Università Degli Studi Di Verona

Dipartimento di Informatica

Doctoral Thesis - XXXV Cycle

# Safe Deep Reinforcement Learning: Enhancing the Reliability of Intelligent Systems

*Author:*
Davide Corsi

*Advisor:*
Prof. Alessandro Farinelli

April 2023

# Abstract

In the last few years, the impressive success of deep reinforcement learning (DRL) agents in a wide variety of applications has led to the adoption of these systems in safety-critical contexts (e.g., autonomous driving, robotics, and medical applications), where expensive hardware and human safety can be involved. In such contexts, an intelligent learning agent must adhere to certain requirements that go beyond the simple accomplishment of the task and typically include constraints on the agent's behavior. Against this background, this thesis proposes a set of training and validation methodologies that constitute a unified pipeline to generate safe and reliable DRL agents. In the first part of this dissertation, we focus on the problem of constrained DRL, leaving the challenging problem of the formal verification of deep neural networks for the second part of this work.

As humans, in our growing process, the help of a mentor is crucial to learn effective strategies to solve a problem while a learning process driven only by a trial-and-error approach usually leads to unsafe and inefficient solutions. Similarly, a pure end-to-end deep reinforcement learning approach often results in suboptimal policies, which typically translates into unpredictable, and thus unreliable, behaviors. Following this intuition, we propose to impose a set of constraints into the DRL loop to guide the training process. These requirements, which typically encode domain expert knowledge, can be seen as suggestions that the agent should follow but is allowed to sometimes ignore if useful to maximize the reward signal. A foundational requirement for our work is finding a proper strategy to define and formally encode these constraints (which we refer to as *rules*). In this thesis, we propose to exploit a formal language inherited from the software engineering community: scenario-based programming (SBP). For the actual training, we rely on the constrained reinforcement learning paradigm, proposing an extended version of the Lagrangian PPO algorithm.

Recalling the parallelism with human beings, before being authorized to perform safety-critical operations, we must obtain a certification (e.g., a license to drive a car or a degree to perform medical operations). In the second part of this dissertation, we apply this concept in a deep reinforcement learning context, where the intelligent agents are controlled by artificial neural networks. In particular, we propose to perform a model selection phase after the training to find models that formally respect some given safety requirements before the deployment. However, DNNs have long been considered unpredictable black boxes and thus unsuitable for safety-critical contexts. Against this background, we build upon the emerging field of formal verification for neural networks to extend state-of-the-art approaches to robotic decision-making contexts. We propose "ProVe", a verification tool for decision-making DNNs that quantifies the probability of violating the specified requirements. In the last chapter of this thesis, we provide a complete case study on a popular robotic problem: "mapless navigation". Here, we show a concrete example of the application of our pipeline, starting from the definition of the requirements to the training and the final formal verification phase, to finally obtain a provably safe and effective agent.

# CONTENTS

# Chapter 1

# INTRODUCTION

Artificial Intelligence and Robotics. The idea of creating artificial life has fascinated humankind for decades. At the same time, there is something in these topics that threatens and scares us, and this can be seen in movies, books, video games, and all the media that can come to our minds. It is not surprising that Isaac Asimov, one of the most world-famous authors of science fiction, had his first concern with building strict laws to guarantee the safety of his intelligent machines. Today, of course, we are far from the scenarios narrated in his stories. However, given the impressive achievement of AI systems in the last decade and the huge interest in this technology expressed by society, industry, and academics, it is of paramount importance to take care of the safety of these systems. For example, consider the recent and ongoing effort by the European Union (EU) commission to clearly define an AI strategy for the EU, where safety and reliability are crucial keywords [EU, 2022].

In this thesis, we focus on the problem of generating intelligent and reliable systems, focusing on a specific technique: *Deep Reinforcement Learning* (DRL). Among the other machine-learning approaches, DRL is probably the closest to how human beings learn. An agent interacts with the environment through a trial-and-error process, learning to solve tasks from its mistakes and driven only by a high-level objective represented with a reward signal. Although this learning approach has shown groundbreaking results in a wide variety of tasks, ranging from robotics, games, and autonomous driving, it suffers from a substantial limitation: the generated agents are typically unpredictable and depending on the context, this assumes different relevance. This thesis focuses on safety-critical tasks, where expensive hardware and human safety are typically involved. In these contexts, unpredictability often means dangerous and potentially unsafe. In this dissertation, we analyze this problem from two different perspectives, safe training and validation, contributing to the same primary objective of our work: *generating safe, reliable, and trustworthy intelligent systems.*

As humans, we learn from our experiences, making mistakes and learning from them. At the same time, the help of a mentor, like a parent, a coach, or a teacher, saves us from wasting time

following the wrong path and driving us to the proper (and safe) way of solving a problem. The first part of this thesis focuses on the application of this concept in a machine-learning context. Our intuition is that a pure trial-and-error approach (i.e., end-to-end) can be slow and often generates unpredictable systems, producing agents that potentially act in a completely different manner from those expected. This behavior causes distrust and thus limits the application of these systems in real-world scenarios. In contrast, we aim to generate intelligent agents that respect criteria beyond simply completing their assigned task, such as safety, human-friendly behaviors, management of optional subtasks, and more. This objective drives the first half of the thesis and, together with the second part, builds a complete pipeline for the generation of safe intelligent agents.

Recalling the parallelism with humans, we now introduce the second part of our work. Before being authorized to conduct dangerous (i.e., safety-critical) operations, we must obtain different kinds of certifications, for example, a license before driving or a degree to perform medical operations. Following this common-sense practice applied to human beings, in this thesis, we propose to further test the intelligent systems after the training with a formal approach that goes beyond a simple empyrical validation process. The second part of the thesis focuses on the analysis and verification of artificial deep neural networks (i.e., DNNs), which are the basic building blocks for modern DRL approaches (and ML in general). These tools are often considered and treated as unpredictable black boxes hence severely hindering the reliability of DRL systems. Our goal is to provide a formal (or, in some cases, probabilistic) certification to guarantee that an intelligent agent respects some safety and behavioral requirements.

Finally, in the third part of this dissertation, we dive into a real-world case study. We face the robotic problem of mapless navigation, i.e., the task of reaching a target in an environment without any information about the map, relying only on local observations. We show that our methodologies can be combined to generate intelligent agents that respect a set of given requirements and to provide formal safety guarantees on their behavior before the deployment on the actual platform in a real-world context.

## 1.1 CONTRIBUTIONS

As introduced in the first part of this chapter, the primary goal of this thesis is to provide a set of methodologies to generate safe, predictable, and trustworthy agents through a Safe Deep Reinforcement Learning process. Our contributions, and consequentially this dissertation, can be subdivided into two parts, *training* and *validation*. Before summarizing the challenges, methodologies, and achievements that constitute our work, we remark that the respect of these additional requirements is a crucial problem when applied to safety-critical tasks, and our arguments are weaker in classical problems (e.g., Atari games, Chess, Go). For example, a chess player's final goal is to win the game, and binding the strategies exploited to achieve this objective is not a crucial requirement. Quite the opposite, we are typically interested in finding new

approaches to overcome the human players.

### SAFE TRAINING AND CONSTRAINED OPTIMIZATION

Including additional requirements into the training loop is a complex challenge, both from a theoretical and practical point of view. An opinion shared by a significant part of the community is that "reward is enough" [Silver et al., 2021]. Although this is theoretically true, it has been shown in different works [Corsi et al., 2022; Roy et al., 2021; Yerushalmi et al., 2022; Kamran et al., 2022] that it becomes increasingly difficult for the optimization algorithm to guarantee the respect of all the additional behaviors as the number of requirements increases. In what follows, we provide a typical reward function that encodes some requirements in addition to the reward function:

$$R(x) = r(x) + \sum_n \alpha_n j_n(x) \tag{1.1}$$

here $r(x)$ is the reward for the primary objective, $j_n(x)$ is the penalty for the $n^{\text{th}}$ additional behavior, and $\alpha_n$ is a multiplier to balance the magnitude of the penalties. In a recent work from Roy et al. [2021], for example, the authors show that through only a reward engineering process, when enforcing more than three behavioral requirements in the reward function, even state-of-the-art algorithms struggle to find a good policy and (or) respect the desired behaviors. Another example is from Yerushalmi et al. [2022]; in this paper, the authors propose adding a reward penalty to discourage unwanted behavior. However, their results show that the agent tends to ignore the requirements if the penalty is too small and, on the other hand, finds only low-performing policies if the penalties cover the primary reward function. Crucially, the authors highlight that finding the correct balance for the penalties is a challenging problem, and the complexity of this task grows fast with the number of requirements. In the work of Kamran et al. [2022], the authors highlight another crucial limitation of the reward engineering approaches: the limited interpretability of the hyperparameter $\alpha_n$. This parameter appears difficult to understand and can assume unbounded values, and this makes it even harder for the users to find an effective value for the penalty. Moreover, they suggest that defining a threshold for a constrained optimization problem is easier, given the direct connection between a constraint and its bound. Finally, in our recent work Corsi et al. [2022], we further validate the aforementioned results by applying these techniques to a real-world robotic problem.

A possible solution is to encode the constraints in the training or inference phase via hard-coded shielding methods [Srinivasan et al., 2020; Thananjeyan et al., 2021]. Although these approaches guarantee the respect of the requirements by construction, it is clearly in contrast to the essence of reinforcement learning, the idea of allowing intelligent agents to learn original strategies to solve a problem. In the dissertation of Marchesini [2022] and in the work of Garcıa and Fernández [2015], the authors show that limiting the exploration phase leads to a significant drop in the quality of the generated policies, and the application of hardcoded shielding is a clear example of this unwanted behavior. In the case study work from Kamran et al. [2022], the

authors show that restricting the search space often produces over-conservative behavior that can potentially lead to a stalemate of the system (i.e., the producing agent prefers a no-action policy). However, there are also less drastic shielding solutions. In the work of Simão et al. [2021], the authors propose a state abstraction approach to focus the search space limitations only on the cost-relevant configurations. Although this approach may mitigate the aforementioned problems, it still inherits some limitations of a shielding approach and assumes specific prior knowledge about the state distribution in the environment.

Against this background, we propose to look at the problem from a different perspective. Our first argument is that, given the numerical optimization nature of DRL, providing a "zero-cost" result (i.e., a policy that never violates the requirements) is practically impossible. Following the intuition of Achiam et al. [2017], we propose to look at our DRL process as a constrained optimization problem in the following form:

$$\begin{aligned} \max_{\vec{x}} \quad & f(\vec{x}) \\ \text{s.t.} \quad & g(\vec{x}) < c \end{aligned} \tag{1.2}$$

where $f(\vec{x})$ is the primary goal and $g(\vec{x}) < c$ encodes some additioal requirements. While with an analytic solution, the respect of the constraints is guaranteed, a numerical optimization process (e.g., *gradient ascent* typically used in DRL) only ensures to reach a local minimum, which usually translates into an approximation of the optimal policies. It is clear that, with a suboptimal policy, it is impossible to ensure complete respect for the given requirements. As support to our claims, we refer to the work of Ray et al. [2019], where the authors show that even with a state-of-the-art algorithm, the given constraints can not always be respected through a DRL approach for complex tasks.

Our second argument is that, in most cases, we are not even looking for a "zero-cost" result. As discussed in the introduction of this thesis, our goal is to guide the training process by providing suggestions into the training process and providing formal guarantees only before the deployment. We thus propose subdividing the requirements into *"soft constraints"* and *"hard constraints"*. Assuming that, as discussed before, securing the latter directly into the training phase is not possible (without significantly hindering the performance [Marchesini, 2022; Kamran et al., 2022]), we focus only on the "soft constraints" for the training while leaving the consideration of the hard requirements before the deployment (i.e., the formal verification). We discuss this last topic in the second part of this dissertation.

Following these intuitions, we propose to rely on the constrained optimization setup applying a relaxation of the requirements. Crucially, our approach has a fundamental characteristic: it does not look for a "zero-cost" result. Although this can appear as a limitation, we believe that (recalling that the problem of the formal guarantees will be addressed in the second part of the thesis) this is a strength of our method. First, this result can be obtained by a numerical optimization method, which is necessary for DRL. Moreover, the agent is allowed to sometimes ignore the requirements to maximize the final reward. In more detail, we propose to rewrite

the requirements as an indicator function that assumes binary values at each time step. This function assumes the value 0 if the requirement is respected and 1 otherwise. An important advantage of this formulation is that, at the end of an episode, we estimate the probability of violating the given requirement by normalizing the sum of these values by the number of steps. In this setup, the objective is to keep this probability below a given threshold, limiting these behaviors rather than reducing them to zero. To provide an intuition, suppose we train an agent to drive a car for a motorsport race. A behavioral requirement can be the ideal trajectory that, in most cases, should be followed. However, there are some cases where the agent should be allowed to ignore this suggestion, for example, to complete an overtake or in difficult rainy conditions. In this context, sometimes ignoring requirements is not only admissible but desirable, and our approach allows us to control the frequency of these *behavioral divergences*.

A fundamental prerequisite for applying our method is to have a simple methodology to define the requirements and calculate the probability of violating them. There are many solutions to encode a function that represents a constraint (usually called *cost function* in literature). For example, in the work of Ray et al. [2019], they propose a state-based encoding; if the current state of the agent is a "forbidden state" (e.g., a specific area of the environment), the cost is increased by a unit. This approach can be useful in some cases but does not suit well for behavioral requirements, where not only the state is relevant but also the state-action combinations or even sequence of them. Among all the various options, in our work, we chose to use "Scenario Based Programming" (SBP) as a formal language to describe and encode the requirements as rules for the system. A crucial feature of our method is that the computation of the rules does not affect the exploration. The evaluation runs parallel to the system, counting the number of violations without limiting the exploration power or the agent's behavior (we refer to Part I for an exhaustive discussion on this topic).

Our experiments show promising results, proving that it is possible to inject a set of requirements into the training loop, generating agents that overall respect the desired behavior, violating them in a limited number of cases and only if useful to maximize the reward. We evaluated our approach in different environments, in particular in Chap. 8 we provide a detailed analysis of a real-world robotic problem, exploiting the proposed methodologies both in simulation and on the real platform. A final important remark is on the work of Yang et al. [2022a]. The authors highlight another important challenge: *the respect of the safety requirement not only at convergence but also during the learning process*. By its nature, DRL requires a trial and error process in order to learn the best actions to perform (or to avoid), and this is in contrast with the idea of guaranteeing safety inside the training loop. However, the authors of the paper propose to exploit a simulated (or controlled) environment where the agent is allowed to violate the constraints in order to learn a safe policy. This "guide policy" can then be exploited as a starting point for the actual training in the real safety-critical scenario. Crucially, this approach is not in contrast with our methodologies, quite the opposite; our entire pipeline can be used in the first pre-training phase.

### VALIDATION AND FORMAL GUARANTEES

Until now, we have focused on the problem of injecting behavioral requirements into the training loop. However, guaranteeing some "hard constraints" is crucial in safety-critical contexts. Recalling the example of the motorsport race, following the ideal trajectory is a behavioral requirement that, in some cases, can (or even should) be violated. In contrast, some behaviors, such as moving directly into an obstacle, can not be tolerated, regardless of the context. These strict requirements can be viewed as *safety properties*, encoded through input-output relations, that describe some behaviors our agents must respect. We propose to evaluate the generated models after the training to provide *formal* guarantees before the deployment in the real world (e.g., on a robotic platform).

The intuition is that, even if it is impossible to generate "safe by construction" models directly from the deep reinforcement learning loop, it is possible to minimize the frequency of violations of the requirements. Consequently, by analyzing a set of trained models that empirically shows zero-cost behavior, it is unlikely that no one formally guarantees to respect these safety properties. In this thesis, we propose to perform a model selection phase after the training and before the deployment to filter only the provably safe models.

Providing formal guarantees about the behavior of a DNN-based controller is challenging and, also for this reason, it is common to look at the DNNs as unpredictable black boxes. However, in the last few years, much work has been done in this direction to demystify and formally analyze these complex and non-linear function approximators (i.e., deep neural networks). In particular, the pioneering work of Katz et al. [2017] has brought attention to a new line of research, the formal verification of neural networks. For our model selection phase, we rely on this concept, formalizing the properties to verify as input-output relations, following the encoding proposed by Liu et al. [2019]. Intuitively, a verification framework for DNNs tries to find an input point that yields a violation of the safety requirement, analyzing the continuous and multi-dimensional domain of the neural network function. If this violation point is found, the verification process returns `SAT`, and the model is considered unsafe. Conversely, if this point does not exist, the model can be considered safe. Although this approach has proven effective in a wide variety of tasks [Ehlers, 2017; Tjeng et al., 2018; Katz et al., 2019; Wang et al., 2021], in the DRL context, which is the focus of this dissertation, the standard approaches have shown some substantial limitations that we address throughout the second part of the thesis. First, the state-of-the-art approaches typically return a binary answer: *safe* or *unsafe*; however, in a DRL context where the search space can be huge, it is often unlikely to obtain a model which is 100% safe. In contrast, we might be interested in finding a safe model with a given probabilistic tolerance. In this thesis, we propose *"ProVe"*, a novel algorithm that exploits the standard verifiers as a backend to quantify the number of violations, replacing the decision problem with a counting one. Finally, we face the problem of the properties definition and encoding in a DRL context, where the evaluation must be performed on sequential invocations of the policy to represent the interactions with the environment.

In conclusion, we consider these two previously described processes (i.e., constrained training and verification) as part of a single framework that generates safe, predictable, and trustworthy agents. In Chap. 8, we extensively evaluated our pipeline on a real-world robotic problem showing that, through our approach, it is possible to provide provably safe and high-performing policies.

SUMMARY OF THE CONTRIBUTIONS

We summarize our contribution in the following points:

- We introduce a novel formalism to encode behavioral requirements by exploiting a formal language: Scenario Based Modeling (SBP).
- We propose a novel method to inject safety and behavioral rules into a DRL loop, encoding them through SBP. Our algorithm exploits the lagrangian relaxation of a constrained optimization problem, presenting a novel constrained DRL algorithm.
- We propose a learning technique that combines evolutionary approaches and DRL to increase the safety of the generated agents.
- We propose a tool for the formal verification of neural networks designed to verify sequential decision-making problems that typically characterize a DRL agent. Furthermore, we present a methodology to quantify the number of violations, estimating the probability of violating the given properties.
- We show how to encode and verify multi-step (or time-dependent) safety properties for deep reinforcement learning in robotic contexts, where the system's safety depends on a sequence of actions and interactions with the environment.
- We present a unified framework that combines our training and verification methods to generate safe and reliable systems (Fig. 1.1 presents a graphical overview of the pipeline).
- We validate the previously presented methodologies on a real-world robotic problem, obtaining agents that provably respect the desired requirements.
- We provide a public repository with the source code of the algorithms presented in this work and the instructions to replicate our experimental results: http://github.com/d-corsi.

## 1.2 THESIS OUTLINE

The thesis begins with a review chapter on the foundational concept of the thesis. We provide a critical analysis of the deep reinforcement learning algorithms, focusing on the evolution of the

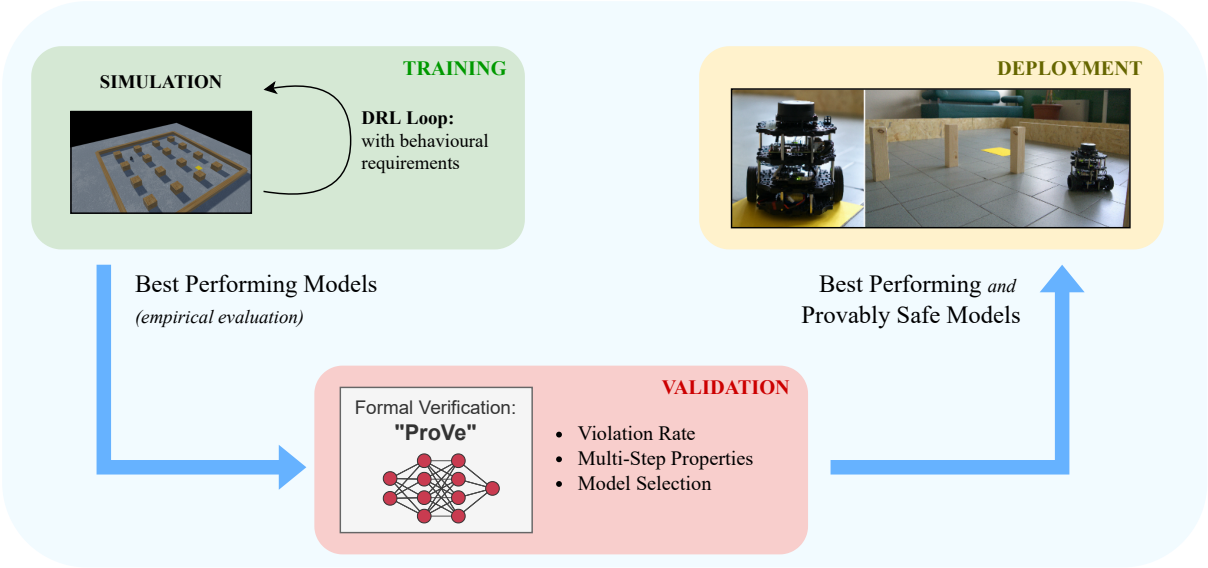



Figure 1.1: An illustrative overview of our pipeline that integrates the key contributions of the thesis; following these methodologies, we obtain reliable models (i.e., DNNs) before the deployment.

approaches and the motivations behind their development. We then present the safety problem for intelligent systems, with a particular focus on robotic problems. The rest of the thesis is divided into three parts containing the main contributions of our work.

