Even though, a more powerful estimate has been proposed by Schulman et. al. [50], the **Generalized Advantage Estimator (GAE)**, in order to balance the bias-variance trade-off:

$$\hat{A}^{GAE(\gamma,\lambda)}_t := \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+1} \tag{2.19}$$

The policy and the value function are updated after every t_{max} actions or when a terminal state is reached. Moreover, the entropy of the policy π can be added to the objective function in order to improve exploration by discouraging premature convergence to sub-optimal deterministic policies, as first proposed by Williams & Peng [66]:

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla A^\pi(S_t, A_t) \log \pi_\theta(A_t|S_t) + \beta \nabla \mathcal{H}(\pi_\theta(\cdot|S_t)) \tag{2.20}$$

where $\mathcal{H}$ is the entropy and the hyper-parameter β controls the strength of the entropy regularization term.

2.2.2 PPO

A simple yet effective method to improve the performances of policy gradient methods is that of enforcing a *trust region* when updating the policy, that is, limiting how much the policy can move in the parameter space during a single update step. The first notorious example of this class of techniques is **Trust Region Policy Optimization** (TRPO) [49], which enforces the trust region constraint via a KL divergence constraint between old and new policy. TRPO features theoretical guarantees of monotonic improvement at each policy update step, yet its reliance on second-order optimization practically becomes expensive and not scalable.

This challenge of relying on second-order optimization was later solved by **Proximal Policy Optimization** (PPO) [51], which proposed a simpler trust region mechanism by employing a surrogate objective function that can optimized with a first-order method. The general problem addressed by trust region methods in RL can be formulated as follows:

$$\begin{aligned} \pi^* &= \arg\max_{\pi} J_\pi(\theta) \\ s.t.\ &\overline{D}_{KL}(\pi_{\theta_t}, \pi_{\theta_{t+1}}) \leq \delta \end{aligned} \tag{2.21}$$

where $\overline{D}_{KL}$ is an average KL-divergence between policies. PPO authors propose two variant of the surrogate objective:

- **PPO-Penalty** incorporates KL divergence into the objective function as a penalty term and dynamically adjusts the penalty coefficient throughout training to ensure appropriate scaling.
- **PPO-Clip** doesn't take into consideration a KL-divergence term in the objective, it relies instead on specialized clipping in the objective function.

For the sake of simplicity and without loss of generality, we'll focus on PPO-Clip since in the original experiments by [51], researchers found it to be the

best performing among the proposed variants. Let $pr_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ denote the probability ratio between policies after an update, so that $pr_t(\theta_{\text{old}}) = 1$. The new surrogate objective is transformed as follows:

$$J_\pi(\theta) = \mathbb{E}\left[\min(pr_t(\theta)\hat{A}_t, \text{clip}(pr_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{2.22}$$

where epsilon is a small hyperparameter, usually $\epsilon = 0.2$. Strategically clipping the probability ratio to be in the interval $[1-\epsilon, 1+\epsilon]$ removes the incentive for moving too far away from the old policy. Similarly to 2.20, the objective can be augmented by adding an entropy bonus to ensure sufficient exploration.

2.2.3 DDPG

Deep Deterministic Policy Gradient (DDPG) [37] can be seen as a combining ideas from the DQN algorithm [43] — notably the use of a target network and the experience replay mechanism — with those from policy gradient methods, resulting in a model-free, off-policy algorithm that learns policies in continuous action domains. In particular, DDPG employs the same principle of *deterministic policy gradient* [54] where the policy function directly maps states to actions, denoted by $\mu_\theta(s)$, with θ representing the parameters of the policy network. This contrasts with stochastic policies used in algorithms like A2C, PPO and SAC, providing a more straightforward approach for continuous actions by eliminating the need to sample them. DDPG also incorporates two key techniques from DQN to stabilize training that were discussed in paragraph 2.2: it adopts target networks for both the actor and the critic, denoted as $\mu_{\bar{\theta}}$ and $Q_{\bar{\phi}}$ respectively, as well as employing an experience buffer replay to store the agent's experiences.

The critic is trained by minimizing the mean squared Bellman error (notice

the usage of the actor target network when computing the target):

$$\begin{aligned} L_Q(\phi) &= \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1},d)\sim\mathcal{D}}\left[\left(Q_\phi(s_t,a_t) - y_t(r_{t+1},s_{t+1},d)\right)^2\right] \\ \text{where} \quad y_t(r_{t+1},s_{t+1},d) &= r_{t+1} + \gamma(1-d)Q_{\bar{\phi}}(s_{t+1},\mu_{\bar{\theta}}(s_{t+1})) \end{aligned} \tag{2.23}$$

and $\mathcal{D}$ is the replay buffer. Then, the actor is learned simply by performing gradient ascent with respect to policy parameters on:

$$\max_\theta \mathbb{E}_{s_t\sim\mathcal{D}}\left[Q_\phi(s_t,\mu_\theta(s_t))\right] \tag{2.24}$$

Exploration poses a significant challenge when learning in continuous action spaces. However, off-policy algorithms like DDPG offer an advantage: the exploration process can be addressed independently from the learning algorithm itself. An exploration policy μ' is constructed by augmenting the action of policy μ_θ with sampled noise from a noise process $\mathcal{N}$:

$$\mu'(s_t) = clip(\mu_\theta(s_t) + \epsilon, a_{low}, a_{high}) \quad \text{where } \epsilon \sim \mathcal{N} \tag{2.25}$$

In the original version by [37], $\mathcal{N}$ was chosen to be an Ornstein-Uhlenbeck process [60].

2.2.4 TD3

Twin Delayed Deep Deterministic policy gradient (TD3) [20] builds upon DDPG by introducing key improvements aimed at addressing the problem of *overestimation bias* in value function estimation, which can significantly impact the performance and stability of training in deep reinforcement learning algorithms. TD3 specifically targets continuous control tasks and proposes three critical enhancements to the basic DDPG framework:

1. **Target Policy Smoothing:** TD3 adds noise to the target action, sampled from a clipped normal distribution, to smoothen the policy. This

smoothing is applied when computing the target values during critic updates, making the algorithm less sensitive to outliers:

$$\bar{a}(s_{t+1}) = \text{clip}(\mu_{\bar{\theta}}(s_{t+1}) + \text{clip}(\epsilon, -c, c), a_{low}, a_{high}), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{2.26}$$

where c is a small constant, preventing excessive deviation from the original action $\mu_{\bar{\theta}}(s_{t+1})$.

2. **Clipped Double-Q Learning:** TD3 learns two critic networks (in particular Q-networks) concurrently, $Q_{\phi_1}(s, a)$ and $Q_{\phi_2}(s, a)$, and uses the minimum of their predictions to update the action values. This approach is inspired by Double-Q Learning [30] and is designed to mitigate overestimation by taking the minimum estimate from the two critic networks:

$$y(r_{t+1}, s_{t+1}, d) = r_{t+1} + \gamma(1 - d) \min_{i=1,2} Q_{\bar{\phi}_i}(s_{t+1}, \bar{a}(s_{t+1})) \tag{2.27}$$

where $Q_{\bar{\phi}_i}$ are the target networks of the two critic networks, and $\mu_{\bar{\theta}}$ is the target network for the actor. The two critic networks are learned by regressing the following target:

$$J_Q(\phi_i) = \mathbb{E}_{(s_t, a_t, r_{t+1}, s_{t+1}, d) \sim \mathcal{D}} \left[(Q_{\phi_i}(s_t, a_t) - y(r_{t+1}, s_{t+1}, d))^2 \right] \tag{2.28}$$

3. **Delayed Policy Updates:** Policy and target networks are updated less frequently than the critic networks, specifically every `policy_delay` steps, where `policy_delay` > 1. This delay reduces the impact of value overestimation on policy updates, promoting a more stable and reliable improvement.

Lastly, the policy is learned just as 2.24, but employing Q_{ϕ_1} as critic. TD3 designs an exploration policy like 2.25 where the noise is sampled from

$\mathcal{N}(0, \sigma)$. Moreover, in order to further improve exploration and remove the dependency on the initial policy parameters, a completely random policy runs on the environment for the first `init_random_frames`.

Together, these innovations form the TD3 algorithm, achieving notable improvements in stability and performance over its predecessor DDPG [20].

2.2.5 SAC

Soft Actor Critic considers a more general maximum entropy objective (see e.g. [71]) that generalizes the standard objective 2.11 by augmenting it with an entropy term, such that the optimal policy additionally aims to maximize its entropy at each visited state:

$$\pi^* = \arg\max_{\pi} \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \gamma^t [r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))] \tag{2.29}$$

where α is the temperature parameter that determines the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy. The entropy is calculated as $\mathcal{H}(\pi(\cdot|s)) = -\log \pi(\cdot|s)$.

In this different settings, the value function and the state-value function change in order to include the entropy term:

$$V^\pi(s) = \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[\sum_{t=0}^{\infty} \gamma^t (r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)))|s_0 = s \right] \tag{2.30}$$

$$Q^\pi(s, a) = \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[\sum_{t=0}^{\infty} \gamma^t (r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)))|s_0 = s, a_0 = a \right] \tag{2.31}$$

where V^π and Q^π are connected by $V^\pi(s) = \mathbb{E}_{a\sim\pi} [Q^\pi(s, a)] + \alpha \mathcal{H}(\pi(\cdot|s))$.

Soft Actor Critic algorithm concurrently learns a policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2}, while in the original formulation ([28]) it learned also a value function V_ψ, recent formulations omit it since it can be substituted by the Q-function. Note that in the following, we will refer to $r(s_t, a_t)$ as r_{t+1}. Both Q-functions can be trained to minimize the soft Bellman residual, by

regressing to a single shared target:

$$J_Q(\phi_i) = \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1},d)\sim\mathcal{D}}\left[\frac{1}{2}(Q_{\phi_i}(s_t,a_t) - y(r_{t+1},s_{t+1},d))^2\right] \quad (2.32)$$

where the target y is computed using the target Q-networks:

$$y(r_{t+1},s_{t+1},d) = r_{t+1}+\gamma(1-d)\left(\min_{j=1,2} Q_{\bar{\phi}_j}(s_{t+1},\tilde{a}_{t+1}) - \alpha\log\pi_\theta(\tilde{a}_{t+1}|s_{t+1})\right) \quad (2.33)$$

where $\tilde{a}_{t+1} \sim \pi_\theta(\cdot|s_{t+1})$ and the target Q-networks ($Q_{\bar{\phi}_{1,2}}$) are obtained by polyak averaging the Q-network parameters over the course of training [43].

The policy parameters can be learned by directly minimizing the expected KL-divergence:

$$J_\pi(\theta) = \mathbb{E}_{s_t\sim\mathcal{D}}\left[\mathbb{E}_{a_t\sim\pi_\theta}\left[\alpha\log(\pi_\theta(a_t|s_t)) - (\min_{j=1,2} Q_{\phi_j}(s_t,a_t))\right]\right] \quad (2.34)$$

From a practical point of view, in order to optimize the policy (given that we use a neural network to approximate the policy) we can apply the **reparametrization trick**, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of the state, the policy parameters and independent noise:

$$\tilde{a}_\theta(s,\xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s)\odot\xi), \quad \xi\sim\mathcal{N}(0,I) \quad (2.35)$$

The $\tanh$ ensures that actions are bounded to a finite range and changes the distribution. The reparameterization trick allows us to rewrite the expectation over actions into an expectation over noise:

$$J_\pi(\theta) = \mathbb{E}_{s_t\sim\mathcal{D},\xi\sim\mathcal{N}}\left[\alpha\log(\pi_\theta(\tilde{a}_\theta(s_t,\xi)|s_t)) - (\min_{j=1,2} Q_{\phi_j}(s_t,\tilde{a}_\theta(s_t,\xi)))\right] \quad (2.36)$$

In the first paper of Soft Actor Critic ([28]), the temperature parameter is subsumed into the reward scaling factor, specifically by scaling the reward by

α^{-1}. In the followup paper ([29]) the reward scaling is not used and the α parameter is learned instead. Unfortunately, choosing the optimal temperature α is non-trivial and it needs to be tuned for each task. Instead of requiring the user to set the temperature manually, this process can be automated by formulating a different maximum entropy reinforcement learning objective, where the entropy is treated as a constraint. Through mathematical manipulation, we can derive the following loss for the α parameter:

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi_t} \left[-\alpha \log \pi_t(a_t|s_t) - \alpha \bar{\mathcal{H}} \right] \tag{2.37}$$

where $\bar{\mathcal{H}}$ is an hyperparameter. A classic heuristic for choosing $\bar{\mathcal{H}}$ is $\bar{\mathcal{H}} \propto dim(\mathcal{A})$, where $\mathcal{A}$ is the action space, e.g. $\bar{\mathcal{H}} = -dim(\mathcal{A})$ as suggested by the authors [29].

The method alternates between collecting experience from the environment with the current policy and updating the function approximators using the stochastic gradients from batches sampled from a replay buffer.

2.3 The UNIFY Framework

The UNIFY framework, proposed in [55], is a novel approach for solving constrained optimization problems by integrating traditional machine learning models with constrained optimization techniques. It outlines how complex decision-making processes, which are traditionally hard to handle due to their requirement for feasible decisions within possibly unstructured spaces, can be simplified through a strategic decomposition of the decision-making policy.

Let $x \in X$ be a vector representing observable information and the constrained policy $\pi(x, \theta) \mapsto z \in C(x)$ with z being the decisions vector. The fundamental insight lies in representing the constrained policy that should address the decision-making problem as a combination of a machine learning (ML) model denoted by $h(x, \theta)$ and a traditional constrained optimization

problem solved by a function $g(x, y)$. This decomposition is formalized as follows:

$$\pi(x, \theta) = g(x, h(x, \theta)) \tag{2.38}$$

Here, the function $g(x, y)$ is defined as a constrained optimization problem:

$$g(x, y) \equiv \arg \min_{z \in \tilde{C}(xy)} \tilde{f}(x, y, z) \tag{2.39}$$

In this formulation, y, the output of the ML model, acts as a *virtual parameter vector* influencing both the feasible set $\tilde{C}$ and the cost $\tilde{f}$ of the optimization problem, hence termed *virtual feasible set* and *virtual cost*, respectively. This approach enables the policy to navigate complex decision spaces and enforce feasibility more conveniently by leveraging any traditional constrained optimization methods (e.g., Mathematical Programming, Constraint Programming, or the Alternating Direction Method of Multipliers).

The reformulated problem becomes a bi-level optimization challenge, combining unconstrained optimization concerning θ and constrained optimization on z. Strategies for tackling this problem might involve the classical subgradient method, especially when the model h is differentiable. This process involves a "forward pass" where h is evaluated and z computed, followed by a "backward pass" where z's value is fixed, and h is differentiated with respect to parameters θ.

A remarkable facet of this reformulated policy is the introduction of virtual parameters (y): the selection of y is identified as a design decision and represents a creative challenge in applying this methodology. Given the influence of y on $\tilde{C}$ and $\tilde{f}$, there's a consequently design flexibility that can be used to partition the challenging aspects of the original problem into either h or g, by introducing or modifying elements like constraints, cost terms, or incorporating buffer schemes within g. This design flexibility empowers tackling the complexities of the original problem more adeptly, even though it does require a degree of creativity and expertise to be effectively used.

In sequential decision-making scenarios, this framework can be extended to frame problems as Markov Decision Processes (MDP), to accommodate situations where decisions unfold over time. In this setting, the MDP is characterized by a set of observable states (X), a decision space (Z), a state transition probability distribution (p_+), a cost function (f), an initial state probability distribution (p_1), and a discount factor $\gamma \in (0, 1]$, which ensures convergence in infinite sequences. At each time step (k), an observation of a new state (x_k) leads to the making of a decision (z_k), resulting in a transition to the next state (x_{k+1}) according to $p_+(x_{k+1}, x_k, z_k)$.

The training problem for sequential decision-making is then formulated as a minimization of the expected discounted cost over trajectories, represented mathematically as:

$$\begin{aligned} &\arg\min_{\theta\in\Theta} \mathbb{E}_{\tau\sim p}\left[\sum_{k=1}^{eoh} \gamma^k f(x_{k+1}, x_k, z_k)\right] \\ &\qquad z_k = g(x_k, y_k) = \arg\min_{z\in\tilde{C}(x_k,y_k)} \tilde{f}(x_k, y_k, z_k) \\ &\qquad y_k = h(x_k, \theta) \end{aligned} \tag{2.40}$$

where (θ) represents the parameters of the machine learning (ML) model, (τ) denotes a trajectory comprising sequences of states (i.e. the ML model outputs) and decisions (x_k, y_k, z_k) from the beginning to the end of a horizon (eoh); $p(\tau)$ is the probability of observing such a trajectory, calculated as the product of the probability of the initial state and subsequent state transitions along the trajectory.

This sequential formulation bridges directly to the realm of Reinforcement Learning (RL), where the function h can be interpreted as an RL policy interacting with an environment that incorporates the function g during training. This identification with reinforcement learning provides a natural pathway in order to apply such algorithms to the training problem, facilitating the incorporation of sequential decision-making challenges into the framework.

Furthermore, the sequential formulation showcases how the UNIFY framework can be extended beyond single-step decision-making problems, emphasizing the framework's versatility and adaptability to complex decision-making scenarios through the integration of machine learning and optimization strategies within an MDP framework.

2.3.1 Comparison with Other Approaches

The UNIFY framework can be compared to other notable approaches used in literature to solve the same class of problems, shedding light on its unique attributes and potential advantages.

UNIFY is more flexible than Decision Focused Learning (DFL)[64], enabling better handling of general constraints, cost functions, and sequential decision-making. Unlike DFL, which lacks virtual parameters and often assumes a fixed feasible space, UNIFY incorporates these features, potentially enhancing convergence and solution quality during training. However, UNIFY may face similar challenges as DFL in certain cases, especially when dealing with linear cost coefficients.

With regards to approaches from constrained RL [39], UNIFY is shown to generalize those methods, overcoming limitations of RL approaches that either reshape rewards to accommodate constraints or project policy parameters to achieve feasible solutions. By separating the machine learning model from decision vectors, UNIFY allows for more complex and flexible problem-solving structures, potentially yielding superior outcomes.

Finally, by incorporating machine learning predictions into optimization models, UNIFY can provide more robust and efficient solutions compared to traditional stochastic optimization techniques [45], given that the computational cost of ML model can be paid in a single offline training phase, making inference much more efficient. For further reference on UNIFY's relation to other approaches, please refer to [55].

Chapter 3

Experimental Setup

In this chapter we discuss the empirical evaluation methodologies. We start by presenting the use cases considered, namely the Energy Management System problem and the Set Multi-Cover with stochastic coverages problem; both use cases are based on [55]. We then describe the training and evaluation methodologies, with a focus on the different decisions taken w.r.t. [55]. We conclude the chapter by describing how the Hyperparameter Tuning phase was structured and performed.

3.1 Use Cases

This section explores two principal use cases presented within the UNIFY framework [55], which are pivotal to understanding its applications: an Energy Management System (EMS) and a Set Multi-cover with stochastic coverages problem. These cases not only exemplify the framework's versatility in handling diverse real-world constrained optimization problems, but also serve as the foundation for this thesis experiments. In particular, the nuanced challenges these use cases present, ranging from real-world data integration to abstract mathematical formulation, will be subjected to empirical examination under various reinforcement learning algorithms integrated into the UNIFY framework.

3.1.1 Energy Management System (EMS)

The Energy Management System (EMS) faces a highly uncertain environment due to uncontrollable fluctuations in consumption load and the integration of Renewable Energy Sources (RES). Based on real-time energy prices and forecasts regarding the availability of distributed energy resources (DERs)[1] and future consumption, the EMS must make critical decisions:

- Energy production: Determine the total amount of energy to be generated.
- Generator selection: Choose the specific generators to be used for production.
- Energy management: Decide whether to store surplus energy or sell it to the energy market.

For this problem, the available data includes historical costs, forecasts, and real-time measurements of power generation and consumption. The optimization process aims to minimize the overall power flows cost while adhering to constraints that ensure power balance and stay within power flow limits. In order to integrate this optimization problem in the UNIFY framework a previous study [14] introduced a virtual model parameter: assigning a (typically absent) cost to the storage equipment improved the performance of a traditional optimization method that relied on KKT conditions.

The problem can be addressed with two different formulations: the first considers a single-stage problem (SINGLE-STEP), where a comprehensive plan for the entire day is created in advance, encompassing 96 time units (each 15 minutes long); the second frames it as a sequential decisions problem (SEQUENTIAL), where actions are taken for each individual time unit (15 minutes), looking one step ahead, until the planning horizon is complete. Either case, the problem is composed of two macro steps: an offline phase involving

the training of a RL agent to find the optimal virtual parameter and an online parametric algorithm, implemented within a simulator, that tries to make optimal online choices, by building over the offline decisions.

Online Phase: LP model

The online step employs a myopic (greedy) heuristic to minimize costs and satisfy energy demands by manipulating energy flows between sources. This approach can be formulated as a linear programming (LP) model. A complete summary of all power flows can be seen in Table 3.1. For each stage k up to n, the decision variables x_k^g are the power flows between nodes in $g \in G$ and c^g are the associated costs. All flows must satisfy the lower and upper physical bounds $\underline{x}^g$ and $\overline{x}^g$. Index 0 refers to the input power flow to storage system and the index 1 to the respective output power flow. Hence the virtual cost associated with the input to storage system is c_k^0 . The battery charge, upper limit and efficiency are τ, Γ and ν. The EMS must satisfy the user demand at each stage k referred to as $\tilde{L}_k$.

SINGLE-STEP formulation:

$$\begin{aligned} \min_x \ & \sum_{k=1}^{n} \sum_{g \in G} c_k^g x_k^g \\ \text{s.t. } & \tilde{L}_k = \sum_{g \in G} x_k^g \\ & 0 \leq \tau_k + \eta(x_k^0 - x_k^1) \leq \Gamma \\ & \underline{x}_k^g \leq x_k^g \leq \overline{x}_k^g \end{aligned} \tag{3.1}$$

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 3.1: Description of each decision variable (i.e. power flows).

SEQUENTIAL formulation:

$$\begin{aligned} \min_x \ & \sum_{g \in G} c_k^g x_k^g \\ \text{s.t. } & \tilde{L}_k = \sum_{g \in G} x_k^g \\ & 0 \leq \tau_k + \eta(x_k^0 - x_k^1) \leq \Gamma \\ & \underline{x}_k^g \leq x_k^g \leq \overline{x}_k^g \end{aligned} \tag{3.2}$$

where the LP model is solved at every stage k.