**Part I**, *Safe Training for Deep Reinforcement Learning*, addresses the problem of generating safe and reliable intelligent agents. This part discusses different approaches and techniques for training the agents.

- In Chap. 3 (*"Preliminaries"*), we introduce the fundamental information and crucial concepts necessary to understand the rest of this part (e.g., constrained optimization, genetic algorithms and constrained MDP).
- In Chap. 4 (*"Constrained Reinforcement Learning"*), we introduce one of the main contributions of this dissertation, A novel approach to injecting prior knowledge, typically human expertise, into the training loop. Our novel method combines techniques from the "formal languages" community (i.e., scenario-based programming) with the recent trend of constrained deep reinforcement learning. This line of work is related to the publication of Corsi et al. [2022].

**Part II**, *Formal Verification for Deep Reinforcement Learning*, addresses the problem of providing formal safety guarantees to the trained agents. This analysis is generally performed after the training phase when the neural networks should be validated before the deployment.

- In Chap. 5 (*"The Verification Problem"*), we introduce the definition and motivations for the formal verification of deep neural networks, focusing on the challenges and the families of approaches to solving this complex task.

- In Chap. 6 (*"Verification of Decisions (ProVe)"*), we present ProVe, a novel pipeline for verifying deep neural networks, designed for the analysis of deep reinforcement learning agents. ProVe is a high-level framework that allows the counting of possible violations to safety requirements and can rely on state-of-the-art verification tools as a backend. The idea is to obtain a metric to rank the safety of an agent, overcoming the limits of the binary answer returned by standard methods (i.e., *safe* or *unsafe*). This line of work is related to the publication of Corsi et al. [2021] and appears in the case study articles from Pore et al. [2021] and Marchesini et al. [2021a].

- In Chap. 7 (*"Time-Dependent Properties"*), we propose a framework for the formal verification of multi-step safety properties. In the context of deep reinforcement learning, the standard formulation for the safety requirements as input-output relations is a substantial limitation. An intelligent agent is supposed to interact with the environment with a long-term horizon, performing sequences of actions. To overcome this limitation, we propose extending the approach from Amir et al. [2021] to a robotic context, where the trajectories selected by the agents are the crucial behavior to analyze. This line of work is related to the publications of Marchesini et al. [2021a] and Marzari et al. [2022].

**Part III**, *Application to Robotics*, presents a complete case study where we apply all the techniques and methodologies presented throughout the thesis. In particular, we exploit the training approach presented in Chap. 4, followed by a model-selection phase based on the verification frameworks presented in Chap. 6 and Chap. 7. In this chapter, we analyze the popular robotic problem of mapless navigation, applied on the research platform TurtleBot3, both in simulation and on the actual robot. Following our pipeline, we show that obtaining safe, reliable, and predictable intelligent agents is possible without compromising the performance. This line of work is related to the publication of Amir et al. [2022]. Finally, in Chap. 9, *"Conclusion and Future Direction"* we summarize the results we presented in the thesis, discussing the future research directions and opportunities.

## 1.3 PUBLICATIONS

Most of the contributions presented in this thesis have been published in top-level international conferences, e.g., IJCAI, ICLR, AAAI, TACAS, UAI, and IROS; some works are submitted and currently under review. The aforementioned publications are presented in the following list, where **(*)** denotes an equal contribution between the authors:

- The #DNN-Verification problem: Counting Unsafe Inputs for Deep Neural Networks
  L. Marzari*, **D. Corsi***, F. Cicalese, A. Farinelli
  *International Joint Conference on Artificial Intelligence (IJCAI) 2023.*

- Verifying Learning-Based Robotic Navigation Systems
  G. Amir*, **D. Corsi***, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, G. Katz
  *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2023.*

- Curriculum Learning for Safe Mapless Navigation
  L. Marzari, **D. Corsi**, E. Marchesini, A. Farinelli
  *ACM/SIGAPP Symposium on Applied Computing (ACM SAC), 2022.*

- Exploring Safer Behaviors for Deep Reinforcement Learning
  E. Marchesini*, **D. Corsi***, A. Farinelli
  *Association for the Advancement of Artificial Intelligence (AAAI), 2022.*

- Formal verification of Neural Networks for Safety-Critical Tasks in Deep Reinforcement Learning
  **D. Corsi**, E. Marchesini, A. Farinelli
  *Conference on Uncertainty in Artificial Intelligence (UAI), 2021.*

- Benchmarking Safe Deep Reinforcement Learning in Aquatic Navigation
  **D. Corsi***, E. Marchesini*, A. Farinelli
  *International Conference on Intelligent Robots and Systems (IROS), 2021.*

- Safe Reinforcement Learning Using Formal Verification for Tissue Retraction in Autonomous Robotic-Assisted Surgery.
  A. Pore*, **D. Corsi***, E. Marchesini*, D. Dall'Alba, A. Casals, A. Farinelli, P. Fiorini
  *International Conference on Intelligent Robots and Systems (IROS), 2021.*

- Formal Verification for Safe Deep Reinforcement Learning in Trajectory Generation.
  **D. Corsi**, E. Marchesini, A. Farinelli, P. Fiorini
  *International Conference on Robotic Computing (IRC), 2020.*

- Genetic Soft Updates for Policy Evolution in Deep Reinforcement Learning.
  E. Marchesini, **D. Corsi**, A. Farinelli
  *International Conference on Learning Representations (ICLR), 2020.*

- Double Deep Q-Network for Trajectory Generation of a Commercial 7DOF Redundant Manipulator.
  E. Marchesini, **D. Corsi**, A. Benfatti, A. Farinelli, P. Fiorini
  *International Conference on Robotic Computing (IRC), 2019.*

Following the list of the papers submitted and under the review process:

- Constrained Reinforcement Learning and Formal Verification for Safe Colonoscopy Navigation
  **D. Corsi***, L. Marzari*, A. Pore*, A. Farinelli, A. Casals, P. Fiorini, D. Dall'Alba
  *The paper is currently submitted and under the review process, available at: https://arxiv.org/abs/2206.09603, 2023.*

- Constrained Reinforcement Learning for Robotics via Scenario-Based Programming
  **D. Corsi***, R. Yerushalmi*, G. Amir, A. Farinelli, D. Harel, G. Katz
  *The paper is currently submitted and under the review process, available at: https://arxiv.org/abs/2303.03207, 2023.*

*- last update May 11, 2023-*

# Chapter 2

# FOUNDATIONS AND CRITICAL ANALYSIS

In this chapter, we define the main concepts and methodologies that constitute the foundations of the thesis. We begin defining the concepts of neural networks and machine learning; then, we dive into the deep reinforcement learning problem, presenting the state-of-the-art methods and showing the evolution of these algorithms. We describe the problems faced in the thesis, focusing on robotics tasks where safety is a critical requirement. Finally, we discuss the problem of simulation, an important component of reinforcement learning, and the sim-to-real transfer, which is a crucial challenge in this field.

Before introducing the problem of *Deep Reinforcement Learning*, which is the crucial component of this dissertation, we must define two preliminary concepts: *Machine Learning* and *Neural Networks*. Machine Learning (ML) is an emerging branch of computer science and Artificial Intelligence (AI) that has recently gained popularity. Given the extraordinary results researcher, engineers, and enthusiasts, were able to achieve using this kind of approaches, ranging from computer vision [Simonyan and Zisserman, 2014], autonomous driving [Bojarski et al., 2016], games playing [Mnih et al., 2013] and many others [Deng and Liu, 2018; Nandkumar et al., 2021], this research area has rapidly grown. The idea behind a ML approach is to iteratively refine and improve the system's accuracy based on algorithms and data. Machine learning and Deep Learning (DL) are often used as synonymous; however, they are different concepts or, more precisely, DL is a subfield of ML. The crucial difference between these two concepts is that DL exploits a powerful tool to learn, which is the foundational concept of this thesis: *Deep Neural Network (DNN).*

## 2.1 NEURAL NETWORKS

Providing a definition for *deep neural network* is a challenging task, not because a DNN is difficult to define, quite the opposite, but more because different users can attribute different meanings

to this powerful tool. Focusing on the definitions that matter for this thesis, we see a neural network from a *mathematical* point of view, defining it as a complex non-linear function [Hornik et al., 1989]. Alternatively, from a *computer scientist* perspective, a DNN is a computational directed graph consisting of multiple layers [LeCun et al., 2015]. However, from an *high-level* point of view, a DNN is just a black box, and it is not a trivialization. An entire branch of the DL community is devoted to developing novel algorithms that are agnostic to the low-level structure of the networks [Mnih et al., 2013; Schulman et al., 2017]. Moreover, different tools have been developed that allow researchers and engineers to focus on applying these methodologies [Pore et al., 2022; Marchesini and Farinelli, 2020] without the need to learn the low-level theory behind DNNs, obtaining exciting scientific results on the possible applications. From a *biological* point of view, it is an attempt o reproduce a simplified version of the neurons and connections which constitute the human brain [Sutton and Barto, 2018]. Although very far from the purpose of this dissertation, there is also a *philosophical* perspective, which sees DNNs as a potential tool for replicating human consciousness, trying to study all the consequences that this could lead to [Binns, 2018].

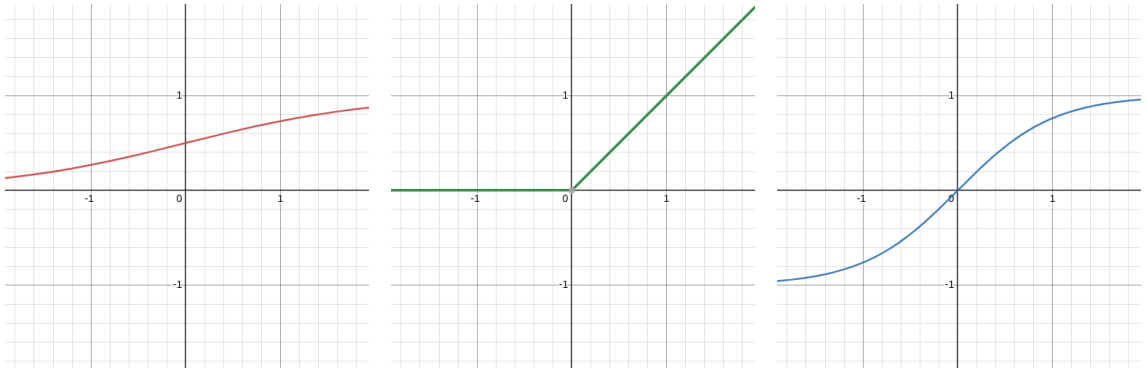
In the following discussion, we focus on the Multi-Layer Perceptron (MLP), sometimes referred to as *feed-forward DNN*. MLP is the most common architecture, constituted only of forward connections (refer to Fig. 2.2 for a high-level intuition of the structure) [Hornik et al., 1989]. There are, however, more sophisticated implementations of the networks, such as Recurrent Neural Networks (RNN) that include backward connections [Lai et al., 2015], or Convolutional Neural Networks (CNN), specifically designed for image recognition [Simonyan and Zisserman, 2014].

Formally, a neural network $f_\theta(x)$ is a function, defined over the parameters $\theta$, that maps an input $x$ (typically a vector) to an output $y$. The computation is performed through a sequence of operations for each layer of the DNN:

$$a^{(l)} = g(W^{(l)}a^{(l-1)} + b^{(l)}) \tag{2.1}$$

where (i) $a^{(l)-1}$ is the input vector to the layer $l$; (ii) the weights matrix $W$ and the biases $b$ are the parameters of the function (conventionally denoted together as $\theta = [W^{(0)}, ...W^{(n)}, b^{(0)}, ...b^{(n)}]$); and (iii) $g$ is the activation function, that provides the non-linearity; notice that $a^{(0)}$ is the input and $a^{(n)}$ is the output. The activation function is a fundamental component for DNNs, introduced in the previous definition with $g$. These support functions introduce the non-linearity in DNN, which is crucial to guarantee the expressiveness of the function [Hornik et al., 1989]. There are different types of functions; Fig. 2.1 provides an overview of three examples of common activations. In this work, we focus on the Rectified Linear Unit (ReLU), defined as $\text{ReLu}(x) = \max(0, x)$. Our choice is motivated by different reasons: (i) it has been shown that it is effective, yet simple to compute [Banerjee et al., 2019]; (ii) it is easy to implement and differentiate for the learning process; (iii) it is piece-wise, which is a crucial property for the formal verification[1].

[1]This topic will be discussed in Part II.



(a) sigmoid (b) ReLU (c) tanh

Figure 2.1: A graphical overview of different common activation functions, *sigmoid*, *ReLU* and *hyperbolic tangent (tanh)*.

Fig. 2.2 shows a toy example to understand better how a neural network computes the output. Suppose the input vector $x = [v_1^1, v_1^2]^T$ with $v_1^1 = 2$ and $v_1^2 = 3$, the second (weighted sum) layer computes the values $v_2^1 = v_1^1 \cdot 2 + v_1^2 \cdot 5 = 20$, and $v_2^2 = v_1^1 \cdot -4 + v_1^2 \cdot 1 = -7$. Then, $v_2^1$ and $v_2^2$ pass through the activation function ReLu, resulting in $v_3^1 = 20$ and $v_3^2 = 0$. Finally, the output is computed as $v_4^1 = v_3^1 \cdot 2 + v_3^2 \cdot -1 = 40$.

The crucial characteristic that enables DNNs to solve the large variety of problems we briefly mentioned before is that the parameters $\theta$ of the function can be learned. Although there are several approaches to obtain the best values, such as the evolutionary method we elaborate in the following chapters, the standard and most common is Gradient Descent (GD) [Ruder, 2016]. To provide intuition on how GD works, we first define a *loss function* ($L$) as a way to measure the distance between the output of the network $y$ and the desired output $\hat{y}$. A crucial requirement for the loss function is differentiability over the parameters of the network. For example, a valid loss function is Mean Squared Error(MSE):

$$L(f_\theta(x), \hat{y}) = \frac{1}{2}\sum_{i=1}^{n}(y^i - \hat{y}^i)^2 \tag{2.2}$$

GD consists in iteratively updating the parameters $\theta$ by stepping in the opposite direction of the gradient to minimize the function $L$. More formally, the $k$-th step of the algorithm is:

$$\theta_{k+1} = \theta_k - \alpha \nabla_{\theta_k} L(f_\theta(x), \hat{y}) \tag{2.3}$$

Fig. 2.3 graphically shows the general idea. There are different variants and improvements over GD, for example, the *momentum* [Sutton and Barto, 2018], *Adam* [Kingma and Ba, 2014] and more, that share the general structure and the main intuition. It is important to remark, and

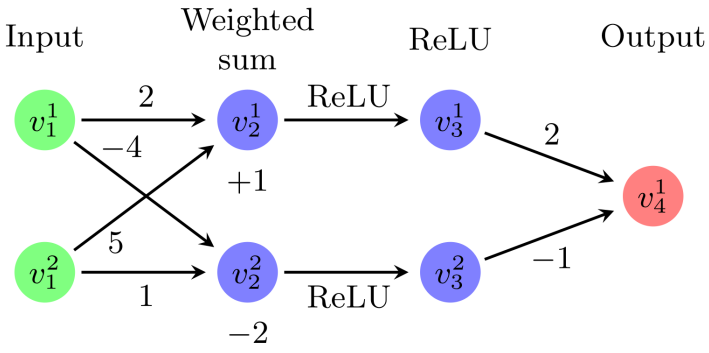



Figure 2.2: An illustrative example of a neural network with two inputs, two hidden layers, and one output. The biases $b^{(l)}$ are not indicated in this simplified version.

this is particularly relevant in our work, that the loss function is not necessarily something we aim to minimize but, in some cases, is the objective we want to maximize. In this last case, the procedure is called *gradient ascent*, and the loss function is denoted with the more general name of *objective function*.

Deep Learning is commonly subdivided into three macro-families of approaches: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. While diving too deeply into these concepts is out of scope for this thesis, it is important to remark on the main differences between these approaches and explain why we focus on the latter.

- *supervised learning:* the key characteristic is the use of labeled datasets. Intuitively, the learning algorithm adjusts the weights of the DNN to learn how to predict the correct output label given an input. The idea is that the generated models can generalize to unseen and new data, learning the correct pattern from the input dataset [Hornik et al., 1989].

- *unsupervised learning:* the objective is to analyze and cluster unlabeled data, finding patterns and subdividing data into different groups. Although an entire branch of DL works in this exciting direction, the objective of these approaches is different from the focus of this thesis. We refer interested readers to more specific papers [Berry et al., 2019].

Supervised learning approaches are widely adopted; they have shown excellent performance and are considered state of the art in image recognition [Jiang et al., 2022], sentiment analysis [Birjali et al., 2021], music generation [Briot and Pachet, 2020], and many more. However supervised learning suffers from a serious limitation intrinsic to the method, it requires a dataset. Obtaining a dataset can be challenging, sometimes even impossible, and obtaining fair data, not biased by human choices, is an additional challenge. To face all these problems, the emerging

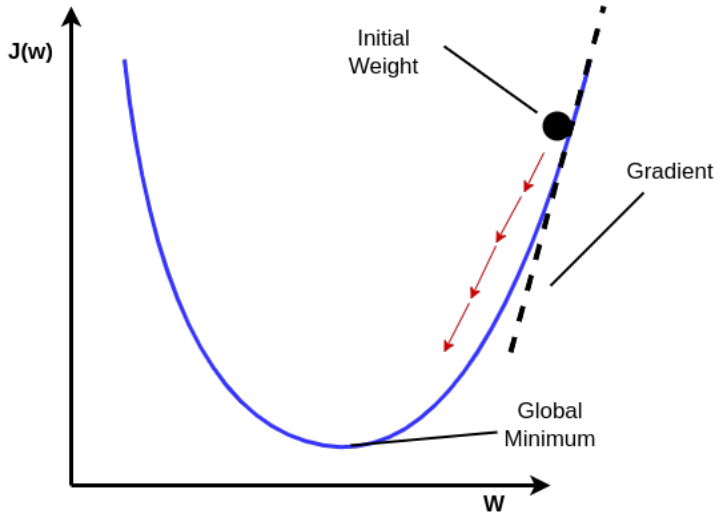



Figure 2.3: High-leve overview of the gradient descent approach. The parameters of the DNN are moved towards the local minimum to minimize the loss function.

field of Deep Reinforcement Learning (DRL) is gaining popularity. Driven by the successes demonstrated in an impressive variety of contexts, ranging from game-playing [Mnih et al., 2013], robotics [Marchesini and Farinelli, 2020] and being applied in many more specific fields such as medical problems [Zhu et al., 2020; Attanasio et al., 2020], industrial applications [Hu et al., 2020] and financial operations control [Meng and Khushi, 2019].