Offline Phase: Training a RL Agent

The offline phase involves the training of a RL agent that interacts with an environment simulating the EMS scenario relying on real data. The electric load demand and photovoltaic production forecasts, upper and lower limits for generating units and the initial status of storage units are taken from a public dataset[1], that yields data from 10,000 days, from now on referred to as instances. The electricity demand hourly prices (€/MWh) have been obtained from the Italian national energy market management corporation (GME)[2]. The diesel price is taken from the Italian Ministry of Economic Development[3] and is assumed as a constant for all the time horizon (one day in our model) as assumed in literature [1] and [19].

In the SINGLE-STEP environment, every episode has a one-step length where the observations that build up the state s are the day-ahead photovoltaic

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

generation $\hat{R} \in \mathbb{R}^n$ and electric demand forecasting $\hat{L} \in \mathbb{R}^n$ and the actions are the set of virtual costs for all the stages $c_k^0 \quad \forall k \in \{1, ..., n\}$. Meanwhile, int the SEQUENTIAL environment, episodes last 96 time steps (representing units of 15 minutes each) where the state s_k keeps track of the battery charge τ_k, along with $\hat{R}$, $\hat{L}$ and a one-hot encoding of the stage k. The agent's action a_k corresponds to the virtual cost c_k^0 and the associated online optimization problem 3.2 is solved at every step. The reward function for the EMS is the negative real cost computed as follows in the two different settings:

SINGLE-STEP

$$r(S = s, A = c^0) = -\sum_{k=1}^{n} \sum_{g \in G} c_k^g x_k^g$$

SEQUENTIAL

$$r(S_k = s_k, A_k = c_k^0) = -\sum_{g \in G} c_k^g x_k^g$$

where (c_k^g) are the costs, and (x_k^g) are the decision variables (i.e. the power flows) at stage (k) for each generator (g).

3.1.2 Set Multi-cover with stochastic coverages

The Set Multi-cover problem (MSC) with stochastic coverage requirements [32] models a simplified production planning problem: given a universe N containing n elements and a collection of sets over N, it requires finding a minimum size sub-collection of the sets such that coverage requirements for each element are satisfied. The sets may correspond to product bundles, where each bundle comprises components that must be manufactured together. Similarly, the coverage requirements can be interpreted as demand constraints for individual products. The variant of the problem considered introduces additional complexities: sets have non-uniform manufacturing costs, the demands are stochastic and unknown at production time. Unsatisfied demands can be met by buying additional items, but at a higher cost. Differently from the previous case study, a synthetic dataset is generated given the number of products N, the number of sets M and the number of data points, referred to as

instances. Initially, the parameters a, c, w are randomly generated as follows:

- $a \in \mathbb{R}^{N \times M}$ is the availability matrix with density (number of 1 in the matrix) of 2%; it is generated following the guidelines of [26] where every column covers at least one row and every row is covered by at least two columns.
- $c \in \mathbb{N}^M$, $c_j \sim U(1, 100) \quad \forall j \in M$ are the set costs randomly generated with an uniform probability.
- $w \in \mathbb{N}^N$ is the penalty vector associated to the violation of the coverage requirements of each product, which is computed as follows:

$$w_i = \max_{j \in M | a_{ij}=1} c_j \cdot 10 \quad \forall i \in N \tag{3.3}$$

  ensuring that covering an element is always more convenient than receiving a penalty.

Subsequently, the dataset instances are generated, where each entry is characterized by (o, λ, d):

- $o \in \mathbb{R}$, $o \sim U(1, 10)$ is an observable variable randomly generated with an uniform probability.
- $\lambda \in \mathbb{R}^N$ are the rates of Poisson distribution assumed to have a linear relationship with o as: $\lambda_i = b_i o$ for all products $i \in N$, where $b \sim U(1, 5)$ is a random integer coefficient.
- $d \in \mathbb{R}^N$, $d_i \sim Poisson(\lambda_i) \quad \forall i \in N$ are the coverage requirements for each product, also referred to as demands.

Online Phase: LP model

Similarly to the previous use case, the online step employs a myopic (greedy) heuristic to minimize costs associated with products production and satisfy

their coverage requirements. This approach is formulated again as a linear programming (LP) model:

$$\begin{aligned} &\min \sum_{j \in M} c_j x_j \\ &\text{s.t.} \sum_{j \in M} a_{i,j} x_j \geq \hat{d}_i \quad \forall i \in N \\ &x_j \in \mathbb{Z},\ x_j \geq 0 \\ &a_{i,j} \in \{0, 1\} \end{aligned} \tag{3.4}$$

where x is the vector of decision variables and $\hat{d}$ is the vector of coverage requirements predicted by the RL agent.

Offline Phase: Training a RL Agent

The offline phase involves the training of a RL agent that interacts with an environment simulating the MSC scenario relying on the synthetic dataset previously described. In this use case, the environment has a single step duration where the state s is simply the observable variable o and the agent's actions are the predicted coverage requirements $\hat{d}$, which, in practice, belong to $\mathbb{R}^N$ so they're converted to the closest integer values. The reward is computed as:

$$\begin{aligned} &r(S = s, A = \hat{d}) = -\sum_{j \in M} c_j \hat{x}_j - \sum_{i \in N} w_i \bar{d}_i \\ &\text{where} \quad \bar{d}_i = \max\left(0, d_i - \sum_{j \in M} a_{i,j} \hat{x}_j\right) \end{aligned} \tag{3.5}$$

where $\hat{x}$ is the solution found using the predicted demands $\hat{d}$ and $\bar{d}$ is the vector of not satisfied demands computed as $d_i - \hat{d} \quad \forall i \in N$.

3.2 Training methodologies

This section highlights key differences in the experimental setup, contrasting previously adopted methods with novel approaches introduced for empirical

evaluation, while maintaining good levels of reproducibility.

3.2.1 Revision of the experimental setup

In the original work [55], the A2C algorithm was employed in the offline phase with a standard architecture without performing much hyperparameters tuning. The policy represents each possible action using a Gaussian distribution parameterized by a deep neural network with two hidden layers of 32 units each activated by the hyperbolic tangent function; a deep neural network with the same architecture is also used to learn the state value function that serves as the critic. The Adam optimizer [35] was used to update the networks' parameters, with different learning rates chosen for each problem. The EMS use case opted for a larger learning rate (0.01), while the MSC case employed a more standard rate of 0.001.

For both use cases, the data is organized into sets of instances. While EMS dataset size is static, counting 10k instances, MSC dataset was generated to comprehend 1000 instances each comprising 200 elements and 1000 sets. The EMS experiments in [55] trained the RL agent on a single instance with noise added to the forecasts $\hat{R}$, $\hat{L}$ so as to allow for a better exploration of the stochastic state space. The final evaluation of the agent was performed on 100 randomly selected instances. On the other hand, MSC experiments equally divided the dataset between training and test sets. The final evaluation has been performed by randomly selecting 50 instances to be tested. For the sake of reproducibility, all experiments were executed on a single thread even though multi-threading was available.

This thesis begins by revisiting critical decisions made regarding the dataset split for both use cases. Concerns regarding the representativeness of a single instance in the EMS use case prompted the investigation of training the agent on multiple instances, to determine whether it would help its generalization capabilities. Experiments presented in the following work investigate

the impact of using 1 (as a baseline), 5, and 30 training instances for EMS. For MSC, the dataset splitting procedure was adapted to ensure consistency across multi-threaded runs and different seeds, fostering reproducible experiments. Additionally, the training set size was increased to 70%, with the remaining 30% split divided between validation and test instances.

In both cases, 20 additional instances were selected to form a validation set. This set facilitated the evaluation of the agent's performance during training, allowing for the retention of the parameter set yielding the best results. A final evaluation was conducted on 100 different instances, acting as test set, by loading the previously saved parameter set to assess the agent's performance in a more reliable way. It is important to note that for the EMS case, during evaluation and the final test, the noise added to the observed forecasts ($\hat{R}$, $\hat{L}$) was fixed, ensuring consistent and reproducible results.

The RL algorithms investigated will be **PPO**, **TD3** and **SAC**, while **A2C** experiments with the originally proposed parameters will be used as baselines.

3.2.2 Training Procedures and Metrics

For a better understanding of the experimental evaluation, a high-level overview of the training procedures and derived metrics is provided. It is important to note that within the general RL training paradigm, which involves alternating between interaction with the environment and updates to the network parameters, a key distinction exists between **on-policy** and **off-policy** algorithms. The former 1 relies on data collected from the current policy π_θ to update network parameters; the latter 2 employs a buffer $\mathcal{D}$ to store each interaction collected during training in the form $(s_t, a_t, r_{t+1}, s_{t+1}, d)$ and samples random batches of data, potentially obtained with an old policy, to update network parameters.

Leveraging the provided pseudocode for Algorithms 1 and 2, this section defines and presents the metrics used for evaluation. To ensure a fair comparison of the agent's performance across different instances, the evaluation

metric cannot rely solely on the raw value of the sum of cumulative reward (episode reward), due to varying optimal costs per instance. The optimal cost and its optimal solution are computed by solving the LP problem stated respectively in Equation 3.4 and 3.1, under the assumption of perfect knowledge and the absence of the virtual parameter (also referred to as *oracle*). During evaluation, the agent's performance is measured using its *optimality*, computed as:

$$\frac{-optimal\_cost}{episode\_reward}$$

where values closer to 1 indicate better performance. Additionally, in the MSC scenario, the agent's *regret* is tracked, defined as

$$\_\frac{optimal\_cost - episode\_reward}{-optimal\_cost}$$

During training, the agent's best optimality on the validation set is tracked: whenever the model improves its performances, additional metrics related to data efficiency are logged. In particular, the total number of episodes collected and the total number of updates done up to that point are tracked; this is because in this work we are not merely interested in obtaining the best-performing agent on a particular problem, rather in analyzing the tradeoff between data efficiency and performances so as to better understand the scalability and sample complexity [34] properties of the UNIFY approach, when a RL algorithm is employed in the offline phase.

Algorithm 1: On-Policy Algorithm

Input: θ, ψ
$\theta \leftarrow \theta_0$ `// Initialize policy`
$\psi \leftarrow \psi_0$ `// and value function parameters`
`best_opt` $= -1$
for *each iteration* $i = 0, 1, ..., N$ **do**
  $\mathcal{D} \leftarrow \emptyset$ `// Initialize buffer to store trajectories`
  for *each environment step or until episode completion* **do**
    $a_t \sim \pi_{\theta_i}(a_t|s_t)$
    execute action a_t in the environment
    observe s_{t+1}, r_{t+1}, d
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r_{t+1}, s_{t+1}, d)\}$
  end
  compute the expected return $\mathcal{G}_t$ for each transition in $\mathcal{D}$
  for *each gradient step* **do**
    compute advantage estimates $\hat{A}_t$ (using any method of advantage estimation) based on V_{ψ_i}
    for *each batch of transitions* $B = \{(s_t, a_t, r_{t+1}, s_{t+1}, d)\} \in \mathcal{D}$ **do**
      update policy parameters according to the specific algorithm objective by one step of gradient ascent:

$$\theta_{i+1} = \theta_i + \alpha^{\theta} \nabla \hat{J}_{\pi}(\theta_i)$$

      fit value function by regression on mean-squared error:

$$\psi_{i+1} = \arg\min_{\psi} \frac{1}{|B|T} \sum_{\tau \in B} \sum_{t=0}^{T} (V_{\psi}(s_t) - \mathcal{G}_t)$$

    end
  end
  if *time to evaluate* **then**
    perform agent's evaluation on the validation set using the deterministic version of the policy $\mu_{\theta}(s_t)$ and obtain `new_opt`
    if *`new_opt` $\geq$ `best_opt`* **then**
      `best_opt` $\leftarrow$ `new_opt`
      save model's parameters, the total number of episodes collected and the total number of updates done
end
Output: θ, ψ `// Optimized parameters`

Algorithm 2: Off-Policy Algorithm

Input: $\theta, \phi_k \quad \forall k \in K$
$\theta \leftarrow \theta_0$ `// Initialize policy,`
$\phi_k \leftarrow \phi_{0_k} \quad \forall k \in K$ `// Q-functions`
$\bar{\phi}_k \leftarrow \phi_k \quad \forall k \in K$ `// and target network parameters`
$\mathcal{D} \leftarrow \emptyset$ `// Initialize an empty replay buffer`
`best_opt` $= -1$
for *each iteration* $i = 0, 1, ..., N$ **do**
  for *each environment step* **do**
    if $|\mathcal{D}| \leq$ *init_random_frames* **then**
      $a_t \sim U(a_{low}, a_{high})$
    else
      $a_t \sim \pi_{\theta_i}(a_t|s_t)$
    end
    execute action a_t in the environment
    observe s_{t+1}, r_{t+1}, d
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r_{t+1}, s_{t+1}, d)\}$
  end
  if $|\mathcal{D}| \geq$ *init_random_frames* **then**
    for *each gradient step* j **do**
      randomly sample a batch of transitions:
      $B = \{b = (s_t, a_t, r_{t+1}, s_{t+1}, d)\} \sim \mathcal{D}$
      compute targets for the Q functions $y(r_{t+1}, s_{t+1}, d)$
      accordingly to the specific algorithm
      update Q-functions by one step of gradient descent:

$$\nabla_{\phi_{i_k}} \frac{1}{|B|} \sum_{b \in B} (Q_{\phi_{i_k}}(s_t, a_t) - y(r_{t+1}, s_{t+1}, d))^2 \quad \forall k \in K$$

      if $j \mod$ *policy_delay* $= 0$ **then**
        update policy parameters according to the specific
        algorithm objective by one step of gradient ascent:

$$\theta_{i+1} = \theta_i + \alpha^\theta \nabla \hat{J}_\pi(\theta_i)$$

        update target network weights:
        $\bar{\phi}_k \leftarrow \tau \phi_k + (1 - \tau)\bar{\phi}_k \quad \forall k \in K$
      fit any other algorithm specific parameter if present
    end
  if *time to evaluate* **then**
    perform agent's evaluation on the validation set using the
    deterministic version of the policy $\mu_\theta(s_t)$ and obtain
    `new_opt`
    if *new_opt* $\geq$ *best_opt* **then**
      `best_opt` $\leftarrow$ `new_opt`
      save model's parameters, the total number of episodes
      collected and the total number of updates done
end
Output: $\theta, \phi_k \quad \forall k \in K$ `// Optimized parameters`

3.3 Hyperparameter Tuning

Hyperparameter tuning, the process of selecting the optimal configuration of parameters for a machine learning model, plays a crucial role in achieving optimal performance. However, it often presents a significant challenge due to its time-consuming and computationally intensive nature: exploring numerous parameter combinations through manual experimentation can be laborious and inefficient, while relying solely on default values might hinder the model's potential. Studies have demonstrated that a simple random search often yields superior performance compared to grid search [4]. Moreover, adopting reinforcement learning methods adds additional complexity to this process for several reasons: first, due to the non-stationarity of the RL objective, RL algorithms tend to be *sample inefficient*; second, these algorithms are very sensible to the stochasticity present in the training process and in the environment [31], yielding results with high variance across random seeds, thus the need for averaging many runs together when reporting them; third, the *sensitivity* to key hyperparameters that typically characterise RL algorithms can significantly impact their behaviour and, consequently, their performances. [33]

Real-world and complex scenarios require acknowledging that a single set of hyperparameters cannot be effectively applied across all such situations. This necessitates tuning algorithm-specific parameters for each environment, one simulating the Set Multi-Cover problem (see Section 3.1.2), and two distinct environments for the EMS use case (see Section 3.1.1) differentiating on the problem formulation (single step, Equation 3.1 and sequential, Equation 3.2). Due to this, numerous tuning phases were carried out in this work: for each RL algorithm considered (i.e. PPO, TD3 and SAC) and for each environment an independent hyperparameter tuning task was performed.

A complete description of the tuned hyperparameters can be found in Tables 3.2 and 3.3. Additionally, it is worth explaining the role of the parameter

Name	Description
actor_lr	Learning rate of policy network.
critic_lr	Learning rate of critic network.
actor_cells	Width of the policy network layers with fixed depth (2).
critic_cells	Width of the critic network layers with fixed depth (2).
schedule_lr	Whether to schedule the LRs.
frames_per_batch	Number of interactions steps with the environment.
update_rounds	Number of update rounds after each rollout. Off-policy algorithms set this equal to frames_per_batch to keep the ratio of interaction steps to gradient steps equal to 1.
batch_size	Batch size used to perform a single update.
num_envs	Number of parallel environments the agents interacts with.
obs_norm	Whether to employ observation normalization.

Table 3.2: Descriptions of hyperparameters common to all algorithms.

Name	Description
buffer_size	Size of the Experience Replay buffer.
init_random_frames	Number of random environment steps before training starts.
num_q_value_net	Number of Q value networks for clipped Q-learning.
tau	Polyak averaging coefficient for updating target networks.
prb	Whether to employ a Prioritized Experience Replay buffer.
alpha_lr	(*SAC only*) Learning rate for of temperature parameter.
policy_delay	(*TD3 only*) Number of Q-value functions updates between each policy update.

Table 3.3: Descriptions of hyperparameters specific to off-policy algorithms.

`total_frames`, i.e. the total number of interactions steps with the environment. Due to design choices in the chosen library (TorchRL [6]), it was required to set the `total_frames` rather than the number of epochs (or training iteration) during the hyperparameter tuning phase. Otherwise, tuning values of `frames_per_batch` combined with a fixed number of epochs would lead to runs of varying lengths, hindering fair comparisons. For the MSC and EMS single-step use cases, the value of `total_frames` was set to 20000, while a larger value of 480000 was used for EMS sequential. This approach essentially fixes the total number of episodes encountered by the algorithm (independently from the value of `frames_per_batch`) during training, aligning with the focus on sample efficiency due to the high computational cost of environment interactions.

As previously mentioned, the EMS experiments in this work explore learning from multiple instances. During hyperparameter tuning, it was randomly chosen to utilize either a single instance or multiple instances (specifically, five), aiming to identify a versatile hyperparameter set that performed effectively in both scenarios.

The platform Weights & Biases was employed to ease the process of tuning hyperparameters, running experiments, store data and visualize early results. All runs can be found at this project.

Chapter 4

Empirical Evaluation and Discussions

In this chapter, we discuss and analyze results of the empirical evaluation. We first discuss the results of the Hyperparameter Tuning process, and then move on to the comparative analysis of RL algorithms.

4.1 Discussion of Hyperparameter Tuning Results

The results of the hyperparameter tuning are summarized in Tables 4.2 and 4.3. It is important to acknowledge that, in the EMS scenario, trying to identify a versatile hyperparameter set that performs effectively both on the single and multiple instances, may not be suitable for all hyperparameters. For example, observation normalization (`obs_norm`) in PPO and SAC algorithms is crucial when training on multiple instances, but detrimental for single-instance training. TD3 makes an exception, meaning that it performs better as long as there is no observation normalization, but it is one of its key characteristics, as evidenced in the original study [20].

Several iterations of hyperparameter tuning were performed, progressively adjusting and narrowing down the search space. Through this iterative process, significant insights emerged. For instance, it was observed that certain

PPO-specific algorithm parameters had minimal impact on performance, and optimal values for some parameters could be readily identified, such as setting $\epsilon = 0.2$ as illustrated in Equation 2.22. Notably, all the presented use cases appeared to require minimal exploration; indeed, deviations from an entropy coefficient of 0 in PPO were found to negatively affect learning outcomes. Similarly, the SAC algorithm promptly disregarded the exploration-controlling temperature parameter α, particularly in EMS environments. Unfortunately, TD3 exhibited crucial limitations in the EMS sequential environment, attributable to the well-documented issue of action saturation [61]. Despite numerous attempts to mitigate this challenge, none provided a definitive solution; some of the approaches that were not successful have been: trying different methods for the initialization of the network's weights, adding more noise to the exploration policy and trying different actor network architecture with a decreasing number of units.

Although, some contributed to overall performance enhancements so they have been retained in the training process. For example, it was introduced a weight decay of 0.1 to the policy optimizer and reduced the size of the replay buffer that, up to that moment, was set to be equal to the number of `total_frames` for both off-policy algorithms. As highlighted in this study [61], TD3 benefits from more recent experience gathered from the environment; to this end, the authors introduce a non-uniform sampling technique that prioritizes recent memories and to approximate this approach, we included in the tuning process unusually small values for the buffer size. Indeed, a general improvement resulted from employing a replay buffer with smaller size, confirming the findings of [61]; this led to the emergent result of **Prioritized Experience Replay** (PER)[48] being preferred over the uniform sampling method, since its sampling technique prioritizes based on a proxy metric for "priority" (by default, the absolute TD-error). This provides further evidence that TD3 learning benefits from non-uniform sampling based on more recent experiences.

Hyperparameter	PPO	SAC	TD3
actor_lr	$2.5e^{-3}$	$5e^{-4}$	$5e^{-4}$
critic_lr	$5e^{-4}$	$1e^{-2}$	$5e^{-3}$
actor_cells	32	8	16
critic_cells	32	32	64
schedule_lr	False	False	False
frames_per_batch	208	32	120
update_rounds	25	32	120
batch_size	32	32	256
num_envs	2	2	8
obs_norm	True	True	True
buffer_size	-	20000	10000
init_random_frames	-	2000	2000
num_q_value	-	4	2
tau	-	$5e^{-3}$	$1e^{-3}$
prb	-	True	True
alpha_lr	-	$3e^{-4}$	-
policy_delay_up	-	-	4

Table 4.1: Best hyperparameter values for each algorithm (after tuning), MSC use case.

Hyperparameter	PPO	SAC	TD3
actor_lr	$5e^{-4}$	$5e^{-4}$	$5e^{-4}$
critic_lr	$1e^{-3}$	$1e^{-3}$	$5e^{-4}$
actor_cells	16	32	16
critic_cells	32	32	16
schedule_lr	False	False	True
frames_per_batch	1200	16	120
update_rounds	25	16	120
batch_size	32	8	128
num_envs	16	4	8
obs_norm	True	True	False
buffer_size	-	480000	30000
init_random_frames	-	4000	6000
num_q_value	-	4	3
tau	-	$1e^{-3}$	$1e^{-3}$
prb	-	True	True
alpha_lr	-	$3e^{-5}$	-
policy_delay_up	-	-	8

Table 4.2: Best hyperparameter values for each algorithm (after tuning), EMS sequential use case.