**Deep and Classical Reinforcement learning** First, we must clarify that reinforcement learning is not necessarily deep (i.e., based on DNNs). Indeed, one of the most popular RL approaches is *Q-Learning*, which is based on tables (notice that other table-based RL approaches exist, such as SARSA). The objective of *Q-Learning* is to update a table (usually called *q-table*), randomly initialized [Sutton and Barto, 2018], with values that represent the desirability value of action $a$ in a given state $s$. This value reflects the cumulative long-term reward the agent expected to obtain by choosing the action $a$ and following the optimal policy. Tab. 2.1 shows a toy example to provide the intuition of a q-table structure.

The q-learning algorithm relies on the *Bellman Equation* to update the values inside the table. Here, we provide only an intuition, referring the interested readers to Sutton and Barto [2018] for more details:

$$Q(s,a) = r(s,a) + \gamma \cdot \max_{a} Q(s',a') \tag{2.4}$$

where $\langle s, a \rangle$ is the state-action pair we are updating (i.e., time $t$), and $\langle s', a' \rangle$ is the state-action pair of next time step (i.e., time $t+1$).

| | Action 1 | Action 2 | Action 3 | Action 4 |
|---|---|---|---|---|
| State 1 | 10 | 7 | 3 | -2 |
| State 2 | 4 | 3 | 1 | 4 |
| State 3 | -12 | -4 | 3 | 12 |

Table 2.1: An illustrative example of a q-table structure. For each state-action pair, the table stores a scalar value representing the Value Q-reflects of the couple.

## 2.2 DEEP REINFORCEMENT LEARNING

Deep reinforcement learning [Sutton and Barto, 2018] is a unique paradigm and setting for training DNNs. In DRL, an *agent*, is trained to learn a *policy* $\pi$, which maps each possible environment *state* $s$ (i.e., the current observation of the agent) to an *action* $a$. The policy can have different interpretations among various learning algorithms. For example, in some cases, $\pi$ represents a probability distribution over the action space. In contrast, in others, it encodes a function that estimates a *desirability score* over all the future actions from a state $s$. During training, at each discrete time-step $t \in 0, 1, 2...$, a *reward* $r_t$ is presented to the agent, based on the action $a_t$ it performed at time-step $t$.

Fig. 2.4 shows an overview of the standard DRL loop. Different DRL training algorithms leverage the reward differently to optimize the DNN-agent's parameters during training. The uniqueness of the DRL paradigm lies in the training process, which is aimed at generating a DNN that computes a mapping $\pi$ that maximizes the *expected cumulative discounted reward* $R_t = \mathbb{E}\big[\sum_t \gamma^t \cdot r_t\big]$. The *discount factor*, $\gamma \in \big[0, 1\big]$, is a hyperparameter that controls the influence that past decisions have on the total expected reward.

A natural way to encode a DRL task, which can be described as a sequential decision-making problem, is through a Markov Decision Process (MDP). An MDP is a popular mathematical framework, defined as a tuple of 4 elements $\langle S, A, R, T \rangle$, where:

- $S$ is the set of states for the environment.
- $A$ is the set of actions the agent can perform.
- $R$ is the reward function that maps the pair state-action to a real value. In some setups, R is defined only on the current state; in others, the reward function also considers the next state and actions.
- $T$ is the transition model. In some cases, it can be deterministic, and in others, stochastic. In the latter, $T$ is the probability of reaching a successor state by selecting a specific action in a given state.

Providing a taxonomy for the DRL algorithms can be complex, and sometimes different views may emerge even from the most influential resources. Here we simplify this subdivision,

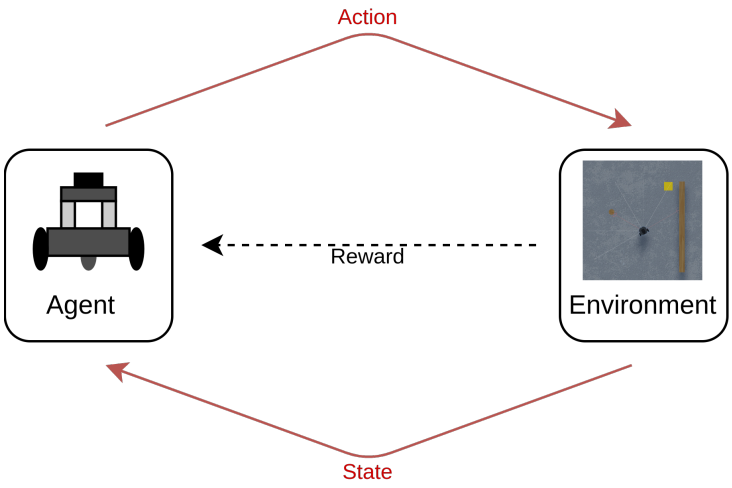



Figure 2.4: The standard reinforcement learning training loop. An agent interacts with the environment, collecting samples from the state and accumulating a reward signal.

accepting some simplifications to clarify the design choices we made in our work. In this thesis, we focus only on *model-free* approaches, where the transition model $T$ is unknown to the algorithm. However, another family of RL approaches, *model-based*, considers the transition model in the training loop [Moerland et al., 2020]. Focusing on model-free approaches, we classify DRL training algorithms into three categories: *value-based*, *policy-gradient*, and *actor-critic* [Amir et al., 2022].

### VALUE BASED

Although q-learning has shown interesting results, using a table as a data structure presents several limitations. Among others, the most relevant are certainly *scalability* and *continuous action spaces*. For complex problems, the number of states, and consequently the size of the table, grow exponentially and become intractable. Moreover, it is possible to store only a finite number of states in a table, excluding all the tasks based on continuous observations. To overcome all these problems, in *Deep Q-Network* (DQN), Mnih et al. [2013] propose the use of a neural network as a function approximator for the q-table. The basic idea behind the algorithm is the same as Q-Learning. However, a DNN can manage continuous input, and the network size does not have to be proportional to the number of possible states, enhancing the scalability. Listing 2.1 presents a naïve implementation of deep Q-network in python-like pseudocode.

DQN is often considered the first value-based approach for DRL. However, despite its enormous success, it presents some limitations and has been improved in several ways. Among the

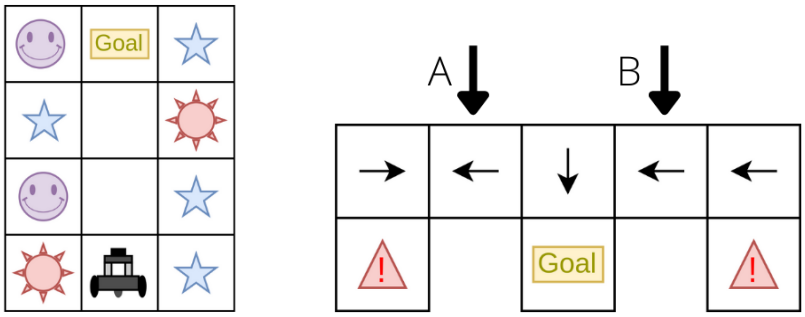
most notable improvements and follow-up works, we mention *Double DQN* [Van Hasselt et al., 2016], *Dueling DQN* [Wang et al., 2016] and *Rainbow* [Hessel et al., 2018]. However, even though the state-of-the-art algorithms can successfully find an excellent approximation of the ideal value function, as explained in the introduction of this section, the main objective of a DRL algorithm is to find the best policy (i.e., the best strategy to solve a task). In the value-based world, the policy $\pi$ is trivially *"follow the action with the highest value"*, a simple approach that introduces different limitations. In the next section, we discuss some of these limitations, introducing the policy gradient class of algorithms as a possible solution.

```
# Function that implements the actual learning phase of the training,
# we assume to have primitive functions for the execution of neural
# networks (e.g., predict and fit)
def update_network( memory_buffer ):
    mini_batch = sample_from_buffer( memory_buffer )
    for (state, action, next_state, reward) in mini_batch:
        target = network_predict( state )
        q_values = network_predict( next_state )
        max_q = max( q_values )
        # Actual implementation of the Bellman Equation
        target[action] = reward + gamma * max_q
        fit_network( state, target )

# Main function that implements the standard reinforcement learning loop
def main():
    # In this loop, we assume to have the initial state for the first iteration
    while True:
        action = select_action( state )
        # 'interact_with_environment' is an ideal function that,
        # given an action compute the interaction with the environment
        # returning the updated state and the reward
        next_state, reward = interact_with_environment( action )
        memory_buffer.append( state, action, next_state, reward )
        update_network( memory_buffer )
        state = next_state
        # 'epsiode_done' is a flag that indicates terminal states
        if epsiode_done: break

if __name__ == "__main__":
    main()
```

Listing 2.1: A python-like implementation of the naïve DQN algorithm. The code follows the structure of a gym-like setup [OpenAI, 2021] and is a simplified version of the complete code presented in our repository [Corsi, 2022].

**Limitations of value-based approaches** Recalling the introduction of this section, the objective of a DRL approach is to find a policy, i.e., a strategy to solve a given problem. Value-

based approaches, in contrast, learn a value function that represents a sort of *proxy* for a model of the environment. Although this can be useful for generalization to unseen environments [Sutton and Barto, 2018], in general, this is out of scope. Fig. 2.5(a) shows a simple example of this first problem. In this example, to reach the goal, the robot should go straights; it does not need to learn the structure of the environment (smiles, stars, and suns). However, a value-based approach would try to learn the value of all squares, which is clearly out of scope. The previous one is an illustrative example; however, we can easily extend it to a navigation problem, where an autonomous car tries to learn the city's structure while only needing information about the road.




(a) The Blind Robot (b) The Deterministic Dilemma

Figure 2.5: Two toy examples to show the limitations of a value-based approach and a deterministic policy. (a) shows the classic example of the *blind robot*, where the solution does not require any information on the environment; (b) shows a trivial problem on which a deterministic policy fails in finding a solution.

Another crucial problem is determinism at inference time. Assuming to have the optimal value function for a robot that can rely only on local observations, the policy is to *always* choose the action with the highest Value, and this leads to a deterministic policy, i.e., in the state $s_k$, the agent always selects the action $a_k$. Fig. 2.5(b) depicts a trivial problem on which a deterministic policy fails. From a local observation point of view, the two states, marked with arrows A and B, are equivalent; this implies the agent always selects the same action, ending in an infinite loop. To solve this last problem, the simplest solution is to have a probabilistic policy, for example, $p(\text{right}) = 0.8$ and $p(\text{left}) = 0.2$, such that the agent has a high probability of solving the problem given a sufficient number of steps.

### POLICY GRADIENT

In this section, we answer the question: *is it possible to learn the policy directly?*. To achieve this goal, we think of the neural network as an approximator for a probability function, such as:

$$f_\theta(x) \sim p_\theta(action|state) \tag{2.5}$$

where $x$ is the input vector of the neural network that represents the current observation (i.e., current state). Recalling the main objective of a DRL algorithm, i.e., maximizing the expected reward ($R$) of a trajectory $\tau$, following a policy $\pi$ defined over the parameters $\theta$. The *objective function* to optimize can be formally written as follow:

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{2.6}$$

the first intuition is to perform a gradient ascent process to maximize this objective function. However, $J(\pi_\theta)$ does not trivially respect the requirements for a suitable objective function described in the previous section, given that $R(\tau)$ is not differentiable over the policy $\pi_\theta$ (i.e., $R(\tau)$ is a scalar value that does not depends on the parameters $\theta$ of the DNN). To obtain a differentiable function to maximize, we rely on the *policy gradient theorem*. In this thesis, we provide a simplified derivation of the theorem, referring interested readers to the original paper for details [Sutton et al., 1999]. The initial step is to apply the gradient for the gradient ascent process:

$$\nabla J(\pi_\theta) = \nabla\mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{2.7}$$

then, applying the definition of expectation, we obtain:

$$\nabla J(\pi_\theta) = \int_\tau \nabla\pi_\theta(\tau)R(\tau)\,d\tau \tag{2.8}$$

notice that the previous step can be interpreted as the probability of performing the trajectory multiplied by the corresponding reward. Before the next step, we introduce the *log-derivative trick* [Achiam, 2018] for a generic function $f(x)$, based on the the notion that $\nabla_x log(f(x)) = \frac{1}{f(x)}\cdot f'(x)$:

$$\nabla f(x) = f(x)\cdot\frac{\nabla f(x)}{f(x)} = f(x)\cdot\nabla log(f(x)) \tag{2.9}$$

by applying the *log-derivative trick* to Eq. 2.8, we obtain:

$$\nabla J(\pi_\theta) = \int_\tau \pi_\theta(\tau)\nabla log(\pi_\theta(\tau))R(\tau)\,d\tau \tag{2.10}$$

Eq. 2.10 has two key features. First, we now have $\pi_\theta$ in the objective function, and second, it is possible to apply back the definition of expectation:

$$\nabla J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla log(\pi_\theta(\tau))R(\tau)] \tag{2.11}$$

before the final step, we analyze in detail the meaning of $\pi_\theta(\tau)$. Intuitively, this is the probability of performing the trajectory $\tau$ following the policy $\pi$. However, we must also consider the transition model $T$ to obtain the probability of moving from one state to another given an

action. Given $s_t$ and $a_t$ the state and action of the time-step $t$, we formally define $\pi(\tau)$ as follows:

$$\pi_\theta(\tau) = p(s_0) \cdot \prod_{t=1}^{T} \pi_\theta(a_t|s_t) T(s_{t+1}|s_t, a_t) \tag{2.12}$$

however, in Eq. 2.11, we have $\nabla_\theta log(\pi_\theta(\tau))$, by applying the logarithmic properties and the derivative rules (e.g., $\frac{d}{dy} f(x) = 0$), we obtain the following equality:

$$\nabla_\theta log(\pi_\theta(\tau)) = \nabla_\theta \sum_{t=1}^{T} log(\pi_\theta(a_t|s_t)) \tag{2.13}$$

finally, applying Eq. 2.13 to Eq. 2.11, we obtain the final version, which is differentiable on the parameters of the DNN and is equivalent to the expectation of the reward following the policy:

$$\nabla J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta} [\sum_{t=1}^{T} log(\pi_\theta(a_t|s_t)) R(\tau)] \tag{2.14}$$

Equation 2.14 is the result of the policy-gradient theorem. This groundbreaking mathematical derivation is a fundamental result for the DRL community and, consequently, one of the foundations of this thesis. Crucially, Eq. 2.14 can be directly implemented as a key function for a DRL algorithm, and the resulting approach is *REINFORCE* (or *naïve policy-gradient*) [Sutton et al., 1999]. In Listing 2.2 we show the naïve implementation of REINFORCE; similarly to the previously discussed case of DQN, this can be considered the first policy-gradient approach, and likewise DQN, much effort has been developed to improve the performance of the algorithm. Among the others, we mention *baseline* [Sutton and Barto, 2018], *reward-to-go* [Tamar et al., 2016] and the *A2C* [Mnih et al., 2016] algorithm. Although all these improvements are crucial, we focus on the latter to introduce the next section. The main intuition is to replace the trajectory total reward $R(\tau)$ with the *advantage* of choosing a specific action $a$ in the state $s$ with respect to the expected reward the agent obtains starting from $s$ and following the current policy ($V(s)$). Following, we show a more formal definition of advantage:

$$A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.15}$$

ACTOR/CRITIC

Notice that, in a standard setup, the value function $V(s)$ of Eq. 2.15 is not provided and, given the model-free nature of the problem we are facing, is not obtainable in closed form. However, we can approximate this function with an additional DNN. The training of a DNN to learn a value can be encoded as a value-based DRL problem and solved with a standard algorithm (e.g., a derivation of DQN). This additional neural network that approximates $V(s)$, is called *critic*, while the original DNN that performs the policy update is called *actor*.

```
# Function that implements the actual learning phase of the training,
# we assume to have primitive functions for the execution of neural
# networks (e.g., predict and apply_gradient)
def update_network( memory_buffer ):
    trajectory_reward = 0
    trajectory_probability = 0
    for (action_probability, reward) in memory_buffer:
        trajectory_probability += math.log( action_probability )
        trajectory_reward += reward
    # Actual implementation of the policy-gradient theorem
    objective_function = trajectory_probability * trajectory_reward
    apply_gradient( objective_function )

# Main function that implements the standard reinforcement learning loop
def main():
    # In this loop, we assume to have the initial state for the first iteration
    while True:
        action, action_probability = select_action( state )
        # 'interact_with_environment' is an ideal function that,
        # given an action compute the interaction with the environment
        # returning the updated state and the reward
        next_state, reward = interact_with_environment( action )
        memory_buffer.append( state, action, action_probability,
            next_state, reward )
        state = next_state
        # 'epsiode_done' is a flag that indicates terminal states
        if epsiode_done: break
    #
    update_network( memory_buffer )
    # Remove all the elements from the buffer
    memory_buffer.clear()

if __name__ == "__main__":
    main()
```

Listing 2.2: A python-like implementation of REINFORCE. The code follows the structure of a gym-like setup [OpenAI, 2021] and is a simplified version of the complete code presented in our repository [Corsi, 2022].

Some might ask whether using the advantage compensates for an additional neural network requirement. Other than the empirical results that show significant improvements in almost every kind of problem [Mnih et al., 2016; Schulman et al., 2017; Haarnoja et al., 2018b,a], we briefly introduce the differences between *on-policy* and *off-policy* to understand better why an actor-critic architecture is useful. A crucial advantage of value-based approaches is that the value of a state is independent of the policy; this allows updating the value function with any source of data, regardless of how we collect them. This approach is called off-policy, which generally

is more *sample-efficient*, limiting the wasting of previously collected data. In contrast, in an on-policy setting (e.g., Eq. 2.14), the policy update rule refers to the trajectory reward $R(\tau)$, which strictly depends on the policy that controls the actions selection. From a mathematical perspective, the policy-gradient theorem can not be formally derived if the policy $\pi_\theta$ of the expectation is not the same as the $\pi_\theta$ of the objective function $J(\pi_\theta)$ (Eq. 2.6). The consequence is that it is possible to update the policy *only* with data collected with the same policy. After each gradient step, the policy is changed, losing all the data previously collected. A crucial strength of actor-critic architectures is that the critic can be trained off-policy (increasing the sample efficiency). At the same time, the actor maintains the advantages of an algorithm based on the policy gradient theorem.

**Proximal Policy Optimization** The final step of our *journey* into deep reinforcement learning is to present a state-of-the-art algorithm: *Proximal Policy Optimization* (PPO) [Schulman et al., 2017]. PPO is largely adopted for many problems and is widely considered the most versatile and better-performing approach. To explain the main idea behind this algorithm, we first briefly introduce another approach, considered its precursor: *Trust Region Policy Optimization* (TRPO) [Schulman et al., 2015]. TRPO proposes a simple intuition: small changes in the parameters of the DNN ($\theta$) can lead to considerable variations in the policy, potentially causing a tremendous decrease in the performance, hard to recover in the training loop (e.g., *catastrophic forgetting* [Goodfellow et al., 2013]). Consequently, instead of bounding the updated to the parameters $\theta$, TRPO tries to limit the policy changes at each gradient step, using the *KL-divergence*. However, computing this value is computationally expensive, making the entire process practically intractable for real-world problems.

Schulman et al. [2017] propose to approximate the KL-divergence by clipping the ratio between the policy at the current time step (the one that must be updated) and the policy in the previous time step. More formally, the new objective function is in the following form:

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[L(s, a, \theta, \theta_k)] \tag{2.16}$$

given the ratio $r(\theta)$ defined as:

$$r(\theta) = \frac{\pi_\theta(s|a)}{\pi_{\theta_k}(s|a)} \tag{2.17}$$

finally, the objective function of PPO is:

$$L(s, a, \theta, \theta_k) = min(r(\theta)A^{\pi_{\theta_k}}(s, a), \begin{cases} (1+\epsilon)A^{\pi_{\theta_k}}(s, a), & \text{if } A^{\pi_{\theta_k}}(s, a) \text{ is} \geq 0 \\ (1-\epsilon)A^{\pi_{\theta_k}}(s, a), & \text{if } A^{\pi_{\theta_k}}(s, a) \text{ is} < 0 \end{cases} \tag{2.18}$$

where $\epsilon$ is the *clip value* (in the original paper Schulman et al. [2017] suggest a fixed value of $\epsilon = 0.2$).