Hyperparameter	PPO	SAC	TD3
actor_lr	$2.5e^{-3}$	$2.5e^{-3}$	$1e^{-3}$
critic_lr	$2.5e^{-3}$	$5e^{-3}$	$2.5e^{-3}$
actor_cells	8	8	16
critic_cells	32	16	16
schedule_lr	False	False	False
frames_per_batch	208	120	120
update_rounds	20	120	120
batch_size	64	64	16
num_envs	16	2	2
obs_norm	True	True	False
buffer_size	-	20000	10000
init_random_frames	-	2000	2000
num_q_value	-	3	4
tau	-	$1e^{-3}$	$5e^{-3}$
prb	-	False	True
alpha_lr	-	$1.6e^{-3}$	-
policy_delay_up	-	-	2

Table 4.3: Best hyperparameter values for each algorithm (after tuning), EMS single step use case.

4.2 Comparative Analysis of RL Algorithms

The hyperparameter tuning results, summarized in Tables 4.1, 4.2 and 4.3, were subsequently used for the final comparative evaluation of the RL algorithms employed within the UNIFY framework. The analysis will be conducted by considering different values of total frames for each environment, to investigate whether longer training could yield better performances, and various random seeds, for better statistical significance [33, 31]. As extensively presented in Section 2.2 and 3.2.2, the RL algorithms A2C, PPO, TD3 ans SAC differ in the training procedure, leading to different results in terms of efficiency on the number of updates to the networks and number of episodes needed during training.

All the environments presented in this study have to solve an optimization problem at every time step, making the environment interaction an expensive computational operation. Given the objective of this algorithm comparison

is preferable to have an algorithm that is more sample efficient, namely that needs fewer episodes to learn a good policy, rather than an algorithm that is more efficient in terms of number of updates to the networks.

4.2.1 MSC

The final results on the MSC environment are presented in the following section: runs were conducted on 10 different seeds for 4 different number of total frames, specifically 20000, 40000, 60000 and 1000000. This last value was introduced to reproduce the experiments presented in the original study [55], where an A2C agent was trained for 10000 epochs performing an update every 100 episodes.

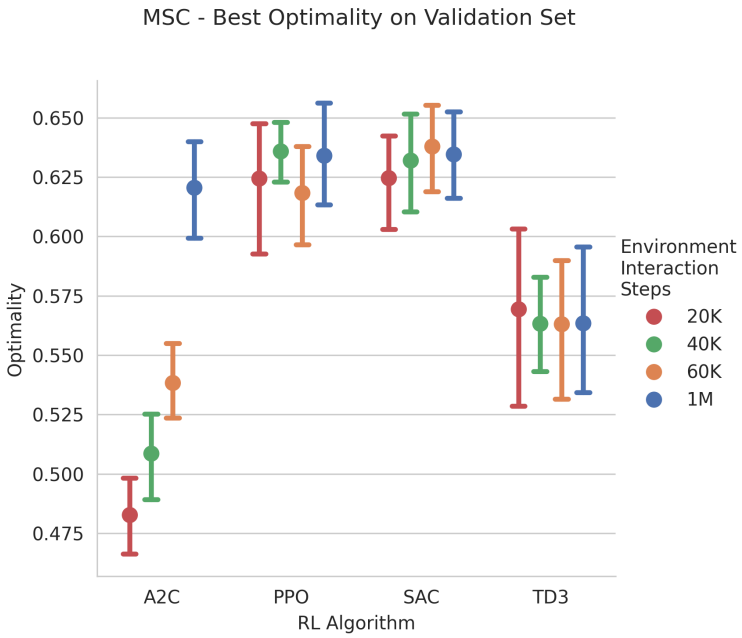

Figure 4.1: Average optimality obtained by the algorithms on 20 instances from the validation set, 95% CI. Runs are grouped according to the total number of frames.

The results clearly identify PPO and SAC as the front-runners in both performance and efficiency.They achieve validation set optimality exceeding 60% with a lower training burden, employing only 20k frames. A2C is able to

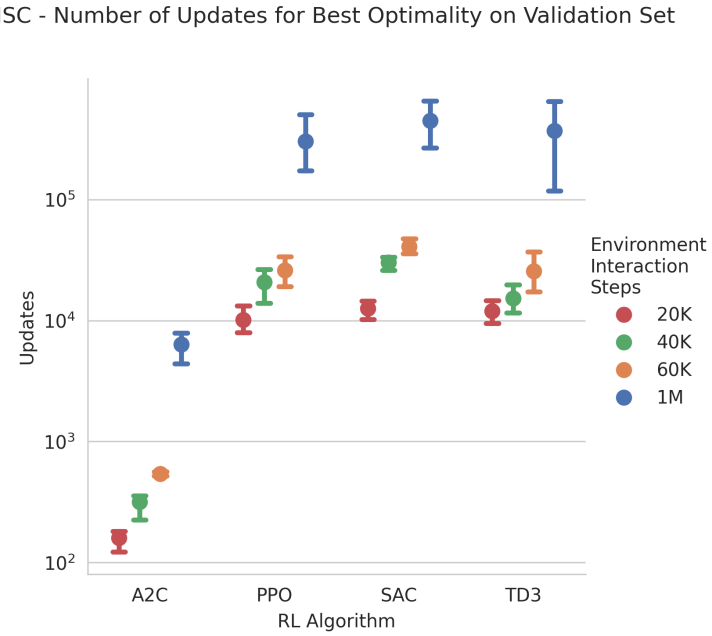

Figure 4.2: Average number of updates needed for each algorithm to obtain 4.1, 95% CI. Runs are grouped according to the total number of frames.

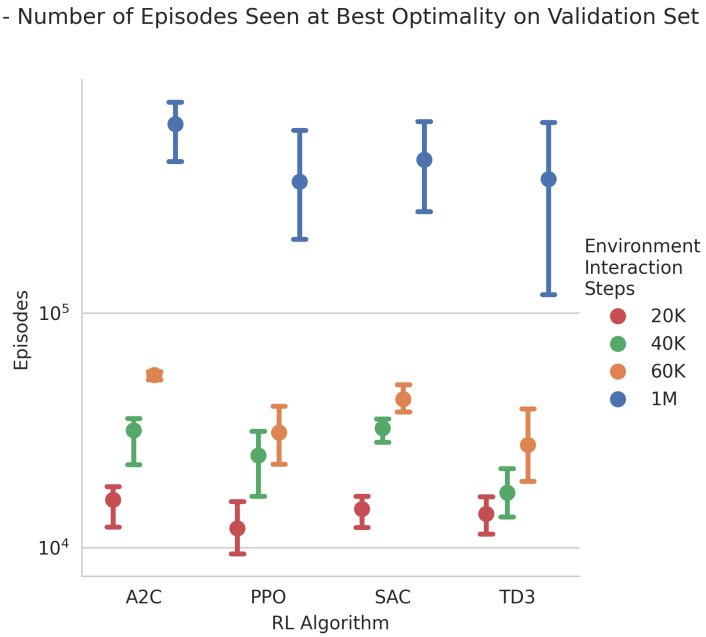

Figure 4.3: Average number of episodes needed for each algorithm to obtain 4.1, 95% CI. Runs are grouped according to the total number of frames.

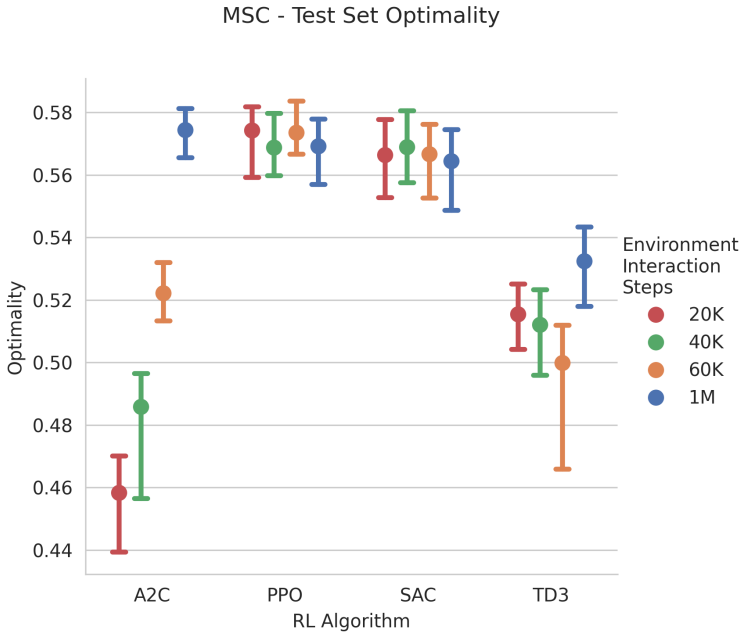

Figure 4.4: Average optimality obtained by the algorithms on 100 instances from the test set, 95% CI. Runs are grouped according to the total number of frames.

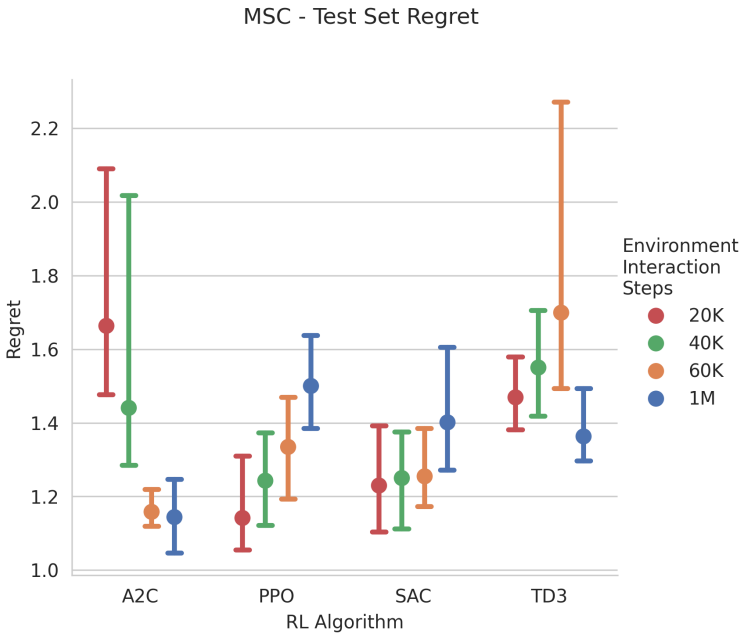

Figure 4.5: Average regret obtained by the algorithms on 100 instances from the test set, 95% CI. Runs are grouped according to the total number of frames.

achieve the similar results, but it requires to be trained for more frames (1 million). TD3 lags behind due to its limitations in exploration compounded by the single-dimensional observation space that restricts the deterministic policy's ability to explore, despite the added noise after the action is chosen.

Given the definition of the reward in this environment 3.5 where the agent is heavily penalised for predicting fewer requirements than really needed, the action that the agent chooses tends to overestimate the real demands (sometimes heavily exaggerating). Refer to Figures 4.6 for a visual example. This is further confirmed by the optimality score which, on average, isn't intuitively high, meaning that the algorithms aren't interested in perfectly learning the relationship between demands ($d_i \sim Poisson(\lambda_i)$) and the observable variable o where $\lambda_i = b_i o$. Further experiments could be conducted by reducing the penalty factor in 3.5 and investigate whether the agent would be able to learn the underlying distribution governing the future coverage requirements.

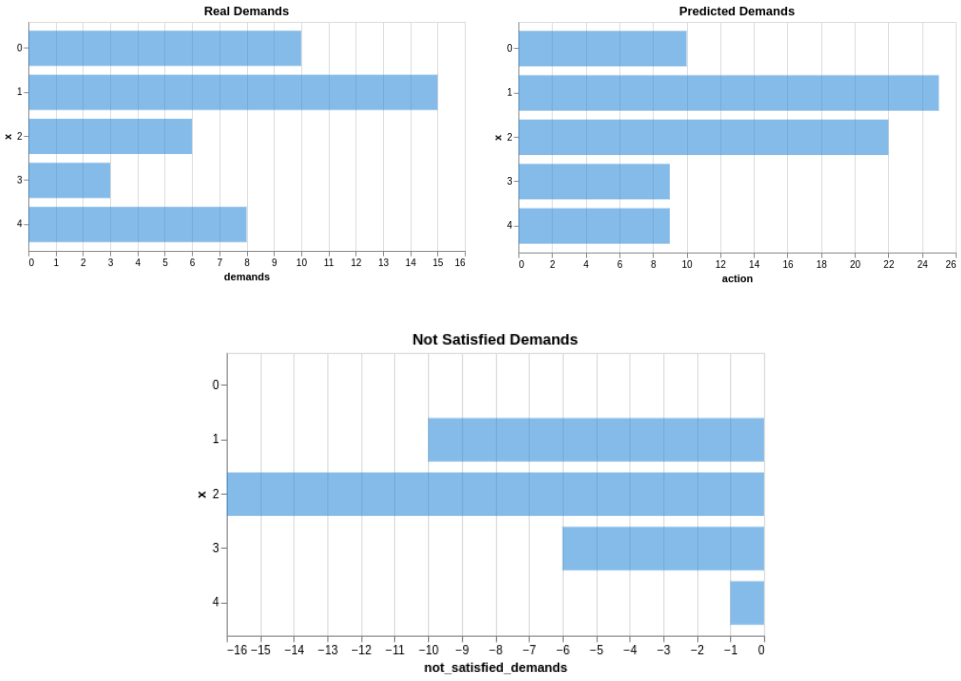

Figure 4.6: An example of SAC on a MSC instance during testing

4.2.2 EMS Sequential

The final results on the EMS Sequential environment are presented in the following section: each RL algorithm was tested on 5 different seeds and 4 different values of the total frames hyperparameter, specifically 96000, 128400, 288000 and 480000. The value 128400 was introduced to reproduce the experiments presented in the original study [55], where an A2C agent was trained for 19 epochs performing an update every 9600 episodes. In order to provide a fair comparison, two distinct single instance experiments were tried. Further investigation on the number of instances used during training is conducted by increasing its size (30 instances) in order to investigate the efficiency of the algorithms in learning from multiple instances and gain insights on the generalization benefit they could add.

The sequential case of the EMS environment is more challenging than its single-step counterpart: there are training runs that are unable to attain any improvement at all. One of them is a SAC sample that might actually be an outlier due to an unfortunate seed, as the others show that the algorithm is capable of learning without any issues. The other case presents 45 samples of the TD3 algorithm (out of 80 in total), which is the one that struggles the most in this environment, since it suffers from the action saturation problem, resulting of an agent only performing actions that are either the absolute maximum or minimum value and get stuck in a local optimal policy. This problem presents itself most frequently when training on one instance, specifically 26 cases (out of 40 in total).

The results reveal that all algorithms, except TD3, exhibit learning and generalization difficulties when trained on a single instance. TD3's performance remains stable due to the previously discussed limitations related to action saturation. The multi-instance setting (using 5 or 30 instances) significantly improves learning and generalization for most algorithms. However,

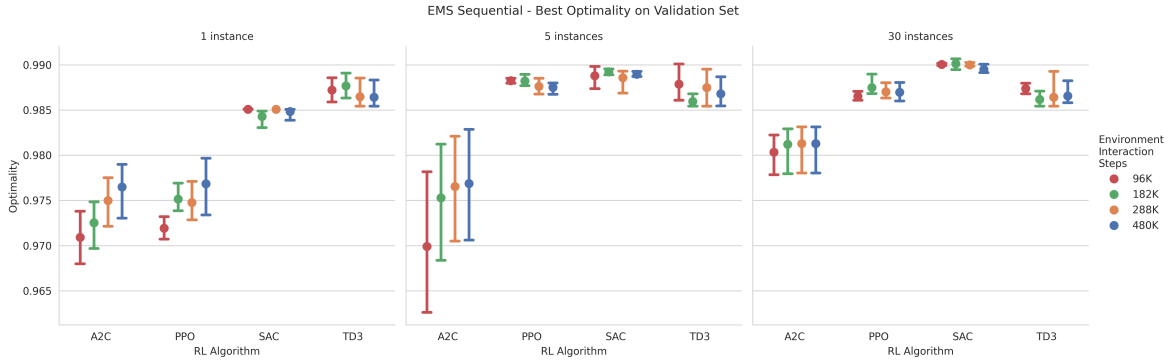

Figure 4.7: Average optimality obtained by the algorithms on 20 instances from the validation set, 95% CI. Runs are grouped according to the total number of frames.

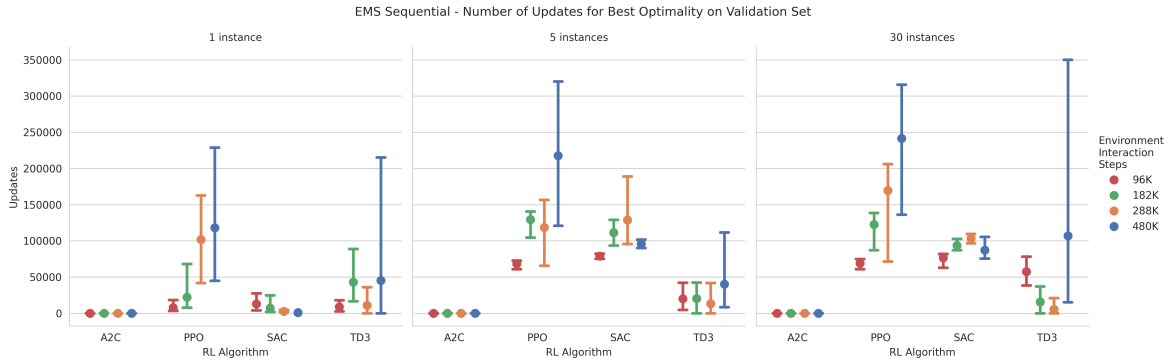

Figure 4.8: Average number of updates needed for each algorithm to obtain 4.7, 95% CI. Runs are grouped according to the total number of frames.

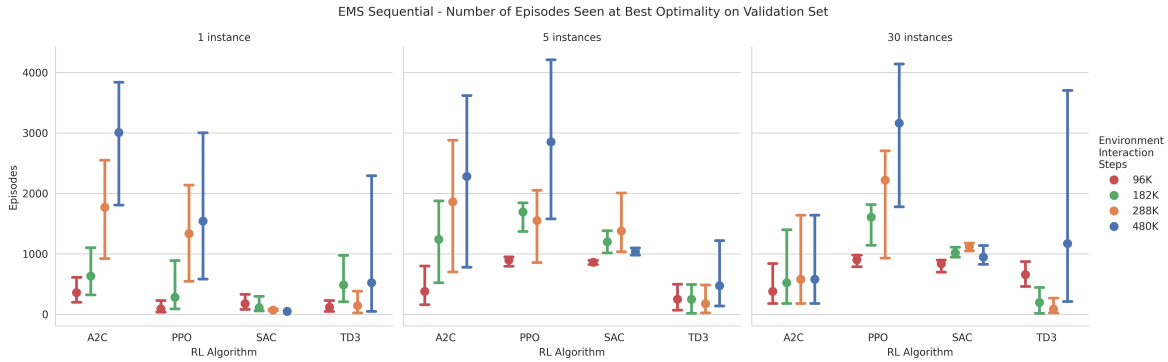

Figure 4.9: Average number of episodes needed for each algorithm to obtain 4.7, 95% CI. Runs are grouped according to the total number of frames.

PPO suffers the most when limited to a single instance, and even with improved performance in the multi-instance setting, it remains the least efficient in terms of training updates and episode exposure. Conversely, SAC emerges

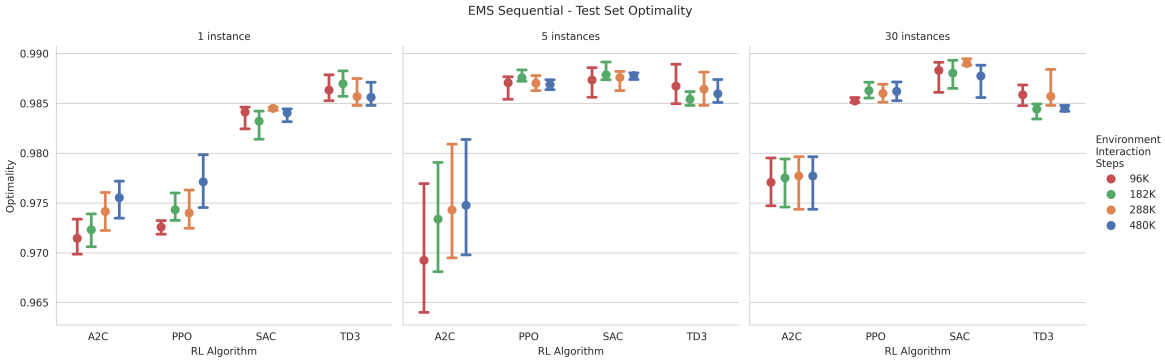

Figure 4.10: Average optimality obtained by the algorithms on 100 instances from the test set, 95% CI. Runs are grouped according to the total number of frames.

as both the most efficient and the best performing algorithm. Finally, A2C demonstrates consistently poor performance, although the multi-instance setting appears to enhance its generalization capabilities.

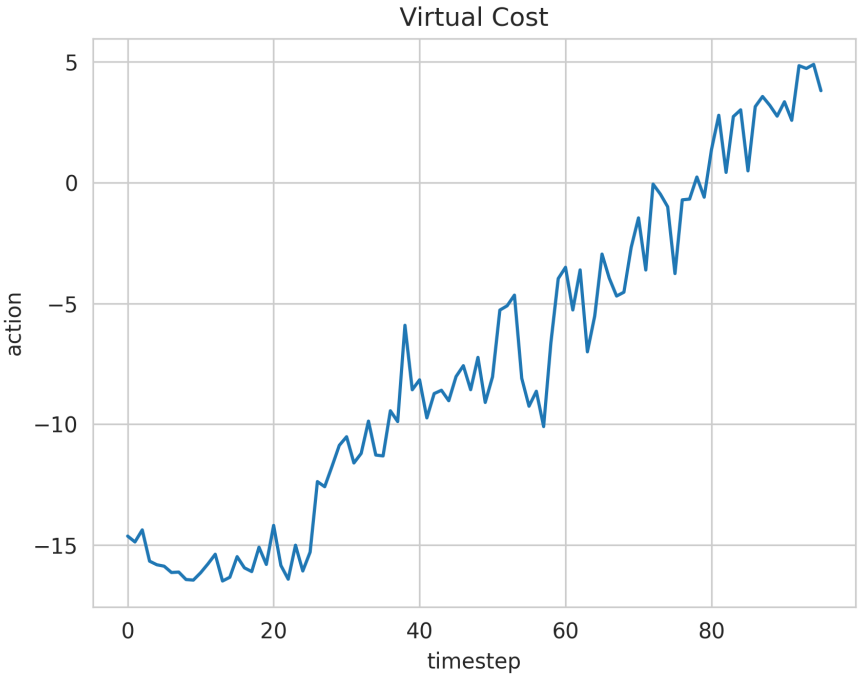

Figure 4.11: An example of the virtual cost predicted by a SAC agent during testing on a particular instance.