## 2.3 SAFETY CRITICAL TASKS

In the previous sections, we only considered a single objective for the training, the reward function. Historically, DRL has been studied and applied to benchmarking applications, such as classical games (e.g., chess [Silver et al., 2017] or go [Silver et al., 2016]), video games (e.g., Atari [Mnih et al., 2013]), and synthetic challenging environments (e.g., MuJoCo [Todorov et al., 2012], OpenAI Gym [Brockman et al., 2016]). Fig. 2.6 shows different examples of classical DRL problems.

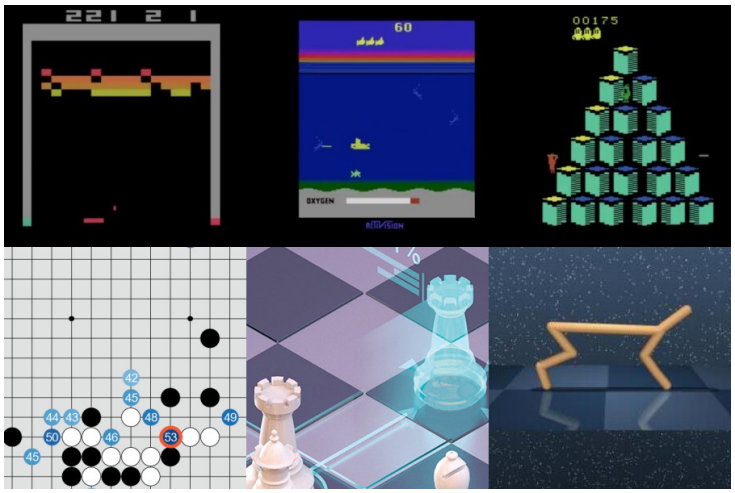



Figure 2.6: Examples of DRL classical problems, ranging from Atari to chess and go.

These problems are challenging and excellent case studies for developing more sophisticated techniques and valuable benchmarks for evaluating novel algorithms. However, in these problems, there is only one objective to achieve, only a function to maximize. Taking as examples the environments of Fig. 2.6, in the game of chess, winning the game is the only goal to reach, while in the Atari games, the only objective is to maximize the score. Moreover, they are simulated environments in which the consequences of wrong actions are not reflected in the real world.

In contrast, there is another class of problems where safety is a priority, sometimes even more important than the primary objective, called *safety-critical*. To provide an intuition, we can think about a plane full of passengers. If a critical failure is detected, it is trivially better to land as fast as possible (safety requirement) instead of trying to reach the destination (primary objective), exposing people to risk. We identify this family of problems as *safety-critical tasks*. Robotics is a typical example; when we work with robots, there are two fundamental reasons to focus on safety: (i) high-cost equipment must be considered, and (ii) human safety can be involved. We identified three fundamental questions and challenges about safety-critical tasks:

- What are the safety-critical tasks? And, what are the additional challenges they present?

- How to formally encode and solve them?
- Can we provide guarantees about the safety of the resulting policies?

In this section, we answer the first question, showing three different real-world problems we faced in our studies, describing the robotic platforms, and explaining the crucial challenges. The answers to the second and third questions constitute the main contribution of this thesis and are the focus of, respectively, Part I and Part II of this manuscript.

### NAVIGATION AND MAPLESS NAVIGATION

Robotic Navigation is the task of navigating a robot through an environment. The robot's goal is to reach a target destination while adhering to predefined restrictions, e.g., selecting as short a path as possible, avoiding obstacles, or optimizing energy consumption. In recent years, robotic navigation tasks have received much attention, among other reasons, for their extendibility to autonomous vehicles [Pan et al., 2017]. Navigation is a classical robotic problem, and together with manipulation, it is often considered one of the two fundamental branches of the robotic research. For this reason, it has been extensively studied over the years, and different *algorithmic* methods have been developed to solve it (e.g., planners and search-based approaches) [Temeltas and Kayak, 2008].

However, there is a variant of the robotic navigation problem, called *mapless navigation*, where the map of the surrounding environment is unknown, and the robot can rely only on local observations (i.e., the robot's sensors). This configuration presents a set of challenging problems. First, the absence of the map prevents (or makes it difficult) the adoption of the previously mentioned planning-based methods. Second, the limited information collected with the local sensors makes the problem *not fully observable*, introducing additional challenges like the noise of the sensors and the impracticability of the consequences of actions [Marchesini and Farinelli, 2020]. Although this formulation may seem to be an unnecessary (and artificial) complication, many real-world problems can be encoded as a mapless navigation problem, for example:

- *Exploration of mines and caves*, where the environment is completely unknown, and the map can not be obtained.
- *Collision avoidance in the presence of dynamic obstacles* (e.g., humans or other agents) that can not be easily modeled.
- *Underwater monitoring* (or in a borderline case, space exploration), where a connection with the main computation unit is not possible for physical limitations, and the agent should be able to make autonomous decisions in unexpected situations.

- *Navigation in unstable contexts*, where the structure of the environment change frequently, and so the computation of the map is unnecessarily expensive (e.g., rooms in which obstacles can be moved daily).

State-of-the-art mapless navigation solutions suggest training a DRL policy to control the robot. Such DRL-based solutions have obtained outstanding results from a performance point of view [Zhu et al., 2017; Bojarski et al., 2016]. Specifically, the recent work of Marzari et al. [2022] has demonstrated how DRL-based agents can be applied to control an agent in a mapless navigation setting by training a DNN with a simple architecture (e.g., small feed-forward neural networks). Notice that a limited DNN's size is also a crucial requirement for the *onboard* control of the robot.

(a) Robotis TurtleBot3 (b) Robotis TurtleBot4

Figure 2.7: The two versions of the Robotis TurtleBot we used in our experiments, respectively TurtleBot3 and TurtleBot4 (in the two versions, standard and lite) *Images from the official website.*

**Robotic Platform** Throughout the thesis, we performed most of the navigation experiments and analysis on the differential mobile robot *TurtleBot* [Nandkumar et al., 2021], Fig 2.7 shows the two versions of the platform. TurtleBot is widely used in robotics research [Ruan et al., 2019; Zamora et al., 2016]. In particular, this robotic platform comes with the actuators required for moving and turning, a lidar sensor for detecting obstacles, and an RGB camera optionally. More details (and technical specifications) about the robot are provided in Chapter 8, in which we conduct a comprehensive analysis of the methodologies proposed in this thesis using this platform as a case study.

**Safety Challenges** Mapless Navigation is a concrete example of the two characteristics that make a task safety-critical: (i) expensive hardware and (ii) human safety involved. In this context, collision avoidance is a requirement, at least as strong as the main objective of *reaching the target position*, in some cases even more (e.g., costly platforms or heavy machines in crowded areas). This concept is further stressed by the inherent constraints of the mapless setup. Some-

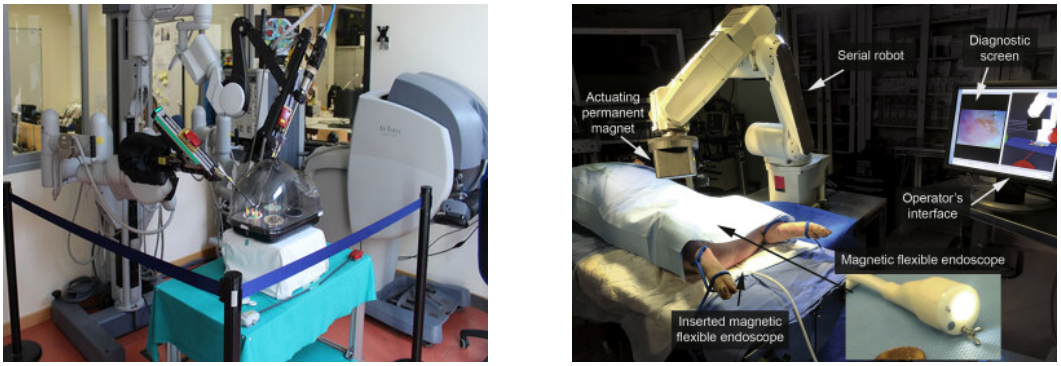
times, the task is so complex that getting stuck in a loop without compromising safety can be considered a good accomplishment.

### MEDICAL APPLICATIONS

As we already introduced, the success of DRL-based systems has naturally led to their integration as control policies in safety-critical tasks. Various papers have shown that DNNs can be highly effective also when applied to medical contexts [Pore et al., 2021, 2022]. In this scenario, human safety is trivially involved. We categorize the medical application into two macro areas:

- *Surgical Assistance*, where the robot helps the surgeon in the operations, replacing a human assistant for some complementary tasks.
- *Autonomous Operation*, where the robot acts autonomously, without human intervention.

In our studies, we experimented with both setups. For the first, we studied the *tissue retraction* problem. A recurring sub-task occurs during a minimally invasive surgery that involves manipulating deformable connective tissues to access the region of interest, such as a tumor. For the second, we analyzed the *colon exploration* problem, which is typically performed with a wireless capsule that navigates through the colon. In the most advanced systems, the capsule is controlled by magnets attached to a robotic manipulator, and a low-level controller translates the local navigation instructions into the commands for the manipulator. Consequently, from a high-level perspective, this problem can also be viewed as a corner case for mapless Navigation.




(a) DaVinci Surgical System (DVSS)

(b) Magnetic Capsule Endoscope

Figure 2.8: The two robotic platforms we studied for the medical applications, on the left the DVSS, used for our experiments on tissue retraction; on the right, the system for the colonoscopy exploration with a magnetic capsule endoscope presented in the work of Slawinski et al. [2018].

**Robotic Platform** Our case studies consider two different platforms. The DaVinci Surgical System (DVSS) consists of several instruments for tissue retraction tasks. DVSS comprises three robotic arms called *patient side manipulator* equipped with articulated minimally invasive surgery instruments [Attanasio et al., 2020; Pore et al., 2021]. For the colonoscopy problem, we refer to the work of Pore et al. [2022] for details; however, from a high-level perspective, the system is composed of a robotic arm that controls a wireless capsule exploiting a magnet-based system. Fig 2.8 shows two pictures of the presented systems.

**Safety Challenges** Medical applications are probably one of the most important examples of safety-critical tasks. In these contexts, DRL controllers operate directly *on* the human and not only *with* the human. Moreover, humans often cannot react to any misbehavior, making the task even more dangerous. In such conditions, it is clear that the priority is to preserve the patient's health, eventually even failing to complete the task. From a practical point of view, in the tissue retraction case, an example of a safety requirement is to always operate inside a safe working space (in Fig. 2.9 this area is marked with a blue cylinder). In the colonoscopy case, the crucial requirement is collision avoidance with the walls, which can cause injuries to the patient.

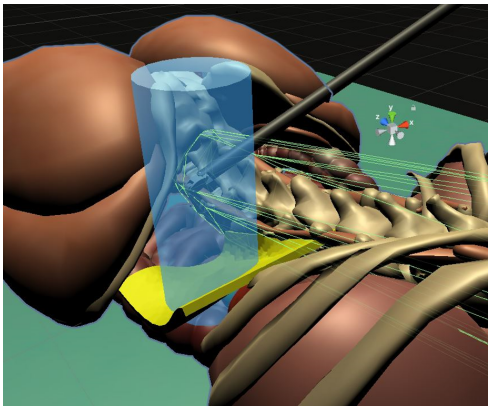


Figure 2.9: example of a safety requirement for the tissue retraction problem, the robot must operate inside a specific workspace to avoid collision with the patient. The blue cylinder marks the safe area.

AQUATIC ENVIRONMENT

The last problem we consider in this analysis can also be seen as an advanced and more challenging version of the mapless navigation problem. The crucial difference that makes this task interesting is the general instability of the environment. The waves, for example, are an unpredictable external agent that significantly increases the system's complexity. Another challenge is the unpredictability of the actions' outcome. The consequences of an action depend not only

on the agent's decisions but also on external elements that are difficult to model (e.g., surface stream, water flow, water density, and, more in general, fluid physics).

Figure 2.10: The aquatic drones used in our experiments. The drone has been developed as part of the INTCATCH 2020 European project.

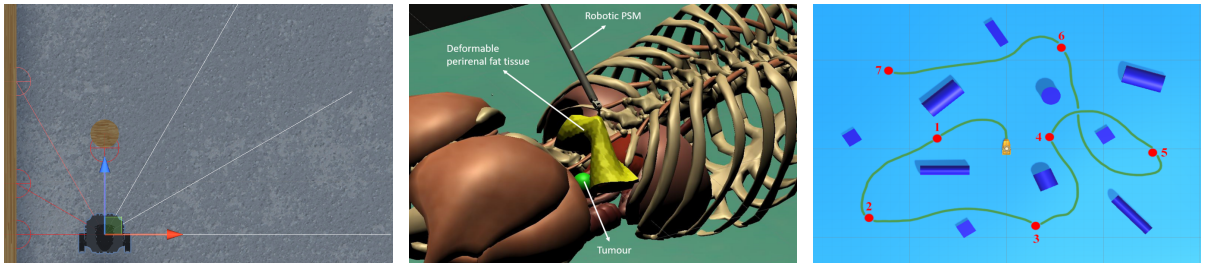
**Robotic Platform** In our work we consider the INTCATCH drone (Fig. 2.10) which is a a differential drive platform. It is based on a hull that is equipped with two in-water propellers. The drone can be deployed in shallow water, with a max velocity of $3m/s$. The onboard sensors (e.g., GPS, compass) provide the localization and orientation information, while a lidar collects the distances between the boat and obstacles.

**Safety Challenges** Generally, the challenges are shared with the standard navigation tasks. However, the requirements are more strict. For example, a collision in an aquatic environment leads to more severe damage to the robot (e.g., a water leak caused by a collision can fatally damage the onboard electronics). Navigating in an aquatic environment is an additional example where mapless navigation is the best way to encode the problem. Not only because the map cannot be easily obtained but also because the decision should be taken based on current and local observation (i.e., it can be hard to *plan* a complete sequence of actions in such a dynamic context).

## 2.4 UNITY SIMULATION ENGINE

To perform the training of our robots, we rely on *Unity3D*, a popular engine originally designed for game development that has recently been adopted for robotics simulation [Pore et al., 2021; Marchesini et al., 2021a]. In particular, the built-in physics engine, the powerful $3D$ rendering algorithm, and the time control system (which allows to speed up the simulation by more than

10 times), have made *Unity3D* a very powerful tool in these contexts [Juliani et al., 2018]. The extreme versatility of this software allows us to adopt it for all the simulations we needed for our experiments. Crucially, the simulation has been demonstrated to be realistic enough to allow the portability of the trained DNNs to the real-world robot. We have shown this key characteristic in our previous works [Pore et al., 2021; Marchesini et al., 2021a; Corsi et al., 2021; Marchesini et al., 2021b]. Moreover, in Part III of this thesis, we show a complete case study of the approaches presented in this thesis. Moreover, the last versions of Unity provide a complete pipeline and additional tools to extend the compatibility of the systems with the Robot Operating System (ROS2) to guarantee a simplified simulation-to-real transfer of the trained models. Finally, Fig. 2.11 shows some examples of our Unity environments, presenting one screenshot for each of the safety-critical tasks we presented in this section (i.e., mapless navigation, tissue retraction, and aquatic environment).




(a) Mapless Navigation (b) Tissue Retraction (c) Aquatic Simulation

Figure 2.11: Three examples of the simulations we developed for our robotic experiments. All the simulations have been developed with the Unity3D engine, showing excellent performance with respect to other state-of-the-art simulation engines (e.g., Gazebo) [Marchesini, 2022].

# Part I

# Safe Training for Deep Reinforcement Learning

# Chapter 3

# PRELIMINARIES

This chapter is intended as an introduction to the Part I of the thesis, where we will present different methods to improve the reliability (and safety) of the policies directly inside the training loop. In particular, we present a novel approach that exploits the *lagrangian duality* to extend the concept of *constrained optimization* to a DRL context, showing how it is possible to optimize the policy respecting a given set of requirements. In detail, in Chap. 4 we introduce a novel approach for the definition of such requirements based on the formal language *Scenario Based Programming (SBP)*. In the next sections, we introduce the foundational concepts to understand the main algorithm we present in the first part of this dissertation: *Scenario Based Lagrangian PPO*. Moreover, we present a critical analysis of the state-of-the-art algorithm for solving a constrained DRL problem, highlighting why we chose to focus on the Lagrangian family of approaches.

## 3.1 CONSTRAINED OPTIMIZATION

The mathematical problem of constrained optimization has been widely studied for centuries. In this section, we provide only a high-level intuition about it, introducing only the fundamental concepts necessary to understand the algorithms covered in this thesis. To further investigate these topics, we refer interested readers to the work of Bertsekas [2014]. The standard form for a continuous optimization problem, defined over the vector variables $\vec{x}$ is:

$$\begin{aligned} \min_{\vec{x}} \quad & f(\vec{x}) \\ \text{s.t.} \quad & g(\vec{x}) = c \end{aligned} \tag{3.1}$$

to solve this problem, the key observation is that the minimum (or the maximum) point of $f$, subject to $g(\vec{x}) = c$, always corresponds with a contour line of $f$, tangent to the curve $g(\vec{x}) = c$. Notice that, in the tangent point between two functions, the two gradient vectors evaluated on

this point are parallel, and there are no more specific requirements on their magnitude [Bertsekas, 2014]. By calling this magnitude $\lambda$ (i.e., the *lagrangian multiplier*), we can formally write this *tangency condition* in the following form:

$$\nabla f(\vec{x}) = \lambda \nabla g(\vec{x}) \tag{3.2}$$

finally, combining Eq. 3.2 with the original constraint $g(\vec{x}) = c$, we obtain a system with the same number of variables and equations, which solution returns the optimal point. Following, we provide a numerical example to better understand the method. Suppose an objective function to minimize $f(x, y) = 2x + y^2$ under the constraint $g(x, y) = x + y = 3$, the system of equations to solve is:

$$\begin{cases} \nabla_x(2x + y^2) = \lambda \nabla_x(x + y) \\ \nabla_y(2x + y^2) = \lambda \nabla_y(x + y) \\ x + y = 3 \end{cases} \tag{3.3}$$

computing the partial derivative of the first equation, we obtain the following system of 3 equations and 3 variables:

$$\begin{cases} 2 = \lambda \cdot 1 \\ 2y = \lambda \cdot 1 \\ x + y = 3 \end{cases} \tag{3.4}$$

solving the system, we obtain that the minimum is 5 in $(2, 1)$. In the 1700's Joseph Louis Lagrange proposes a unique function that expresses all these conditions in only one equation, called the *lagrangian function*:

$$\mathcal{L}(\vec{x}, \lambda) = f(\vec{x}) + \lambda(g(\vec{x}) - c) \tag{3.5}$$

the solution to the constrained optimization problem is the same as finding the stationary point of $\mathcal{L}(\vec{x}, \lambda)$. Notice that the lagrangian function can be extended to the case of multiple constraints, adding a multiplier $\lambda$ for each constraint. The general form for multiple constraints results as follows:

$$\mathcal{L}(\vec{x}, \vec{\lambda}) = f(\vec{x}) + \sum_{i=0}^{n} \lambda_i (g_i(\vec{x}) - c_i) \tag{3.6}$$

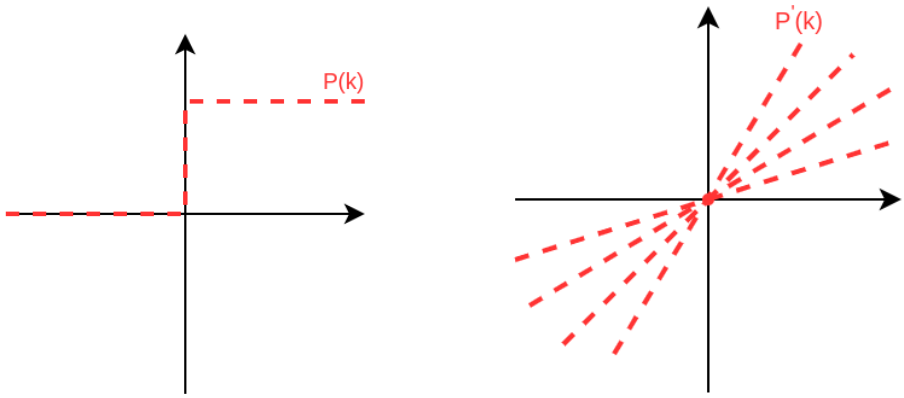
where $n$ is the number of constraints. So far, we have analyzed the case with only equality constraints. This is a strong limitation, especially in the case of DRL, where the numerical optimization process intrinsically does not allow focus on such a strict requirement. We now focus on another class of optimization problems, which includes inequalities constraints, in the following form:

$$\begin{aligned} \min_{\vec{x}} \quad & f(\vec{x}) \\ \text{s.t.} \quad & g(\vec{x}) > c \end{aligned} \tag{3.7}$$

given the nature of DRL, to obtain a problem that we can address via a numerical method (e.g., gradient descent), the ideal solution is to relax this constrained optimization problem into an unconstrained one in the following form:

$$\min_{\vec{x}} f(\vec{x}) + P(k) \tag{3.8}$$

where $k = g(\vec{x}) - c$ encodes the constraint and $P$ is an indicator function that approaches $\infty$ if the constraint is violated (i.e., to minimize the global function, the respect of the constraints must be prioritized). The exact solution involves the use of a step function (Fig. 3.1a). However, this kind of function is discontinuous(and so not differentiable), and thus it can not be optimized with a numerical method (e.g., gradient descent).