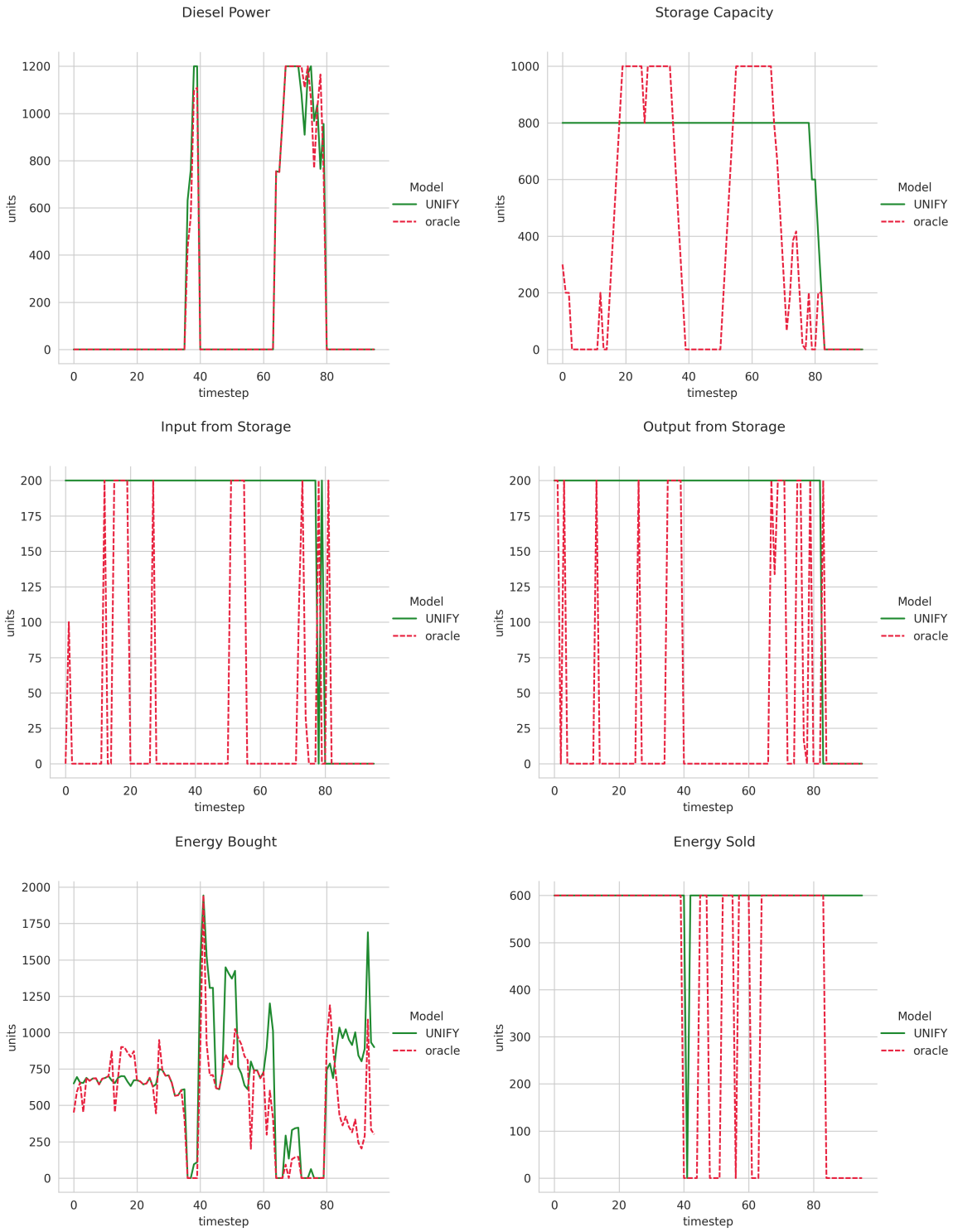

Figure 4.12: Powerflows influenced by the actions of the SAC agent on the test instance 4.11 with an optimality of 0.99. The *oracle* represents the optimal powerflows obtained by solving the LP problem under the assumption of perfect knowledge and no virtual cost.

4.2.3 EMS Single Step

The final results on the EMS Single Step environment are presented in the following section: each RL algorithm was tested on 5 different seeds and 3 different values of the total frames hyperparameter, specifically 20000, 40000 and 60000. The experiments presented in the original study [55] trained an A2C agent for 37 epochs performing an update every 100 episodes; as this results in slightly less than 40000 total frames, we can consider the 40000 frames variant as being the replication of the original experiments. The settings on the number of instances used during training is the same as in the previous case 4.2.2.

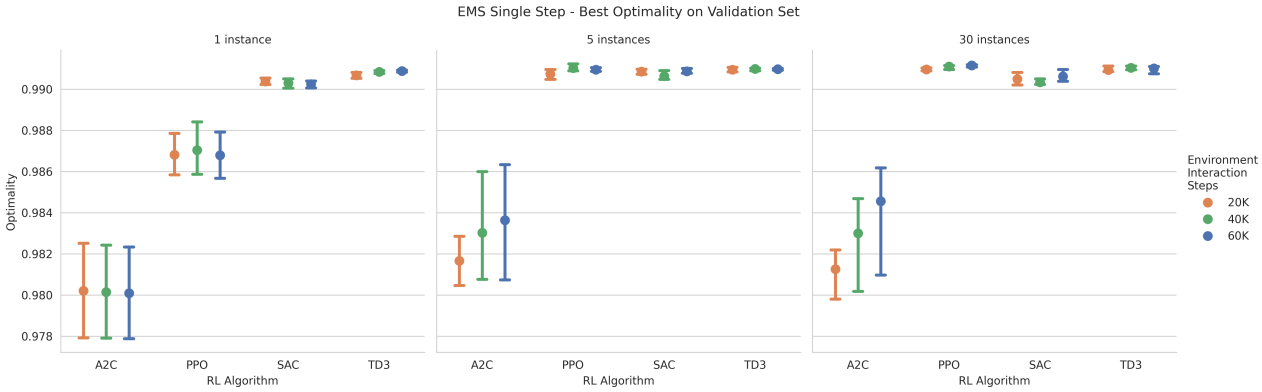

Figure 4.13: Average optimality obtained by the algorithms on 20 instances from the validation set, 95% CI. Runs are grouped according to the total number of frames.

Results show that PPO struggles again in the single instance setting, with respect to SAC and TD3 which perform well even when with limited data. exhibits consistently poor performance, although the multi-instance setting appears to improve its generalization capabilities. Interestingly, TD3 emerges as quite inefficient, especially on fewer total frames, requiring more training updates and episodes compared to others. In the multi-instance setting, SAC and PPO perform similarly, however, they excel in different aspects of efficiency: PPO minimizes the number of network updates, while SAC minimizes the total training episodes.

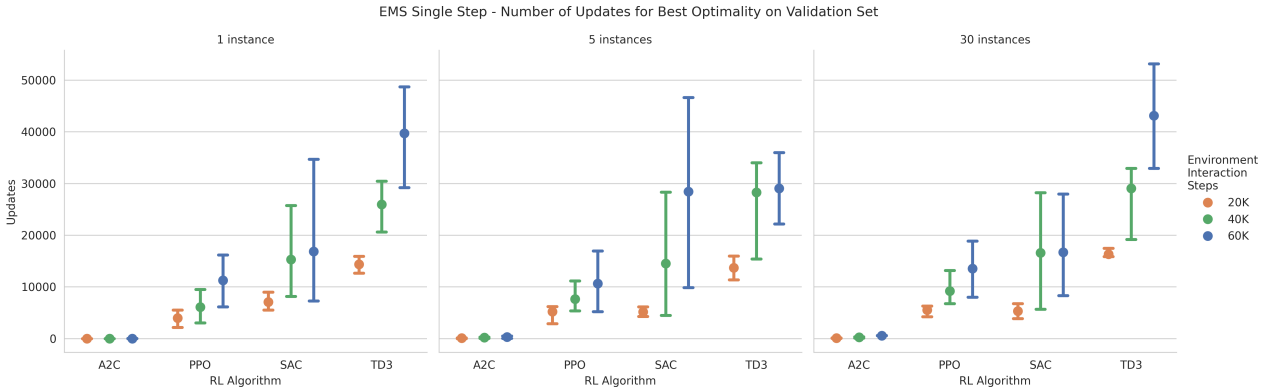

Figure 4.14: Average number of updates needed for each algorithm to obtain 4.13, 95% CI. Runs are grouped according to the total number of frames.

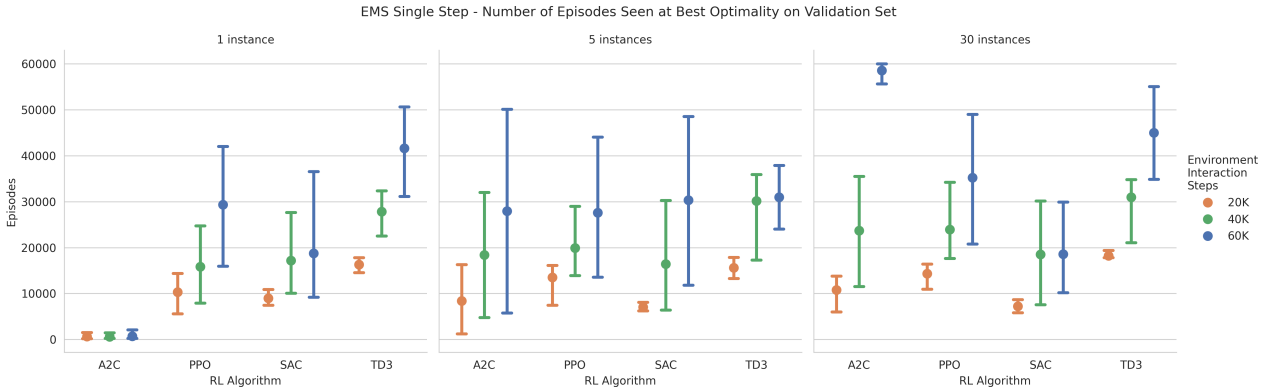

Figure 4.15: Average number of episodes needed for each algorithm to obtain 4.13, 95% CI. Runs are grouped according to the total number of frames.

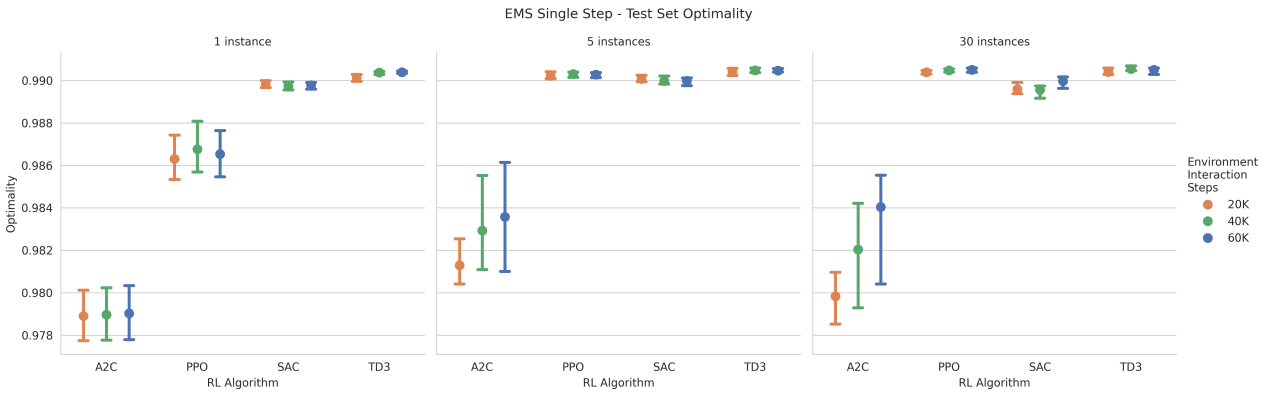

Figure 4.16: Average optimality obtained by the algorithms on 100 instances from the test set, 95% CI. Runs are grouped according to the total number of frames.