(a) Step Function (b) Family of Straight Lines

Figure 3.1: Comparison between the step-function and the family of straight lines for the lagrangian relaxation.

Ideally, a differentiable linear function should be used. However, representing a step function with only one *line* is not possible; a popular solution is to use a family of straight lines (Fig. 3.1b) in the following form:

$$P(k) = \max_{u \geq 0} k \cdot u \tag{3.9}$$

notice that if $k$ is negative (i.e., the constraint is respected), the support variable $u$ is reduced to zero; otherwise (i.e., the constraint is violated), $u$ grows to infinity. Finally, combining Eq. 3.8 and Eq. 3.9 we obtain the following unconstrained relaxation of the optimization problem:

$$\min_{\vec{x}} \max_{u \geq 0} f(\vec{x}) + u \cdot (g(\vec{x}) - c) \tag{3.10}$$

Notice that the objective function has a similar structure as the lagrangian function; for this reason, this method is often called *lagrangian relaxation* and the support variable $u$ indicated

with $\lambda$. In conclusion, the constrained optimization problem of Eq. 3.7 can be relaxed to obtain the following unconstrained problem:

$$\min_{\vec{x}} \max_{\vec{\lambda} \geq 0} \mathcal{L}(\vec{x}, \vec{\lambda}) \tag{3.11}$$

## 3.2 EVOLUTIONARY ALGORITHMS

In this section, we introduce the general idea behind the Evolutionary Algorithms (EA) as an alternative black-box optimization for a deep neural network (DNN) that shows promising results when applied in a constrained DRL setting, especially in combination with a standard gradient-based optimization process. An EA, and in particular a *Genetic Algorithms* (GA), is characterized by three main operators [Fogel, 2006]: (i) generation; (ii) alteration; and (iii) selection. The main loop is depicted in Fig. 3.2 while python-like pseudocode is reported in Listing 3.1

```
# Function that implements the genetic algorithm, we assume to have the
# following functions for the support operations, that implements
# the functions described in this chapter:
#    -> generate_random_population
#    -> evaluation
#    -> crossover
#    -> mutation
def genetic_step( population ):
    # Evaluation Step
    score = [ evaluate(individual) for individual in population]
    # Selection Step
    best_p1, best_p2 = evaluation(  population, score )
    # Crossover
    merged_model = crossover( best_p1, best_p2 )
    # Mutation
    new_population = mutation( merged_model )

# Main DRL training loop,
def main_loop():
    # Generation of the initial random population
    population = generate_random_population()
    # Iterate the genetic step until convergence
    while not converged( best_model ):
        best_model, population = genetic_step( population )
    return best_model
```

Listing 3.1: A python-like implementation of the standard loop of a Genetic Algorithm.

In detail, a GA evolves a population of $N$ individuals, each one represented by the network vector parameter $\theta$ (genome). Each $\theta_i$ ($i \in [0, .., N-1]$) is evaluated with respect to some

requirements to produce a fitness $F(\theta_i)$, used by the selection operator to choose the best genome. The best-performing models can be merged together and, with an additional mutation step, used to generate a new population of individuals. The process iterates until convergence, i.e., there are no significant behavioral differences between successive generations or when a model matches the desired requirements.

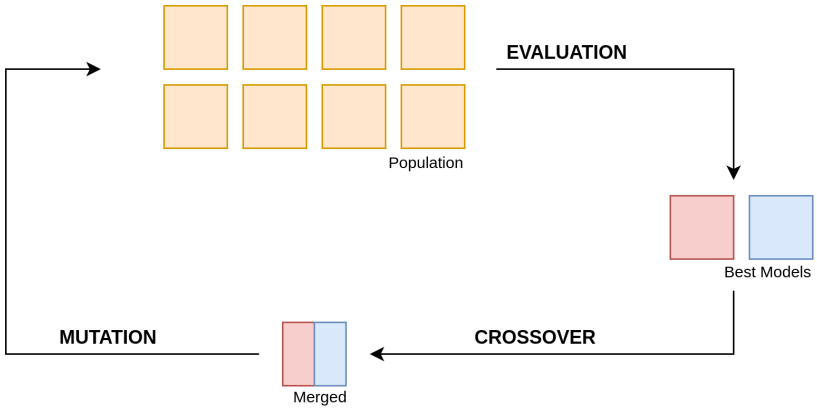



Figure 3.2: The standard loop for a Genetic Algorithm.

**Genetic Soft Update** An EA is a black-box optimization characterized by generation, perturbation (mutation), and selection operators [Fogel, 1995], which can be used to augment exploration [Khadka and Tumer, 2018]. EAs typically evolve a population of $p \in \mathbb{N}$ individuals (genomes), represented by parameters (weights) $\theta_i$ ($i \in \{1, \dots, p\}$). The individuals are evaluated to produce a fitness score used by the selection operator to choose the best genome. Mutating a policy with simple Gaussian noise $\mathcal{N}$, however, can lead to disruptive changes [Lehman et al., 2018] that can be naively addressed using zero-mean and low standard deviation [Martín and Lope, 2009]. Otherwise, if we define a genome as a DNN parameterized by $\theta$ that represents a function $f_\theta : \mathcal{D}_\mathbf{x} \to \mathcal{D}_\mathbf{y}$ (input $\mathbf{x} \in \mathcal{D}_\mathbf{x} \subseteq \mathbb{R}^n$ and output $\mathbf{y} \in \mathcal{D}_\mathbf{y} \subseteq \mathbb{R}^m$, with input/output size $n$, $m$), and a vector of states $\mathbf{s}$, we can express the average divergence of the outputs $\mathbf{y}$ as a result of a perturbation $\delta$ as:

$$d(f_\theta, \delta) = \frac{\|f_\theta(\mathbf{s}) - f_{\theta+\delta}(\mathbf{s})\|_2}{|\mathbf{s}|} \tag{3.12}$$

where $f_\theta(\mathbf{s})$ are the forward propagation of the states through the DNN. A more flexible way to avoid disruptive mutations assumes using a differentiable DNN to approximate $d$ with gradient information [Lehman et al., 2018]. In detail, it considers the following first-order Taylor expansion to model an output $y_j \in \mathbf{y}$ ($j \in \{0, \dots, |\mathbf{y}|\}$) as a function of perturbations $\delta$ over the

states $\mathbf{s}$:

$$y_j(f_\theta, \delta) = f_\theta(\mathbf{s})_j + \delta \nabla_\theta f_\theta(\mathbf{s})_j \tag{3.13}$$

In later sections, we discuss how our SM computes safety-informed perturbations, specializing in the naive gradient-informed mutations based on Eq. 3.13, to foster safety-oriented exploration.

## 3.3 CONSTRAINED MARKOV DECISION PROCESS

In the previous section, we introduced the fundamental concept of optimization with inequalities constraints. However, in the context of numerical policy optimization for DRL, a specific encoding for the problem is required. A standard formalization, proposed in a wide variety of state-of-the-art works, and considered as a standard in the community [Ray et al., 2019; Liu et al., 2020; Achiam et al., 2017; Corsi et al., 2022; Marchesini et al., 2021b, 2020], is through a *Constrained Markov Decision Process* (CMDP). The idea is to complement the standard reward function of the DRL process with an additional function (or more than one) called *cost function*. While the optimization process aims at maximizing the reward signal, the cost function is constrained to stay below a given threshold. Through this formulation, it is possible to encode a large variety of requirements. In Chap. 4, for example, we show how it is possible to exploit this method to inject prior knowledge, and in Chap. 9 we introduce some ideas on how to exploit this method for future research directions. Crucially, the cost function is not necessarily required to be minimized toward zero, exploiting the threshold, it is possible to guarantee safety without compromising the expressiveness and the generalization power of the deep neural networks. Following a formal definition for CMDP.

**Formalization** A CMDP [Altman, 1998] is a Markov Decision Process (MDP) with an additional set of constraints $\mathcal{C}$ which consists of $C_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ $(i \in \{1, \dots, m\})$ cost functions (similar to the reward) with $\mathbf{h} \in \mathbb{R}^m$ thresholds for the constraints. The $C_i$*-return* is thus defined as $J_{C_i}(\pi) := \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t C_i(s_t, a_t)]$, where $\tau = (s_0, a_0, \dots)$ is a trajectory, $\pi = \{\pi(a|s) : s \in \mathcal{S}, a \in \mathcal{A}\}$ denotes a policy in state space $\mathcal{S}$ and action space $\mathcal{A}$, and $\gamma \in (0, 1)$ is the discount factor. The constraint-satisfying policies $\Pi_\mathcal{C}$ (i.e., feasible policies), and the optimal policies $\pi^*$ are thus defined as:

$$\Pi_\mathcal{C} := \{\pi \in \Pi : J_{C_i}(\pi) \leq h_i,\ \forall i\} \qquad \pi^* = \arg\max_{\pi\in\Pi_\mathcal{C}} J(\pi). \tag{3.14}$$

where $\Pi$ are the stationary policies, $J(\pi) := \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)]$ is the expected discounted return that we aim at maximizing in a standard MDP, and $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. Without loss of generality, we consider the case of one cost function (as in recent constrained DRL literature [Ray et al., 2019; Stooke et al., 2020; Liu et al., 2020]) and we will discuss later how SOS could handle multiple cost functions.

## 3.4 STATE OF THE ART APPROACHES TO CMDP

In this section, we discuss the most relevant approaches and algorithms to solve a CMDP via deep reinforcement learning. Lagrangian-DRL methods (e.g., lagrangian-PPO [Ray et al., 2019]) directly exploit the lagrangian relaxation of a constrained optimization problem into a policy gradient DRL algorithm. In the next chapter, we dive into this family of approaches, introducing our algorithm to face this challenging problem. However, some other methods exist; in this section, we briefly introduce the intuition behind these methodologies, leaving interested readers to the original papers for more details, derivations, and experimental results on different standard benchmarking environments. Crucially, some of the following algorithms constitute the baselines for the next chapters.

### CONSTRAINED POLICY OPTIMIZATION (CPO)

Constrained Policy Optimization (CPO) is one of the first algorithms to provide a methodology to enforce a set of constraints into the training loop [Achiam et al., 2017]. The key idea behind CPO is to use a trust region optimization method to simultaneously optimize the policy while enforcing a set of constraints. The concept of trust region has been introduced in TRPO (trust region policy optimization) by Schulman et al. [2015]; here the key idea was to update the policy without drastically changing the behavior of its most recent iteration, enforcing a limit in the KL-divergence between a policy at time $t$ and the updated policy at time $t+1$. In addition to this requirement, CPO additionally requires the policy to meet the constraints at each iteration, limiting the trust region for the policy search in a feasible area for the constraints. More formally:

$$\begin{aligned} \pi_{k+1} = \arg\max_{\pi \in \Pi_\theta} \quad & J(\pi) \\ \text{s.t.} \quad & J_c(\pi) \leq d \\ \text{s.t.} \quad & D(\pi, \pi_k) \leq \delta \end{aligned} \tag{3.15}$$

where $\Pi_\theta$ is the space of the possible policies, $J(\pi)$ is the objective function for the reward, $J_c(\pi)$ the objective function for the cost with the corresponding limit $d$, and $D(\pi, \pi_k)$ a distance metric between two policies, (typically the KL-divergence). Although CPO provides strong theoretical guarantees of convergence to an optimal and safe policy (we refer to the original paper for the proof [Achiam et al., 2017]), in practice it requires a series of relaxations to solve the trust region optimization problem. These approximations required for the actual implementation, however, lead to poor performance on complex tasks. This result is highlighted in a follow-up paper from the same authors [Ray et al., 2019], where they empirically show that the lagrangian approaches perform better than CPO on a set of complex locomotion benchmarks and it is further validated in a series of more recent papers [He et al., 2023; Marchesini et al., 2021b; Liu et al., 2020]. In conclusion, recent developments in the field have revealed that CPO is no longer the best option for solving a CMDP; however, it is still considered a fundamental baseline and a potential

base for further improvement and extension (e.g., projection-based CPO [Yang et al., 2020] and RCPO [Tessler et al., 2019]).

### INTERIOR POINT POLICY OPTIMIZATION (IPO)

To overcome some of the limitations of CPO, and in particular the implementation complexity and the required approximation, Interior Point Policy Optimization (IPO) [Liu et al., 2020] proposes a first-order optimization method based on the concept of logarithmic barrier function. The key idea is to add a penalty to the objective function of PPO to push the policy towards a safer region. Recalling the notation from the previous section, we call $J(\pi)$ the objective function for the reward, $J_c(\pi)$ the objective function for the cost, and $d$ the limit for the cumulative cost. The objective function for IPO becomes:

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] - I(C(\tau)) \tag{3.16}$$

Ideally, $I$ is a penalty that should be modeled as an indicator function in the following form:

$$I(J_c(\pi) - d) = \begin{cases} 0 & J_c(\pi) \leq d \\ -\infty & J_c(\pi) > d \end{cases} \tag{3.17}$$

broadly speaking, Eq.3.17 means that if the constraint is satisfied, the problem translates into an unconstrained optimization problem, focusing only on the reward, while, if the constraint is violated, adjusting the constraint becomes a priority for the optimization algorithm (i.e., the penalty is $-\infty$). However, an indicator function such as Eq.3.17 is not differentiable, therefore, to be optimized with a gradient-based approach, the problem should be relaxed with an approximation of $I$; in IPO, the authors propose the use of a logarithmic barrier function in the form of $\Phi(J_c(\pi) - d) = \frac{-log(J_c(\pi)-d)}{t}$, where $t$ is a hyperparameter for the optimization. Although IPO shows promising results in some environments, it suffers from different limitations. For example, the impact of the parameter $t$ is opaque and requires a binary search to be tuned; more crucially, the binary nature of Eq.3.17 does not allow for finding a continuous value for the penalty that balances the reward and the cost. This crucial limitation can be addressed with a Lagrangian approach that, at the same time, maximizes the objective functions while searching for the best possible penalty. For this reason, IPO is typically not considered a state-of-the-art algorithm in the most recent work on CMDP; however, it provides an interesting baseline for the reward-shaping family of algorithms.

### REWARD CONSTRAINED POLICY OPTIMIZATION (RCPO)

As mentioned before, IPO provides an algorithm that exploits an efficient and elegant technique to provide a penalty to the objective function without requiring solving an additional optimization problem to find the best magnitude for a penalty (e.g., lagrangian reinforcement learning).

In the same direction, Reward Constrained Policy Optimization (RCPO) [Tessler et al., 2019] proposes to exploit an actor-critic architecture to directly learn a penalized reward function that can be exploited by any state-of-the-art policy-gradient algorithm (e.g., PPO or TD3). The key idea is to enrich the reward function with a weighted penalty based on the cost, more formally:

$$r_{rcpo}(\lambda, s, a) = r(s, a) - \lambda \cdot c(s, a) \tag{3.18}$$

where $r(s, a)$ is the reward and $c(s, a)$ is the cost function. Notice that, in this formulation, the weight value $\lambda$ is similar to a lagrangian multiplier but, in contrast to a *"complete"* lagrangian method, is not part of the optimization process. The crucial insight of RCPO is that it exploits only one critic for the optimization to directly predict the advantage of the penalized reward function, i.e., $V^{\pi}(\lambda, s) = V_R^{\pi} - \lambda \cdot V_C^{\pi}$. The policy network can be trained with any actor-critic approach (we refer to Chap. 2 for more details on this class of algorithms), while the $\lambda$ parameter is updated inside the loop increasing its value when the constraint is violated. Formally: $\lambda_{k+1} = \lambda_k + \eta \cdot (C - d)$, where $\eta$ is a scaling factor that can be viewed as a learning rate, $C$ is the cost of the episode and $d$ the corresponding threshold.

### SAFETY-ORIENTED SEARCH (SOS)

We finally introduce Safety-Oriented Search (SOS), an alternative approach that combines evolutionary algorithms with a classical gradient-based DRL approach [Marchesini et al., 2021b]. The key intuition behind this algorithm is to bias the policy toward safer regions within an evolutionary cost optimization performed concurrently with the standard gradient-based DRL loop. SOS leverages the exploration benefits of *Evolutionary Algorithms* (EA) to design a novel concept of safe mutations that use the visited unsafe states to explore safer behaviors. The behavior of the policies is then characterized by an additional selection step based on the evaluation of a set of given properties that encodes some prior human knowledge or safety requirements. Hence, driving the learning process toward safer regions of the policy space. In more detail, the algorithm exploits the visited states, which are deemed unsafe according to the cost, to approximate the per-weight sensitivity of the actions over such undesired situations. The sensitivity is then used to compute safety-informed perturbations that locally bias the agent policy to explore different actions in the proximity of the unsafe states (i.e., the *exploration process*).

### DISCUSSION

In recent years, constrained optimization has gained considerable attention, and researchers have dedicated significant effort toward developing more effective and efficient algorithms to tackle this challenging problem. To this end, several baselines and benchmarks have been established, and a range of algorithms and environments are continually being developed and analyzed. Despite these efforts, it remains unclear which algorithm or method should be considered the best performing. For example, although Constrained Policy Optimization (CPO)

has demonstrated strong theoretical guarantees, it requires a set of approximations that limit its effectiveness, particularly when dealing with complex problems, such as those encountered in robotic control contexts. Notably, in a follow-up paper Ray et al. [2019], the same authors of CPO, acknowledge the limitations of their approach and show that "complete" Lagrangian methods outperform CPO in a set of locomotion benchmarks. Genetic-based approaches offer a promising alternative for solving Constrained Markov Decision Processes (CMDPs); however, their black-box nature poses a significant challenge in terms of the learning process transparency and the applicability of algorithms, such as Safe Optimization via Simulation (SOS). Moreover, genetic-based approaches require additional effort to be implemented in conjunction with standard deep reinforcement learning techniques. Despite these challenges, recent work by the authors of SOS [Marchesini et al., 2021b] shows that incorporating an evolutionary step within the loop can yield significant benefits and warrants further investigation. Finally, while "naive" reward-shaping approaches have been shown to be ineffective for complex tasks, more sophisticated methods, such as Reward Constrained Policy Optimization (RCPO), have demonstrated promising results. Nonetheless, recent benchmarking studies suggest that Lagrangian methods provide better overall performance, particularly in robotics control problems [He et al., 2023; Ray et al., 2019; Gronauer, 2022]. In Chap. 4, we provide a comparison between CPO, RCPO, IPO, and the standard version of Lagrangian PPO, showing that the latter outperforms the others in our robotics and locomotion benchmarks. Therefore, we focus on Lagrangian algorithms in this thesis, although we recognize the importance of investigating other methodologies.