4.3 Summary

Across different environments, SAC consistently emerges as the most efficient and effective algorithm, particularly in multi-instance settings where it displays superior performance and training efficiency. PPO shows promise but struggles in the single-instance scenarios of EMS and lags in efficiency metrics. A2C generally underperforms, facing challenges in both learning and generalization, marginally improved in environments allowing for multiple instances. TD3, while stable in certain conditions, has notable exploration limitations and inefficiencies, particularly highlighted in environments that restrict deterministic policy exploration.

Chapter 5

Conclusions

This thesis presented a comprehensive study of Reinforcement Learning (RL) applications within the UNIFY framework, utilizing the Energy Management System (EMS) and the Set Multi-Cover with stochastic coverages problem to evaluate the efficiency and effectiveness of various RL algorithms. Through extensive experimentation and analysis involving the hyperparameter tuning and performance evaluation of A2C, PPO, SAC, and TD3 algorithms, several key insights and contributions have been made to both theory and practice.

5.1 Key Findings

The empirical evaluations revealed notable differences in performance and efficiency among the RL algorithms in different scenarios and settings. Among the primary findings:

- **Algorithm Performance:** SAC consistently emerged as the most effective algorithm across various settings, aided by its sample efficiency and robust handling of continuous action spaces. PPO exhibited competitive performance, particularly in multi-instance settings, highlighting its flexibility and generalization capabilities. A2C, while foundational, showed limitations in complex environments. TD3 faced challenges,

mainly in the EMS sequential environment, due to its deterministic policy gradient approach and the action saturation problem.

- **Efficiency and Generalization:** The results demonstrated that learning from multiple instances significantly improves the generalization capabilities of the RL models, as opposed to training on a single instance. This was particularly evident in environments with high variability, where SAC and PPO particularly benefited from multi-instance training settings.

- **Hyperparameter Sensitivity:** Hyperparameter tuning played a crucial role in optimizing the performance of the RL algorithms. SAC and PPO's adaptability highlighted the importance of careful hyperparameter selection, especially in varying environmental conditions and problem settings. These findings provided additional evidence that optimal hyperparameter settings are scenario-dependent, highlighting the need for tailored tuning approaches.

5.2 Contributions

The thesis contributes to the understanding of the application of RL within frameworks integrating machine learning models with optimization techniques, such as the UNIFY framework. The systematic analysis of RL algorithms in varied settings, both in terms of environment complexity and training configurations, adds valuable insights into the design and deployment of these algorithms in real-world scenarios. Moreover, the consideration of multi-instance training settings underlines the importance of diversity in training samples for enhancing the generalization of learned policies.

Additionally, the study also sheds light on the complex interaction between

algorithmic characteristics and their performance in specific contexts, offering guidance for researchers and practitioners on selecting and tuning RL algorithms for similar decision-making and optimization problems.

5.3 Future Work

Building on the findings of this thesis, several avenues for future research can be explored:

- **Exploring More Complex Environments:** Extending the application of the UNIFY framework to more diverse and complex environments also from different domains, incorporating higher dimensions of constraints and decision variables, would serve to further validate the flexibility and robustness of RL algorithms.
- **Algorithmic Enhancements:** Evaluating different RL algorithms, particularly to address specific challenges like learning environment constraints, that could simplify the online phase, while maintaining robust and efficient solutions.
- **Meta-Learning for Hyperparameter Tuning:** Investigating the use of meta-learning techniques for dynamic hyperparameter tuning and adaptation in changing environments could improve the efficiency and performance of RL algorithms.

In conclusion, this thesis underscores the potential of integrating RL with optimization techniques in solving complex decision-making problems. The UNIFY framework, augmented by the insights derived from the comparative analysis of RL algorithms, offers a promising avenue for advanced research and practical applications in various domains.

Bibliography

[1] 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.

[2] T Assavapokee, M. Realff, and J. Ammons. Min-max regret robust optimization approach on interval data uncertainty. *Journal of Optimization Theory and Applications*, 137(2):297–316, 2008.

[3] T. Assavapokee, M. J. Realff, J. C. Ammons, and I.-H. Hong. Scenario relaxation algorithm for finite scenario-based min–max regret and min–max relative regret robust optimization. *Computers & operations research*, 35(6):2093–2102, 2008.

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

[5] J. R. Birge and F. Louveaux. *Introduction to stochastic programming*. Springer Science & Business Media, 2011.

[6] 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]`.

[7] M. Branda. Stochastic programming problems with generalized integrated chance constraints. *Optimization*, 61(8):949–968, 2012.

[8] C. W. Chan, V. F. Farias, N. Bambos, and G. J. Escobar. Optimizing intensive care unit discharge decisions with patient readmissions. *Operations research*, 60(6):1323–1341, 2012.

[9] Y. Chen, Q. Guo, H. Sun, Z. Li, W. Wu, and Z. Li. A distributionally robust optimization model for unit commitment based on kullback–leibler divergence. *IEEE Transactions on Power Systems*, 33(5):5147–5160, 2018.

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

[11] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

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

[13] A. De Filippo. *Hybrid Offline/Online Methods for Optimization Under Uncertainty*, volume 349. IOS Press, 2022.

[14] A. De Filippo, M. Lombardi, M. Milano, et al. The blind men and the elephant: integrated offline/online optimization under uncertainty. In *IJCAI*, volume 2021, pages 4840–4846. International Joint Conferences on Artificial Intelligence, 2020.

[15] D. den Hertog and K. Postek. Bridging the gap between predictive and prescriptive analytics-new optimization methodology needed. *Tilburg Univ, Tilburg, The Netherlands*, 2016.

[16] S. Deo, K. Rajaram, S. Rath, U. S. Karmarkar, and M. B. Goetz. Planning for hiv screening, testing, and care at the veterans health administration. *Operations research*, 63(2):287–304, 2015.

[17] C. Duan, L. Jiang, W. Fang, and J. Liu. Data-driven affinely adjustable distributionally robust unit commitment. *IEEE Transactions on Power Systems*, 33(2):1385–1398, 2017.

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

[19] A. N. Espinosa. Dissemination document "low voltage networks models and low carbon technology profiles", 2015.

[20] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[21] V. Gabrel, C. Murat, and A. Thiele. Recent advances in robust optimization: an overview. *European journal of operational research*, 235(3):471–483, 2014.

[22] J. Gallien, A. J. Mersereau, A. Garro, A. D. Mora, and M. N. Vidal. Initial shipment decisions for new products at zara. *Operations Research*, 63(2):269–286, 2015.

[23] J. Gao, Y. Xu, J. Barreiro-Gomez, M. Ndong, M. Smyrnakis, and H. Tembine. Distributionally robust optimization. *Algorithms-Examples*:1, 2018.

[24] H. Gassmann and W. T. Ziemba. *Stochastic programming: applications in finance, energy, planning and logistics*, volume 4. World Scientific, 2013.

[25] E. Greensmith, P. L. Bartlett, and J. Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(9), 2004.

[26] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *European journal of operational research*, 101(1):81–92, 1997.

[27] Y. Guo, K. Baker, E. Dall'Anese, Z. Hu, and T. Summers. Stochastic optimal power flow based on data-driven distributionally robust optimization. In *2018 Annual American Control Conference (ACC)*, pages 3840–3846. IEEE, 2018.

[28] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[29] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[30] H. Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[31] 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.

[32] Q.-S. Hua, D. Yu, F. C. Lau, and Y. Wang. Exact algorithms for set multicover and multiset multicover problems. In *Algorithms and Computation: 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings 20*, pages 34–44. Springer, 2009.

[33] 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.

[34] S. M. Kakade. *On the sample complexity of reinforcement learning*. University of London, University College London (United Kingdom), 2003.

[35] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[36] C. Li and I. E. Grossmann. A review of stochastic programming methods for optimization of process systems under uncertainty. *Frontiers in Chemical Engineering*, 2:34, 2021.

[37] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[38] L.-J. Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[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] M. Mehrotra, M. Dawande, S. Gavirneni, M. Demirci, and S. Tayur. Or practice—production planning with patterns: a problem from processed food manufacturing. *Operations research*, 59(2):267–282, 2011.

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[42] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[43] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[44] C. Ning and F. You. Optimization under uncertainty in the era of big data and deep learning: when machine learning meets mathematical programming. *Computers & Chemical Engineering*, 125:434–448, 2019.

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

[46] W. B. Powell. *Reinforcement Learning and Stochastic Optimization: A unified framework for sequential decisions*. John Wiley & Sons, 2022.

[47] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[48] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[49] 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.

[50] 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.

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

[52] S. Sen, S Gass, and M Fu. Stochastic programming. *Encyclopedia of Operations Research and Management Science*:1486–1497, 2013.

[53] A. Shapiro and A. Philpott. A tutorial on stochastic programming. *Manuscript. Available at www2. isye. gatech. edu/ashapiro/publications. html*, 17, 2007.

[54] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

[55] 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.

[56] J. E. Smith and R. L. Winkler. The optimizer's curse: skepticism and postdecision surprise in decision analysis. *Management Science*, 52(3):311–322, 2006.

[57] J. C. Spall. *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons, 2005.

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

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

[60] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[61] C. Wang, Y. Wu, Q. Vuong, and K. Ross. Striving for simplicity and performance in off-policy drl: output normalization and non-uniform sampling. In *International Conference on Machine Learning*, pages 10070–10080. PMLR, 2020.

[62] C. Wang, R. Gao, F. Qiu, J. Wang, and L. Xin. Risk-based distributionally robust optimal power flow with dynamic line rating. *IEEE Transactions on Power Systems*, 33(6):6074–6086, 2018.

[63] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[64] 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.

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

[66] R. J. Williams and J. Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

[67] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *Advances in neural information processing systems*, 30, 2017.

[68] P. Xiong, P. Jirutitijaroen, and C. Singh. A distributionally robust optimization model for unit commitment considering uncertain wind power generation. *IEEE Transactions on Power Systems*, 32(1):39–49, 2016.

[69] C. Zhao and Y. Guan. Data-driven stochastic unit commitment for integrating wind generation. *IEEE Transactions on Power Systems*, 31(4):2587–2596, 2015.

[70] C. Zhao and Y. Guan. Unified stochastic and robust unit commitment. *IEEE Transactions on Power Systems*, 28(3):3353–3361, 2013.

[71] B. D. Ziebart, A. L. Maas, J. A. Bagnell, A. K. Dey, et al. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

Acknowledgements

I would like to express my sincere gratitude to Mattia and Allegra for their extensive support in carrying out this thesis project. I'd also like to thank my advisor Prof. Michele Lombardi, for his invaluable guidance and insightful feedback, which significantly shaped the direction of this research.