# Chapter 4

# CONSTRAINED REINFORCEMENT LEARNING

In the previous sections, we introduced the concept of *safety-critical* problems in *Deep Reinforcement Learning* (DRL), where the learning community has been seeking to create DRL-based controllers that simultaneously demonstrate high *performance* and high *reliability*; i.e., are able to perform their primary tasks while adhering to some prescribed properties, such as safety and robustness. An emerging family of approaches for achieving these two apparently concurrent goals, known as *constrained DRL* [Achiam et al., 2017], attempts to simultaneously optimize two functions: the *reward*, which encodes the main objective of the task; and the *cost*, which represents the safety constraints. In the previous chapter, we discussed some of the current state-of-the-art algorithms including IPO [Liu et al., 2020], SOS [Marchesini et al., 2021b], CPO [Achiam et al., 2017], and Lagrangian different approaches [Ray et al., 2019; Roy et al., 2021]. Despite their success in some applications, all these methods generally suffer from significant setbacks: (i) there is no uniform and human-readable way of defining the required safety constraints; (ii) it is unclear how to encode these constraints as a signal for the training algorithm; and (iii) there is no clear method for balancing the numerical instability of the cost and reward optimization during training, and thus there is a risk of producing sub-optimal policies (i.e., over-conservative or unsafe). In the previous chapter, we presented a novel framework to face this problem: *Safety-Oriented Search* (SOS) [Marchesini et al., 2021b]. Although this approach shows promising empirical results, the genetic nature of the algorithm intrinsically limits the explainability and fails to provide both convergence and safety guarantees. These are strong limitations that become crucial in the context of safety-critical tasks.

In this chapter, we present a novel approach for addressing these challenges by enabling users to encode constraints into the DRL training loop in a simple yet powerful way. Our approach generates policies that strictly adhere to these user-defined constraints without compromising performance. We achieve this by extending and integrating two approaches: the *Lagrangian-PPO* algorithm [Ray et al., 2019] for DRL training, and the *scenario-based programming* framework (SBP) [Damm and Harel, 2001; Harel et al., 2012b] for the encoding of

user-defined constraints. Scenario-based programming is a software engineering paradigm intended to allow engineers to create a complex system in a way aligned with how humans perceive that system. A scenario-based program is comprised of scenarios, each of which describes a single desirable (or undesirable) behavior of the system at hand. These scenarios are then combined to run simultaneously in order to produce cohesive system behavior. We show how such scenarios can be used to directly incorporate subject-matter-expert (SME) knowledge into the training process, thus forcing the resulting agent's behavior to abide by various safety, efficiency, and predictability requirements.

## 4.1 DESCRIBING THE REQUIREMENTS

In order to inject prior knowledge into the training loop, the first challenge is to find a method to describe and encode the requirements, preferably in a human-friendly way, that is differentiable and thus optimizable in the training loop. At the same time, as motivated in Chap 3, in a DRL context, we should rely on the Constrained Markov Decision Process (CMDP) framework, where an addition signal (i.e., the *cost function*) has to be minimized concurrently with the maximization of the reward function. In this section, we first present three classes of constraints before diving into the formalism of our choice: Scenario Based Programming (SBP).

- **Network Level Constraints** A possible approach to describe requirements in a DRL context is through input-output relations. For example, following the formulation proposed by Liu et al. [2019], originally designed to encode safety requirements, we obtain the following encoding:

$$\Theta : \text{If } x_0 \in [a_0, b_0] \land ... \land x_n \in [a_n, b_n] \Rightarrow y_j \in [c, d] \tag{4.1}$$

  where $x_k \in X$, with $k \in [0, n]$ (where $n$ is the number of input) and $y_j$ is a generic output. Although this approach is useful for the offline formal verification of safety properties[1], it presents a crucial limitation that prevents its application in an on-policy DRL loop: *it does not depend on the trajectory ($\tau$)* (remember that, in a DRL loop, a trajectory $\tau$ is the sequence of state and action in a single episode: $\tau = s_0, a_0, s_1, a_1, ..., s_n, a_n$). Crucially, this is true not only for properties in the form of Eq. 4.1 but also for all the *static* formulations, i.e., the formulations that depend only on the policy and do not depend on the actual behavior of the agent in the environment. To better explain why these kinds of formulations for the cost function can not be optimized by an on-policy DRL algorithm, we analyze the problem both from a theoretical and an intuitive point of view.

  Recalling the derivation of the *policy-gradient theorem* presented in Chap. 2, the objective function to maximize is $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$, and this is a foundational requirement for

[1] This topic is the focus of Part II of this dissertation.

the policy gradient theorem. In the case of the cost for a CMDP, the objective function to minimize becomes $J_c(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[C(\tau)]$. First, it is trivial that the function $C(\tau)$ depends on $\tau$, and in Eq 4.1 the parameter $\tau$ does not appear because the violation of the property depends only on the parameters of the policy (DNN). Second, expanding $J_c$ by applying the definition of expectation, and considering that the cost function does not depend on $\tau$ (e.g., Eq 4.1), we obtain that $J_c(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[K]$ where $K$ is a constant for the expectation, and then $J_c(\pi_\theta) = C$; finally, by applying the gradient, we obtain that $\nabla_\theta J_c(\pi_\theta) = \nabla_\theta C = 0$ which has clearly no meaning in this context. Even without considering the theoretical reason, there is an intuitive explanation of why the cost function must be trajectory-dependent. The intuitive idea behind the gradient optimization for a policy is the following: *if the trajectory is evaluated as good, push the policy in the right direction to increase the probability of doing the same trajectory again*, trivially, if the cost function does not depend on $\tau$, the trajectory, and thus the policy that generated it, can not be evaluated. Notice that this discussion is relevant only when applied to a gradient-based optimization method; in the previous chapter, for example, we presented SOS, a framework based on a genetic approach that is not gradient-based, and thus the cost function can be trajectory-independent.

- **State Level Constraints** An alternative approach, that is currently considered a standard in the literature [Achiam et al., 2017; Marchesini et al., 2020] is to use a cost function based on the current state (i.e., $C(s)$). Typically $C(s)$ is an indicator function that assumes the value 1 if the state is *undesired* and 0 otherwise, summing up this value over the whole trajectory ($\tau$), it returns a value that represents the number of violations that the policy generated executing $\tau$. In contrast to the previous example, a cost function formulated in this way respects the requirements of the policy gradient theorem and can be optimized through gradient descent to train an agent. Although this approach is sound and shows promising results in complex environments [Ray et al., 2019], its expressiveness is limited. A cost function encoded through this state-dependent approach is particularly effective in limiting the agent to reach *dangerous* states but can not control the complex *behavior* of the agent (for example, state/action pairs or sequences), which is a key requirement for our prior-knowledge injection purposes.

- **Trajectory Level Constraints** To overcome these limitations, in the following sections, we propose a novel approach for the encoding of cost functions. From a high-level point of view, the intuition is to use a Finite-State Machine (FSM) that encodes the properties and runs in parallel to the real loop of the agent. The states of the FSM are not necessarily the same states of the CMDP but can represent combinations of states, actions, and sequences of them. The FSM execution does not influence the behavior of the agent in the training loop, and thus it does not compromise the exploration phase. We then increase the cost signal by a unit if the state of the FSM is a *forbidden* state. This is clearly trajectory-dependent and, at the same time, allows the characterization of the behavior. Although

an FSM can encode constraints with combinations of states and actions and can thus characterize a complete desired or undesired behavior. However, combining together the executions of different FSM can be hard, making it difficult to describe a sequence of desired behaviors. In this direction, we propose to extend this concept to more sophisticated languages, such as scenario-based programming (SBP), which we introduce in the next sections as a fundamental component of our novel approach.

## 4.2 LAGRANGIAN REINFORCEMENT LEARNING

In this chapter, we present the idea behind a Lagrangian Reinforcement Learning algorithm, an approach to solving a CMDP that exploits the theoretical concept of Lagrangian relaxation for constrained optimization problems. Following the definition from Chap. 3, we define a CMDP as follows:

$$\Pi_{\mathcal{C}} := \{\pi \in \Pi : J_{C_i}(\pi) \leq h_i,\ \forall i\} \qquad \pi^* = \arg\max_{\pi \in \Pi_{\mathcal{C}}} J_R(\pi). \tag{4.2}$$

changing the order of the elements we redefine the problem as a standard constrained optimization problem (we consider the single constraint problem for simplicity, but the derivation can be easily extended to a multiple constraints version):

$$\begin{aligned} \max_{\pi} \quad & J_R(\pi) \\ \text{s.t.} \quad & J_C(\pi) < h \end{aligned} \tag{4.3}$$

solving this equation with a gradient-based (i.e., numerical) approach is hard and, for complex cases, even not possible. For this reason, we exploit the concept of lagrangian relaxation, following the definition from Chap. 3, to relax the problem to the following unconstrained one:

$$\max_{\pi} \min_{\lambda \geq 0} \mathcal{L}(\pi, \lambda) \tag{4.4}$$

finally, by the definition of lagrangian function, we obtain:

$$\max_{\pi} \min_{\lambda \geq 0} J_R(\pi) - \lambda(J_C(\pi) - h) \tag{4.5}$$

To find a solution to this problem in different recent works, such as Roy et al. [2021], the authors have proposed to iteratively take a gradient ascent step of the *max min* problem in the variable $\pi$ and a gradient descent one in $\lambda$. The first step is to take a gradient descent step on the lagrangian multiplier $\lambda$ (notice that the first part of Eq. 4.5 does not depend on the variable $\lambda$ and cancels out) by applying the definition of gradient, we formally obtain:

$$\nabla_{\lambda} \mathcal{L}(\pi, \lambda) = h - J_C(\pi) \tag{4.6}$$

recalling that the update must respect the constraint $\lambda \geq 0$, we obtain two possible cases: (i) if the property is violated (i.e., $J_C(\pi) \geq h$), the value of $h - J_C(\pi)$ is positive, and then the gradient

descent step increase the value of *lambda*; or (ii) if the property is respected (i.e., $J_C(\pi) < h$), the value of $\lambda$ is decreased until reaching 0.

The second step is the maximization over the policy $\pi$, notice that both $J_C(\pi)$ and $J_R(\pi)$ depend on the policy, while the term $(\lambda \cdot h)$ cancels out because it does not depend on the variable $\pi$; more formally:

$$\nabla_\pi \mathcal{L}(\pi, \lambda) = \nabla_\pi (J_R(\pi) - \lambda J_C(\pi)) \tag{4.7}$$

The gradient ascent step maximizes the function $J_R(\pi) - \lambda J_C(\pi)$, by simultaniously increasing the positive element $J_R(\pi)$ and decreasing the negative one $J_C(\pi)$. However, the magnitude depends on the value of $\lambda$. Considering the gradient descent step on $\lambda$, if the property is respected (i.e., the system respects the requirements) $\lambda$ is reduced, and the gradient ascent step focuses more on the maximization of the reward; in the opposite case when the property is violated $\lambda$ is increased, and the gradient ascent step focuses more on the minimization of the cost. The borderline case is when $\lambda = 0$, which means that the cost is below the given threshold $h$ and the optimizer focuses exclusively on the reward maximization. Finally, for the gradient ascent step, any existing unconstrained method can be applied. A state-of-the-art solution is to exploit the Proximal Policy Optimization algorithm (PPO) [Schulman et al., 2017], which is often referred to as Lagrangian-PPO in this version [Ray et al., 2019].

## 4.3 COMBINING FORMAL LANGUAGES AND LAGRANGIAN DRL

In the previous sections, we discussed the problem of defining the requirements and how we can exploit the concept of constrained optimization and lagrangian relaxation in the contexts of DRL. In this section, we show how we address both these problems, presenting a novel framework that we refer to as *SBP Lagrangian* [Corsi et al., 2022]. Our approach exploits scenario-based programming for the definition of the requirements, and an optimized version of the Lagrangian PPO algorithm for the training.

### SCENARIO BASED MODELING

Scenario-based programming (SBP) [Damm and Harel, 2001; Harel and Marelly, 2003] is a paradigm designed to facilitate the development of reactive systems, by allowing engineers to program a system in a way that is close to how it is perceived by humans, with a focus on inter-object, system-wide behaviors. In SBP, a system is composed of *scenarios*, each describing a single, desired, or undesired behavioral aspect of the system; and these scenarios are then executed in unison as a cohesive system.

The execution of a scenario-based (SB) program is formalized as a discrete sequence of events. At each time step, the scenarios synchronize with each other to determine the next event to be triggered. Each scenario declares events that it *requests* and events that it *blocks*,

corresponding to desirable and undesirable (forbidden) behaviors from its perspective; and also events that it passively *waits-for*. After making these declarations, the scenarios are temporarily suspended, and an *event-selection mechanism* triggers a single event that was requested by at least one scenario and blocked by none. Scenarios that requested or waited for the triggered event wake up, perform local actions, and then synchronize again; and the process is repeated ad infinitum. The resulting execution thus complies with the requirements and constraints of each of the individual scenarios [Harel and Marelly, 2003; Harel et al., 2012b]. For a formal definition of SBP, see [Harel et al., 2012b].

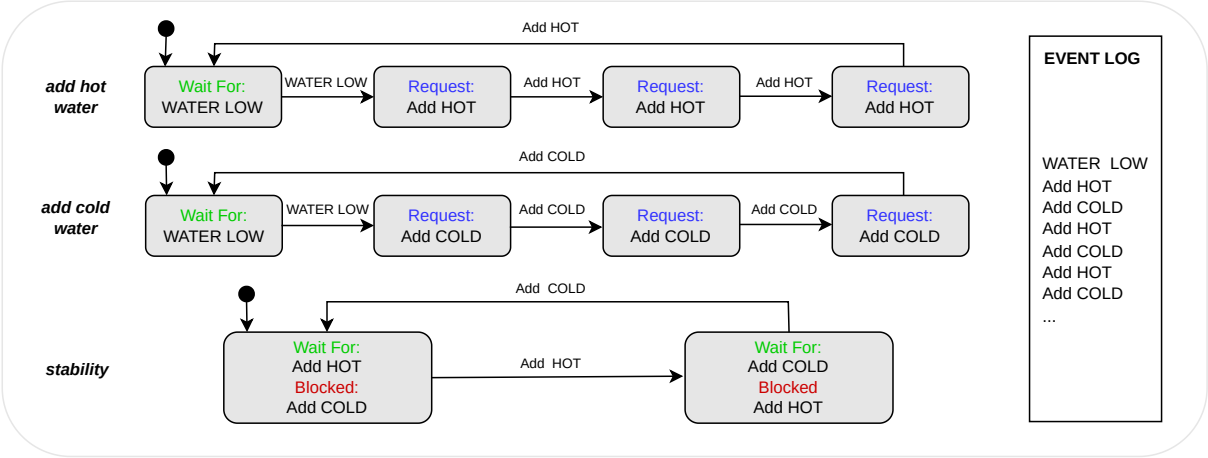



Figure 4.1: The state transition graphs represent the scenarios of a scenario-based program for controlling a water tank. The *add hot water* and *add cold water* scenarios wait in their initial state for a `WATER LOW` event. Once `WATER LOW` is triggered, they each move to their next state, requesting `Add Cold` and `Add Hot` events, respectively. The *stability* scenario waits in its initial state for a `Add Hot` event, while blocking `Add Cold` events. Once an `Add Hot` event is triggered, the scenario transitions to its second state, where it blocks `Add Hot` events while waiting for an `Add Cold` event. Once an `Add Cold` event is triggered, the scenario transitions back to its initial state, in which it waits for an `Add Hot` event while blocking `Add Cold` events.

Although SBP is implemented in many high-level languages, it is often convenient to think of scenarios as transition systems, where each state corresponds to a synchronization point, and each edge corresponds to an event that could be triggered. Fig. 4.1 uses that representation to depict a simple SB program that controls the temperature and water level in a water tank (borrowed from [Harel et al., 2012a]). The scenarios *add hot water* and *add cold water* repeatedly wait for `WATER LOW` event, and then request three times the event `Add HOT` or `Add COLD`, respectively. Since these six events may be triggered in any order by the event selection mechanism, a new scenario *stability* is added to keep the water temperature stable, achieved by alternately blocking `Add HOT` and `Add COLD` events. The resulting execution trace is shown in the event log.

While a python-like implementation code of this program, with the scenarios *add hot water*, *add cold water* and *stability*, appears in Listing 4.1. SBP is an attractive choice for the incorporation of domain-specific knowledge into a DRL agent training process, due to being formal, fully executable, and support of incremental development [Gordon et al., 2012; Alexandron et al., 2014]. Moreover, the language it uses enables domain-specific experts to directly express their requirements specifications as an SB program.

```
def add_hot_water():
        while True:
                yield {waitFor: BEvent("WATER_LOW")}
                yield {request: BEvent("Add␣HOT")}
                yield {request: BEvent("Add␣HOT")}
                yield {request: BEvent("Add␣HOT")}

def add_cold_water():
        while True:
                yield {waitFor: BEvent("WATER_LOW")}
                yield {request: BEvent("Add␣COLD")}
                yield {request: BEvent("Add␣COLD")}
                yield {request: BEvent("Add␣COLD")}

def stability():
        while True:
                yield {waitFor: BEvent("Add␣HOT"), block: BEvent("Add␣COLD")}
                yield {waitFor: BEvent("Add␣COLD"), block: BEvent("Add␣HOT")}
```

Listing 4.1: The Python implementation of the three scenarios: *add hot water*, *add cold water*, and *stability*. The code will run until it reaches a synchronization point, indicated by a *yield* statement, where it will stop and declare events it waits for, requests, and blocks. Once an event that the scenario requested or waited for is triggered, the run-time engine will resume the scenario's execution, and the code will run until its next synchronization point and repeat.

### FROM RULES TO CONSTRAINED DRL

Even after defining constraints as an SB program, obtaining a differentiable function for the training process is not straightforward. To this end, we propose to use the following binary (indicator) function:

$$c_k(s_t, a, s_{t+1}) = I(\text{the tuple } \langle s_t, a, s_{t+1} \rangle \text{ is a blocked state in the SB program, by the } k^{th} \text{ rule}) \tag{4.8}$$

Intuitively, summing the values of the different $c_k$'s over the training episode yields the exact number of violations to the respective $k^{th}$ rule during the full trajectory; those are the values we aim to minimize; moreover, following the intuition of Roy et al. [2021], this value, if normalized

over the number of steps, can be seen as a probability of having a violation. This value can be treated as a cost function, and the corresponding objective function is defined as follows:

$$J_{C_k} = \sum_{N} c(s_i, a_i, s_{i+1})$$

for a trajectory of $N$ steps. This value is dependent on the action policy $a$ and is therefore differentiable on the parameters $\theta$ of the policy through the *policy gradient theorem*.

**State Space Expansion.** In the previous sections, we motivated the adoption of a trajectory-based encoding for properties as it effectively characterizes intricate behaviors and their combinations. To achieve this objective, it is crucial to combine actions and states in a time-dependent manner. For instance, if we seek to encode the behavior "never turn left four times consecutively" for a robotic drone, we can easily implement it using a scenario-based property (SBP). However, a naive encoding of the state may lead to a violation of the Markov properties of the MDP, where the next state and reward are independent on previous actions and states given the current state and action.

Considering the above example, assume that the robot is in state $s_0$ after three left actions, and we calculate the cost function using Eq. 4.8, which yields $C_{s_0} = 0$. In contrast, if the robot is in the same state after four left actions, it obtains $C_{s_0} = 1$. Although this problem does not always hinder the policy from achieving a good local maximum, such a setting does not respect a key assumption to guarantee the convergence of the training [Sutton and Barto, 2018], potentially leading to instability in the learning process. In a CDRL context, this issue becomes evident for our actor-critic setup, where we may update the critic with $Q(s_0, \mathit{left}) = 0$ in some cases and $Q(s_0, \mathit{left}) = 1$ in others, leading to brittle convergence properties for the Q-function. To address this limitation, we propose enriching the state of the critic by incorporating the internal state of the SBP, i.e., adding the current internal state of the Finite State Machine (FSM) as an input for the network, formally expressed as:

$$Q(S^*, \mathit{left}) \quad s.t. \quad S^* = \langle S, S_i \rangle$$

here, $S$ represents the standard state of the MDP, and $S_i$ represents the internal state of the FSM or SBP. It is noteworthy that these modifications affect only the critic and can be eliminated after the training loop. This feature is critical because it does not increase the space complexity of the actor or policy network and does not require computing the scenarios during inference.

**Optimized Lagrangian-PPO.** In this chapter, we proposed to relax the Lagrangian constrained optimization problem into an unconstrained, *min-max* version thereof. Taking the gradient of Equation 4.5, and some algebraic manipulation, we derive the following two simultaneous problems:

$$\nabla_\theta \mathcal{L}(\pi, \lambda) = \nabla_\theta (J_R(\pi) - \sum_{K} \lambda_k J_{C_k}(\pi)) \qquad \forall k, \quad \nabla_{\lambda_k} \mathcal{L}(\pi, \lambda) = -(J_{C_k}(\pi) - d_k) \tag{4.9}$$

In closed form, the Lagrangian dual problem would produce exact results. However, when applied using a numerical method like *gradient descent*, it has shown strong instability and the proclivity to optimize only the cost, limiting the exploration and resulting in a poorly-performing agent [Achiam et al., 2017]. To overcome these problems, we introduce three key optimizations that proved crucial to obtaining the results we present in the next section.

1. *Reward Multiplier*: The standard update rule for the policy in a Lagrangian method is given in Equation 4.9. However, as mentioned above, it often fails to maximize the reward. To overcome this failure, we introduce a new parameter $\alpha$, which we term *reward multiplier*, such that $\alpha \geq \sum_K \lambda_k$. This parameter is used as a multiplier for the reward objective:

$$\nabla_\theta \mathcal{L}(\pi, \lambda) = \nabla_\theta(\alpha \cdot J_R(\pi) - \sum_K \lambda_k J_{C_k}(\pi)) \tag{4.10}$$

2. *Lambda Bounds and Normalization*: Theoretically, the only constraint on the Lagrangian multipliers is that they are non-negative. However, when solving numerically, the value of $\lambda_k$ can increase quickly during the early stages of the training, causing the optimizer to focus primarily on the cost functions (Eq. 4.9), potentially not pushing the policy towards a high-performance reward-wise. To overcome this, we introduced dynamic constraints on the multipliers (including the reward multiplier $\alpha$), such that $\sum_K \lambda_k + \alpha = 1$. In order to also enforce the previously mentioned upper bound for $\alpha$, we clipped the values of the multipliers such that $\sum_K \lambda_k \leq \frac{1}{2}$. Formally, we perform the following normalization over all the multipliers:

$$\forall k, \;\; \lambda_k = \frac{\tilde{\lambda}_k}{2(\sum_K \tilde{\lambda}_k)} \qquad\qquad \alpha = 1 - \sum_K \lambda_k \tag{4.11}$$

3. *Algorithmic Implementation*: The primary objective of the previously introduced optimizations is to balance the learning between the reward and the constraints. To further stabilize the training, we introduce additional, minor improvements to the algorithm: (i) *lambda initialization:* we initialize all the Lagrangian multipliers with zero to guarantee a focus on the reward optimization during the early stages of the training (consequently, following Eq. 4.11, $\alpha = 1$); (ii) *lambda learning rate:* to guarantee a smoother update of the Lagrangian multipliers, we scale this parameter to 10% of the learning rate used for the policy update; and (iii) *delayed start:* we enable the update of the multipliers only when the success rate is above 60% during the last 100 episodes. Intuitively, this delays the optimization of the cost functions until a minimum performance threshold is reached.

**Additional Hyperparamters.** In the preceding paragraph, we presented three algorithmic implementations to enhance the stability of a standard Lagrangian-PPO algorithm. However, these methods introduce the need for additional hyperparameters that are not always easy to

tune. In our discussion on the *Lambda Bounds and Normalization*, we encoded the normalization rule such that $\alpha \geq 0.5$, which yielded favorable results in our benchmarking setup. However, we cannot exclude the possibility that in other benchmarking environments, this value may need to be modified. To provide a more general formulation, we introduce an additional parameter, denoted as $min_\alpha$, which satisfies:

$$\sum_K \lambda_k JC_k(\pi)) \leq min_\alpha$$

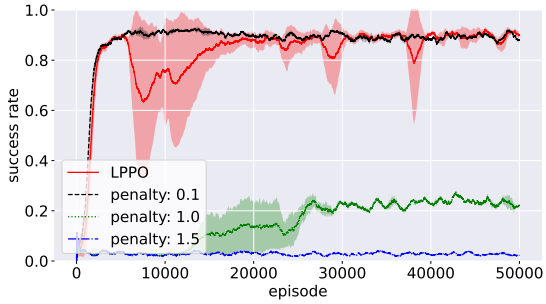
this parameter enables to balance the focus of the optimization between the reward and the cost functions, however, it can be hard to tune and heavily depends on the scale of these two signals. Another parameter is related to our proposed *delayed start* approach. In our configuration, we start the training of the lagrangian multiplier only after a fixed period of time (i.e., after the success rate reaches a value of 60%), however, this threshold is specifically tuned to our benchmarks and can not be uniformly applied to all environments. To generalize this formulation, we introduce an additional parameter called *lambda-start* ($\lambda_s$), which serves as an indicator for when the value of the Lagrangian multiplier can be increased. Although our experiments have shown that $min_\alpha$ and $\lambda_s$ are easy to tune, the introduction of these parameters can be viewed as a limitation of our method. As a future direction, we plan to investigate how to learn these values as parameters for the optimization process, in a manner similar to how the Lagrangian multiplier ($\lambda$) is trained within the constrained reinforcement learning loop.

## 4.4 RESULTS

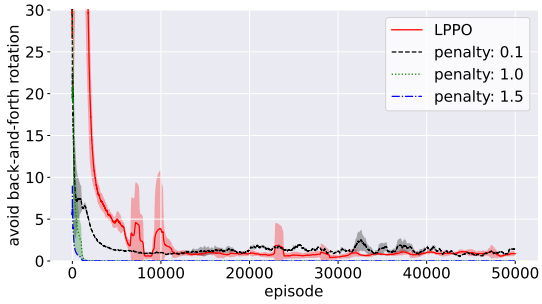
We performed training on a distributed cluster of HP EliteDesk machines, running at 3.00 GHz, with 32 GB RAM. We collected data from more than 100 seeds for each algorithm, reporting the mean and standard deviation for each learning curve, following the guidelines of Cédric et al. [2019]. For training purposes, we built a realistic simulator based on the *Unity3D* engine [Juliani et al., 2018], which we used for the comparison with the penalty-based approach; Our ablation studies are based on *Bullet Safety Gym* [Gronauer, 2021].

**SBP Lagrangian vs Penalty-Based** To the best of our knowledge, this is the first work that combines scenario-based programming into the training of a constrained deep reinforcement learning system. However, to show our results, we proposed a comparison with the work of Yerushalmi et al. [2022], the authors proposed integration between SBP and DRL using a reward-shaping (or penalty-based) approach that penalizes the agent when rules are violated, with an unconstrained optimization method. Our approach, based on constrained optimization, provides many advantages compared to the mentioned work, which results in high-performing agents and fewer rule violations. We provide an extensive comparison between the two approaches below. The study has been performed on two rules, formalized through SBP, the definition of the rules

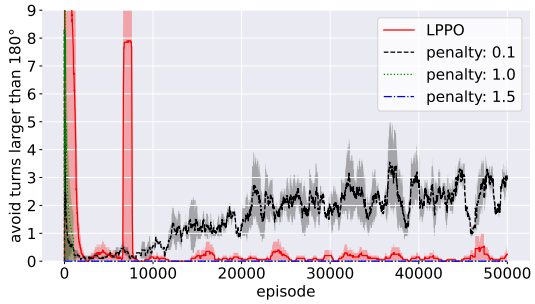
is out of scope for this comparison, however, more details can be found in the original paper [Corsi et al., 2022] and in Chap. 8.




(a) success rate




(b) avoid back-and-forth rotation




(c) avoid turns larger than 180°

Figure 4.2: The graphs compare results achieved by our approach, denoted by *LPPO*, with those achieved by [Yerushalmi et al., 2022], denoted by *penalty* and its value: fixed penalty of 0.1, 1.0, or 1.5: graph (a) measures the success rates with all three scenario-based rules. The results of using the *penalty* approach with a penalty value of 0.1 are practically the same as using our approach. However, using the *penalty* approach with penalty values of 1.0 and 1.5 results in poor performance; graph (b) measures the frequency of violations to the *first* rule. The results of using the *penalty* approach with penalty values of 0.1 and 1.0 are similar to ours. When using the *penalty* approach with a penalty value of 1.5, the violations diminish completely. However, also the performance, as mentioned above; graph (c) measures the frequency of violations to the *second* rule. The results of using the *penalty* approach with penalty values of 1.0 and 1.5 are practically the same as ours.

Fig. 4.2 compares the results of our approach and those of Yerushalmi et al. [2022]. As shown in Fig. 4.2 using their approach, a low penalty value allows the agent to reach high-performance reward-wise but fails to minimize the cost (e.g., the number of rule violations). In contrast, a high penalty value reduces the agent's rule violations but fails to reach adequate performance in terms of the reward function. Our approach is shown here to reach similar performances as the best from Yerushalmi et al. [2022], using a penalty value of 0.1, and reducing the agent's rule violations as the best of it, using a penalty value of 1.0 or 1.5. Our approach adopts a

constraint-driven DRL framework that differentiates between optimizing the main reward and minimizing the costs. This differentiation presents significant advantages, including:

- Allows the setting of constraint thresholds independently for each rule/property and the handling of multiple such constraints in the same way, unlike methods such as [Yerushalmi et al., 2022] that only allow a global minimization to zero of the total cost.
- Separates reward maximization from cost minimization, simplifying the reward engineering task.
- Automatically balances the focus of the training, between the different cost elements and the reward, by learning the values of the different multipliers for each cost factor.
- Introduces novel numerical optimizations to the training phase, resulting in a more stable algorithm with a higher cumulative reward on a synthetic set of benchmarking environments).

**Optimized Lagrangian** To further validate our method, we provide a more intensive study of our optimized implementation of the Lagrangian PPO. We perform our analysis on the standard benchmark *Bullet Safety Gym* [Gronauer, 2021]. Bullet Safety Gym is an open source suite of different environments based on *PyBullet* and the most updated versions of Python, which implements the standard environments from *SafetyGym* [Ray et al., 2019], adding more robots and tasks. The crucial feature of the environments from Bullet Safety Gym is that they implement a Constrained Markov Decision Process (CMDP). In these environments, the objective is to maximize the reward function and maintain a cost function below a given threshold. We refer to the main paper from Ray et al. [2019] for more details. We selected a subset of environments for our analysis: *Ball Circle v0*, *Ball Reach v0* and *Car Reach v0*. A detailed description of the environments can be found in the open source repository of Bullet Safety Gym [Gronauer, 2021]. Fig. 4.3 shows our results on the three environments, comparing cost and reward functions obtained with (i) the standard PPO algorithm (*PPO*); (ii) a standard Lagrangian PPO (*Base-LPPO*); and (iii) our optimized Lagrangian PPO (*LPPO*) described in this chapter.

Overall, these results show that our algorithm can get similar performance reward-wise as *PPO*, and can get the cost below the required threshold in two out of three test cases. However, in Fig. 4.3(a), on the *Ball Circle v0* environment, *PPO* reaches significantly better reward-wise performance than ours. Moreover, in Fig. 4.3(f), in the *Car Reach v0* environment, our algorithm fails to get the cost below the required threshold. We leave it for future work, to study the impact of adding expert-knowledge rules with our optimizations, and if those will enable obtaining good performance reward-wise and cost-wise, also in those cases, for the *Ball Circle v0* environment reward, shown in Fig. 4.3(a), and the *Car Reach v0* environment cost, shown in Fig. 4.3(f).

In summarizing, Fig. 4.3 shows that, not surprisingly, the standard PPO reaches good performance reward-wise but can not optimize the cost. A naïve implementation of the Lagrangian PPO can minimize the cost function but struggles to obtain good performance reward-wise. Our optimized approach (without SBP rules) is the only one that, at the same time, succeeds


(a) Ball Circle v0 - Reward

(b) Ball Reach v0 - Reward

(c) Car Reach v0 - Reward

(d) Ball Circle v0 - Cost

(e) Ball Reach v0 - Cost

(f) Car Reach v0 - Cost

Figure 4.3: A comparison between the original PPO [Schulman et al., 2017] (*PPO*), a standard implementation of Lagrangian PPO [Ray et al., 2019] (*Base-LPPO*), and our optimized version of the Lagrangian PPO (*LPPO*). The analysis is performed on the standard benchmark *Bullet Safety Gym* [Gronauer, 2021], and in particular on three environments of the suite (*Ball Circle v0*, *Ball Reach v0* and *Car Reach v0*).

in reducing the cost under the given threshold in two out of three cases while reaching good performance reward-wise. We are confident that we can define SBP rules that will help to reduce those costs as well as further improve the reward.

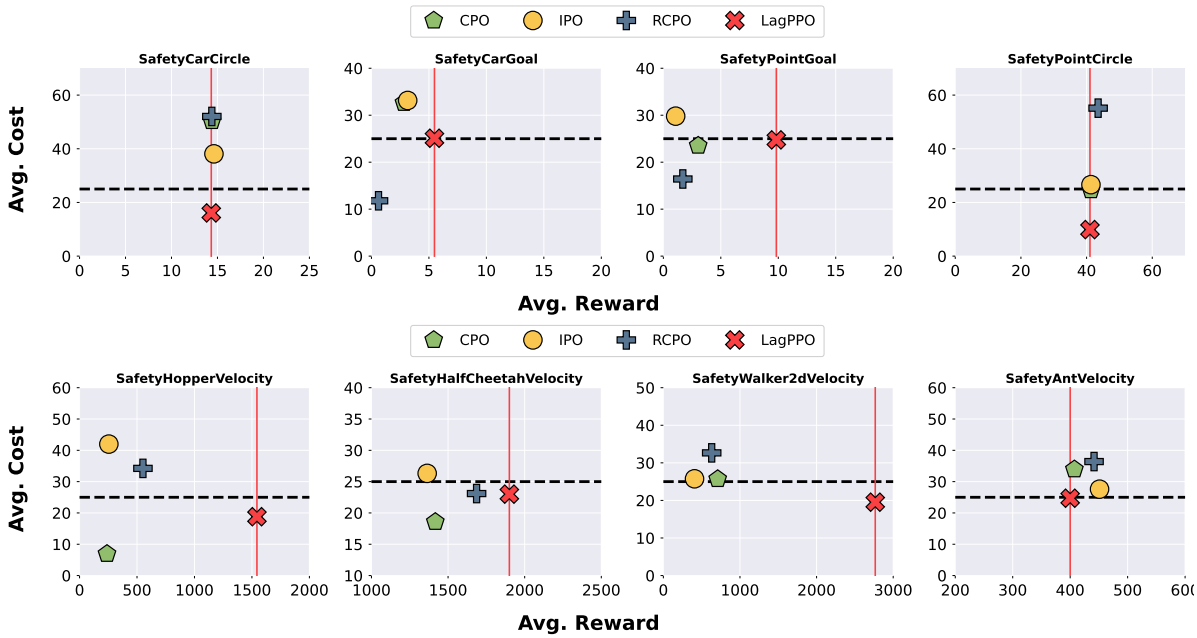



Figure 4.4: A comparison between our improved version of the LagrangianPPO and alternative approaches for a CMDP; the analysis is performed on the standard benchmark Safety Gymnasium [Yang, 2023]. The first row shows a comparison of four safe navigation problems (i.e., *SafePointGoal1-v0*, *SafeCarGoal1-v0*, *SafePointCircle1-v0*, and *SafeCarCircle1-v0*); while the second row shows the comparison on four locomotion benchmarks based on MuJoCo (i.e., *SafetyHopperVelocity-v1*, *SafetyHalfCheetahVelocity-v1*, *SafetyWalker2dVelocity-v1*, and *SafetyAntVelocity-v1*).

**Comparison with Non-Lagrangian Methods** In the previous paragraph, we provided an ablation study to show the superior performance of our improved version of Lagrangian PPO against the naïve implementation presented by Ray et al. [2019]. In this paragraph, in contrast, we compare our algorithm against a set of non-lagrangian baselines, including CPO, RCPO, and IPO (we refer to Chap. 3 of this dissertation for a detailed description of these methodologies). We perform the analysis exploiting a set of environments from SafetyGymnasium [Yang, 2023], an open-source version of the standard SafetyGym that we already discussed in the previous sections. In particular, we select six different robots and three tasks, obtaining a combination of 8 benchmarking environments. In particular, we focus on two families of problems that provide a good overview of the tasks presented in SafetyGymnasium. With *SafePointGoal1*,

*SafeCarGoal1*, *SafePointCircle1*, and *SafeCarCircle1* the objective of our analysis is to show the performance on a complex task, even if the dynamic of the robot is simple to control; in contrast, in *SafetyHopperVelocity*, *SafetyHalfCheetahVelocity*, *SafetyWalker2dVelocity*, and *SafetyAntVelocity* the task is simpler (i.e., move as fast as possible respecting some constraints on the velocity for the joints), while the dynamic of the robot is more complex and hard to control.

The experimental outcomes are presented in Fig.4.4, where we report, for each algorithm, the performance results after $1M$ of steps in the environment, averaged over five different seeds. The choice of showing the results at convergence time follows our intuition presented in Sec.1. For our training pipeline, the constrained DRL is followed by a verification process and the objective of the first phase is, in fact, to provide the best possible model to the verification engine. We are thus interested in the performance at convergence and not averaged over the training. In Fig.4.4, the black dotted line denotes the cost threshold, while the red line highlights the reward obtained with our proposed algorithm (i.e., Optimized Lagrangian PPO); on the x-axis we report the reward, while on the y-axis the cumulative cost. From our perspective, the most successful algorithm is the one that achieves the highest reward while remaining below this given threshold. Our results reveal that, overall, our algorithm achieves better performance over the baseline. In some cases (e.g., SafetyAntVelocity), our lagrangian algorithm obtains lower performance reward-wise but is the only one that respects the cost constraints and, in a safety-critical context, should be considered the best choice.

**Related Work and Future Direction** In this chapter, we showed the promising results of our approach with a set of experiments on different benchmarking environments. However, our method suffers from some limitations, that we plan to address as part of future directions. First, it does not provide formal guarantees that the resulting policies are safe. Second, the scalability of the method on multiple constraints needs to be investigated. We showed in this work that the algorithm can easily handle one to three constraints, in addition to the main objective. We leave to future work the analysis of performance when the number of constraints increases further. Third, we noticed some performance deterioration after about 10,000 episodes. We believe that the performance deterioration is related to the activation of the cost multipliers, especially as the performance was recovered afterward. An alternative approach is presented in the recent work of Roy et al. [2021], where the authors advocate an optimized version of Lagrangian-PPO. They propose a different approach to balance the constraints and the return, based on the softmax activation function and without imposing bounds on the values for the multipliers. Moreover, their work focuses on game development, a different domain from our focus, which presents very different challenges, e.g., safety and efficiency are not considered crucial requirements. In addition, they do not encode constraints using a framework geared for this purpose, such as SBP, and are thus not suitable for a direct comparison.

Moving forward, we plan to extend our work to complex environments including navigation

in complex domains (e.g., air and water). Another key challenge for the future is to inject rules aiming to encode behaviors in a cooperative (or competitive) multi-agent environment. Finally, we plan to exploit the concept of rules and cost function to inject different kinds of prior knowledge, including human demonstrations, e.g., moving towards an imitation learning framework.

# Part II

# Formal Verification for Deep Reinforcement Learning

# Chapter 5

# THE VERIFICATION PROBLEM

Throughout this thesis, we showed that *Deep Neural Networks* (DNNs) have become extremely popular, achieving state-of-the-art results in a wide variety of fields; popular examples come from computer vision [Simonyan and Zisserman, 2014], financial operations control [Meng and Khushi, 2019] and games playing [Mnih et al., 2013]. In Chapter 2, we also highlighted that, in recent years, *Deep Reinforcement Learning* (DRL) has also been applied in several safety-critical domains, such as robotics [Marchesini and Farinelli, 2020] and healthcare [Pore et al., 2022]. Unfortunately, despite the widespread success of DNNs, they have been shown to suffer from various safety issues [Katz et al., 2017; Szegedy et al., 2013]. In the first part of this dissertation, we focused on the training of the models, trying to inject a set of requirements directly during the training process. We now face the problem from a different and more formal perspective: *verification and validation*.

## 5.1 PROBLEM FORMALIZATION

A well-known issue that prevents the application of DNNs in safety-critical contexts is related to the *adversarial configurations*, i.e., a small perturbation to the inputs, which can be either intentional (e.g. malicious attacks) or the result of noise (e.g., robotic sensors), that may cause DNNs to react in unexpected ways [Moosavi-Dezfooli et al., 2017]. These inherent weaknesses are observed in almost every kind of neural network and indicate a need for techniques that can supply formal guarantees regarding the safety of the DNN in question. Specifically, related to the focus of this dissertation, these weaknesses have also been observed in DRL systems [Eliyahu et al., 2021; Kazak et al., 2019; Amir et al., 2021], showing that even state-of-the-art DRL models may err miserably in some specific corner cases. These particular input configurations are challenging to detect through only an empirical testing phase and consequently ignored by the standard metrics. This underly the limits of the traditional evaluation approaches and

highlights the need for more formal methods. To mitigate such safety issues, the verification community has recently developed a plethora of techniques and tools [Katz et al., 2017; Gehr et al., 2018; Wang et al., 2018b; Lyu et al., 2020; Lomuscio and Maganti, 2017] for formally verifying that a deep neural network (DNN) is provable safe before the deployment.

*Formal verification* (FV) of deep neural networks can be addressed in many ways. Typically, a verification framework checks whether an input-output relation (i.e., *safety property*) respects some given constraints [Liu et al., 2019]. Given a neural network function $N_\theta(x)$ (with parameters $\theta$) with an input domain $\mathcal{D}_x \subseteq \mathbb{R}^{k_i}$ and output domain $\mathcal{D}_y \subseteq \mathbb{R}^{k_o}$, where $k_i$ is the number of input nodes and $k_o$ the number of output nodes, solving the verification problem requires to formally show that a property in the following form holds:

$$\mathbf{x} \in \mathcal{X} \Rightarrow \mathbf{y} = N_\theta(\mathbf{x}) \in \mathcal{Y} \tag{5.1}$$

where, $\mathcal{X} \subseteq \mathcal{D}_x$ and $\mathcal{Y} \subseteq \mathcal{D}_y$. The input set $\mathcal{X}$ can have different geometries, a common representation is based on *hyperrectangle*, which corresponds to a multi-dimensional rectangle with a defined center $c \in \mathbb{R}^{k_i}$ and $r_0 \in \mathbb{R}^{k_i}$:

$$\mathcal{X}_h = \{\mathbf{x} : \|\mathbf{x} - c\|_2 \leq r_0\} \subseteq \mathcal{D}_\mathbf{x} \tag{5.2}$$

More in general the input domain of a safety property is represented with polytopes, defined as *halfspace-polytopes* in the following form:

$$\mathcal{X}_p = Cx \leq d \tag{5.3}$$

where $C \in \mathbb{R}^{k \times k_i}$, $d \in \mathbb{R}^k$ and $k$ is the number of inequalities that define the polytope. Commonly, the desired output for an FV problem is a single counterexample, a specific input configuration that violates the relation of Eq. 5.1, or a formal proof that this output does not exist. However, there is an alternative version of the FV problem, that we do not discuss in this thesis (we refer to the work of Casadio et al. [2022] for more details), that aims at verifying the *robustness* of a network, i.e., the tolerance to adversarial input configuration that causes a misclassification in the output [Goodfellow et al., 2015], in this last case, the objective of the verification is to find the maximum disturbance around an input point ($x_0$) that does not change the output label (expressed as an $\epsilon$-ball around $x_0$). Starting from this general definition, we introduce from the literature two more specific formulations. It is possible to show the equivalence between them, however, our objective is to remark that FV can be applied in different contexts to reach different objectives. We exploit these two additional formulations of the problem, respectively, in Chapter 6 and Chapter 7.

**Minimization Formulation [Weng et al., 2018]** This formulation reduces the FV into a minimization problem. The key intuition is that a formal verification query in the standard form proposed by Liu et al. [2019], and representable with the tuple $\langle N_\theta, \mathcal{X}, \mathcal{Y} \rangle$, can be translated in

an equivalent form encoded with an alternative tuple $\langle N'_\theta, \mathcal{X} \rangle$, where $N'_\theta$ is a modified version of $N_\theta$ that includes $\mathcal{Y}$. The idea is to add an additional dummy layer to $N_\theta$, that directly encodes the property, in a way that the property holds only if all the output of $N'_\theta$ is positive. The formal verification problem can then be redefined as follows:

$$N'_\theta(\mathbf{x}) \geq 0 \;\; \forall\, \mathbf{x} \in \mathcal{X} \Rightarrow \min_{x \in \mathcal{X}} N'_\theta(\mathbf{x}) \geq 0 \tag{5.4}$$

For example, suppose $N_\theta$ has two output nodes, $y_0$ and $y_1$ and the objective is to demonstrate that $y_0 \in [4, 6]$. We create a new DNN $N'_\theta$, equivalent to $N_\theta$ with add an additional two nodes layer: $y_l = (y_0 \cdot 1 - 4) + (y_1 \cdot 0)$ and $y_u = (y_0 \cdot -1 + 6) + (y_1 \cdot 0)$. The equivalent property holds only if all the outputs of $N'_\theta$ (i.e., $y_l$ and $y_u$) are greater (or equal) than 0.

**Satisfiability Formulation [Katz et al., 2017]** To define the satisfiability formulation we must re-define first the requirements for an FV problem. A DNN verification algorithm receives as input: (i) a trained DNN $N$; (ii) a precondition $P$ on the DNN's inputs, which limits their possible assignments to inputs of interest; and (iii) a postcondition $Q$ on $N$'s output, which usually encodes the *negation* of the behavior we would like $N$ to exhibit on inputs that satisfy $P$. The verification algorithm then searches for a concrete input $x_0$ that satisfies $P(x_0) \wedge Q(N(x_0))$, and returns one of the following outputs: (i) `SAT`, along with a concrete input $x_0$ that satisfies the given constraints; or (ii) `UNSAT`, indicating that no such $x_0$ exists. When $Q$ encodes the negation of the required property, a `SAT` result indicates that the property is violated (and the returned input $x_0$ triggers a bug). In contrast, an `UNSAT` result indicates that the property holds.

## 5.2 LITERATURE REVIEW

In literature, and following the taxonomy proposed by Liu et al. [2019], verification approaches are subdivided in two different categories: (i) *optimization* approaches, that use optimization methods (e.g., linear programming [Bastani et al., 2016], mixed integer linear programming [Lomuscio and Maganti, 2017; Tjeng et al., 2018] os SMT-solver [Katz et al., 2017]) to falsify an assertion; and (ii) *reachability* approaches which, given the input domain for the property $\mathcal{X}$, try to compute the corresponding output domain (or reachable set) [Wang et al., 2018b; Weng et al., 2018].

### REACHABILITY APPROACHES

In contrast to optimization approaches, a reachability framework does not directly return `SAT` or `UNSAT`. Instead, it tries to compute the output reachable set, formally, given the neural network function $f_\theta(x)$ and the property input domain $\mathcal{X}$, the reachability set is defined as:

$$\Gamma(\mathcal{X}, f_\theta) := \{\mathbf{y} : \mathbf{y} = f_\theta(\mathbf{x}),\; \forall \mathbf{x} \in \mathcal{X}\} \tag{5.5}$$

In this scenario, a property is violated if $\Gamma(\mathcal{X}, f_\theta) \nsubseteq \mathcal{Y}$. However, even state-of-the-art approaches [Wang et al., 2018b; Weng et al., 2018] struggle to compute the exact reachable set, succeeding only in finding an *overestimation* of the real set $\tilde{\Gamma}(\mathcal{X}, f_\theta)$. Research, in recent years, has thus been focused on finding different strategies to reduce the *overestimation* and find a $\tilde{\Gamma}$ as close as possible to $\Gamma$. To better understand this class of approaches, the first element to introduce is the *propagation*.

It is hard to indicate the first work to introduce this concept, however, traditionally it has been attributed to Pulina and Tacchella [2010]. The idea is to extend the concept of *Interval Analysis* [Moore, 1963] in the field of DNNs verification, partially mitigating the computational limitation of the previous approaches, to compute the reachability set. In detail, these methods propagate layer by layer the input domain represented as one bound for each input node. Naive approaches compute the bound ($[l_{new}, u_{new}]$) for each node of the network in an independent fashion, applying node-wise the following linear mapping:

$$\begin{aligned} l_{new} &= \max(\theta, 0) \cdot l + \min(\theta, 0) \cdot u \\ u_{new} &= \max(\theta, 0) \cdot u + \min(\theta, 0) \cdot l \end{aligned} \tag{5.6}$$

adding the biases if required and propagating the obtained bound through the activation function. Fig. 5.1 shows a numerical example of this approach. The propagation has been done node-wise, following Eq. 5.6, for example, $n_1 = [(max(2,0) \cdot 0 + max(-1,0) \cdot 1) + (min(2,0) \cdot 2 + min(-1,0) \cdot 5), (max(2,0) \cdot 2 + max(-1,0) \cdot 5) + (min(2,0) \cdot 0 + min(-1,0) \cdot 1)] = [(0+0)+(0-5), (4-1)] = [-5, 3]$. For each layer, is then necessary to apply the activation function, in the example ReLU. Recalling that *ReLU(x)=max(x, 0)*, we obtain that $a_1 = [max(-5,0), max(3,0)] = [0, 3]$; notice that this last operation is valid because the activation function is monotonic, however, this is a characteristic shared with most of the common functions (e.g., ReLU, tanh, and sigmoid).

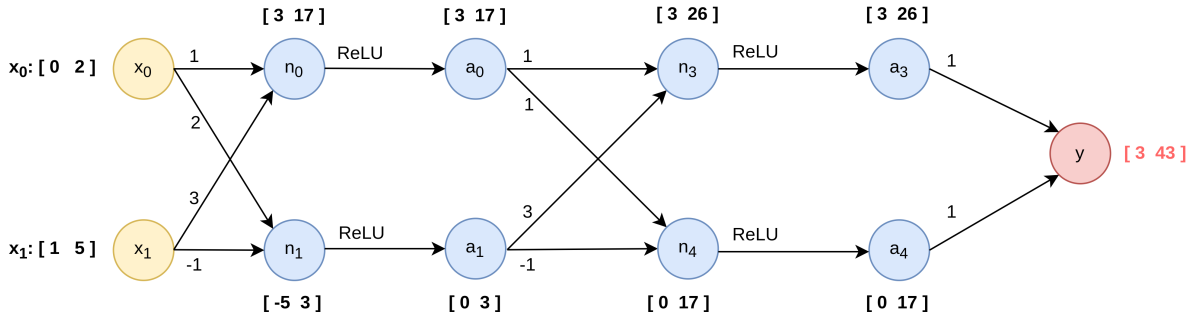



Figure 5.1: Example of the *naïve bound propagation* method. The DNN consists of two inputs, two hidden layers of two nodes with the ReLU activation function, and one output. The input domain to propagate is $x_0 : [0, 2]$ and $x_1 : [1, 5]$, after the node-wise propagation we obtain an overestimation for the output node of $y : [3, 43]$.

This method is simple and easy to implement; however, it ignores the interdependencies of

the input variables and introduces an overestimation of the bounds for each application of the activation functions (Fig. 5.4), resulting in a very loose estimation. To limit this overestimation problem, in Wang et al. [2018b] the authors introduce the *symbolic propagation* method. The intuition is to propagate the inputs symbolically as long as possible, to preserve the interdependencies between the nodes. A crucial requirement for this process is to always keep the equations linear, this is necessary for the concretization step where a Linear Programming (LP) tool is used. This is not a strong requirement for linear mapping, but can not be properly applied for the non-linear activation functions. The presented algorithm that exploits this idea is called *Reluval* and, as the name suggests, focuses on DNN with ReLU (or linear) activation functions. ReLU is a piece-wise linear function, this means that if we know the phase of the function it is possible to linearize it. The intuition is that every time ReLU must be applied, an LP query is instantiated and three outcomes are possible: (i) if the lower bound is positive it means that ReLU is in the active phase ($f(x) = x$); (ii) if the upper bound is negative it means that ReLU is in the inactive phase ($f(x) = 0$); or (iii) the phase is unknown and, only in this last case, Wang et al. [2018b] propose to concretize with the scalar values. Fig. 5.2 shows an example of this approach. Crucially, $a_1$ requires a concretization step because the lower bound $(2x_0 - x_1)$ can assume negative values in the input domain (e.g., $x_0 = 0$ and $x_1 = 5$) and the upper bound $(2x_0 - x_1)$ can assume positive values in the input domain (e.g., $x_0 = 2$ and $x_1 = 1$). In contrast, $a_0$ is in the positive linear phase of ReLU because the lower bound $(x_0 + 3x_1)$ can not assume negative values (the worst case is when $x_0 = 0$ and $x_1 = 1$ that results in a lower bound of 3), in this second case, it is possible to propagate the linear equation.

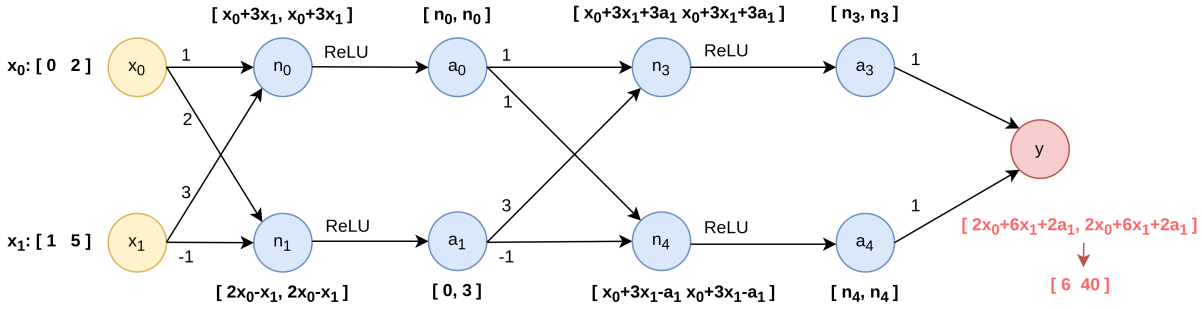



Figure 5.2: Example of the *symbolic bound propagation* method. The structure of the DNN is the same as in Fig. 5.1. After the process, we obtain an overestimation for the output node of $y : [6, 40]$, tighter than the naïve approach.

Along with the symbolic propagation, in the same paper Wang et al. [2018b] propose another improvement to reduce the overestimation: the *iterative refinement*. This technique obtains a more accurate estimation of the output bound, leveraging the fact that the dependency error for Lipschitz continuous functions decreases as the width of the interval decreases. In detail, iterative refinement subdivides the input domain into smaller *subdomains*, computing the corresponding

output bound for each of them. The union of the output bounds results in a more accurate bound of the original area. Fig. 5.3 shows an example of this approach. The input domain is split into two subdomains ($[[0,1],[1,3]]$ and $[[12],[3,5]]$), obtaining two different output bounds ($[5,23]$ and $[20,34]$), the union of these two intervals provides the final bound $[5,34]$, which is a strong improvement with respect to the previous approaches. Crucially, the *symbolic propagation* and the *iterative refinement* can be used together to further reduce the overestimation.

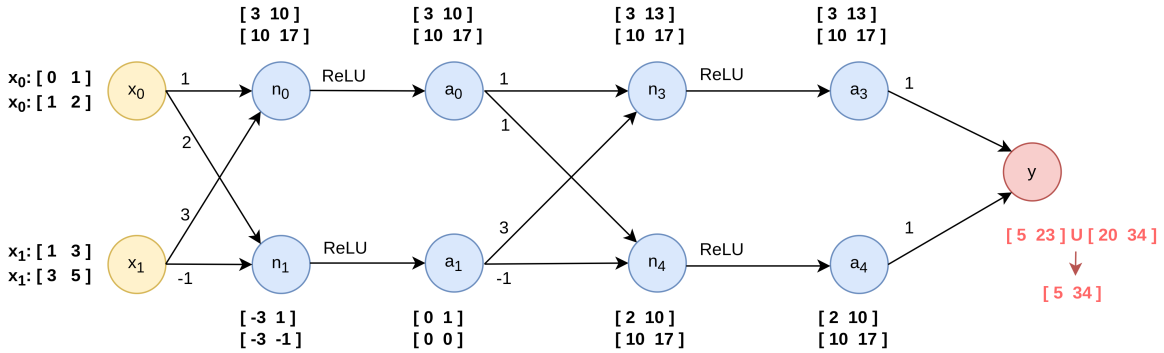
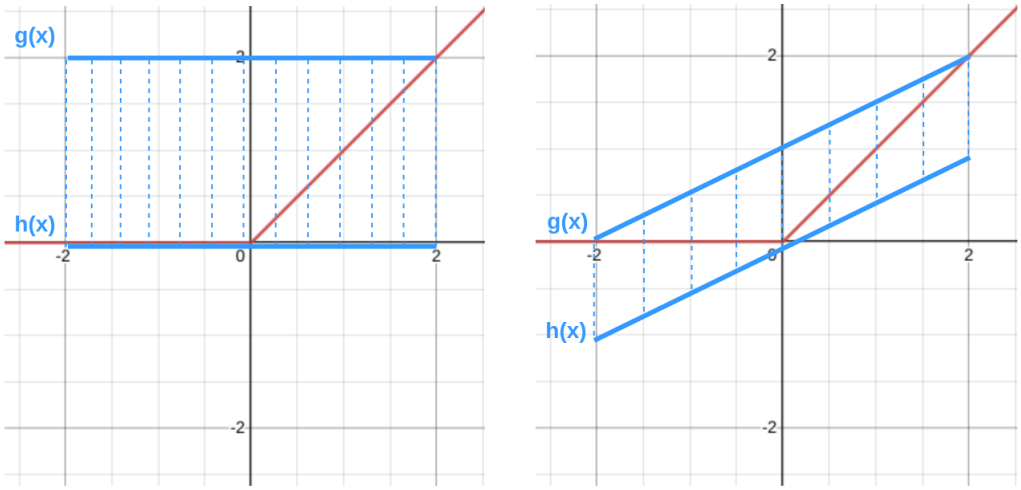



Figure 5.3: Example of the naïve bound propagation method exploiting the *(iterative) refinement*. The structure of the DNN is the same as in Fig. 5.1. After the process, we obtain an overestimation for the output node of $y$ : $[5,34]$, tighter than the naïve and the symbolic approaches. Crucially, this method can be combined with the one of Fig. 5.2 to significantly improve the final result.

Moreover, the two concurrent works from Wang et al. [2018a] and Weng et al. [2018], propose a method to overcome the limitation of the concretization step, the *linear relaxation*. Fig. 5.4 provides a high-level intuition of the approach. Suppose to have a symbolic equation $Eq^{(1)}$, limited by a generic interval $[l,u]$, to propagate through ReLU. Following the structure of Fig. 5.4, the concretization method, depicted in (a), ignores the $Eq^{(1)}$ and sets the new interval to $[min(l,0), min(u,0)]$, losing the information of $Eq^{(1)}$. In contrast, the *linear relaxation* preserves the $Eq$, allowing to maintain the interdependencies between the nodes. The post-activation bounds are visually represented in (b) with two parallel lines, more formally the new bound is $[h(x), g(x)]$, where $g(x) = \frac{u}{u-l}Eq$ and $h(x) = \frac{u}{u-l}(Eq - l)$. Beyond the theoretical intuition, it is also possible to demonstrate geometrically that the area between the lines is smaller in (b), and this corresponds to a lower overestimation. For more details, we refer to the original papers [Wang et al., 2018a; Weng et al., 2018]. The follow-up work from Zhang et al. [2018], shows that the linear relaxation method can be extended to different activation functions, Fig. 5.5 shows an example of the method on the hyperbolic tangent (tanh).

In recent years, research in the field of formal verification for DNN has grown exponentially. Although the basic structure of the most recent work is based on the presented foundations, more efficient approaches, improvements, and follow-up works have been developed. Among the others, we mention $\alpha/\beta$-*crown* which is widely considered the state of the art among the




(a) Concretization (b) Parallel Relaxation

Figure 5.4: Visual representation of the difference between the *naïve concretization* and the *linear relaxation*.

reachability approaches [Wang et al., 2021].

### OPTIMIZATION APPROACHES

In contrast to the previously presented methods, optimization approaches do not try to compute the reachable set (or output domain). A verification algorithm of this family searches for a concrete input $x_0$, that violates the property, if such input does not exist it means that the property is guaranteed. Recalling the definition of *satisfiability formulation* for an FV problem (previously introduced in this section). A verification query consists of a tuple $\langle N, P, Q \rangle$, Given that $Q$ encodes the negation of the required property, a `SAT` result indicates that the property is violated (and the returned input $x_0$ triggers the violation), while an `UNSAT` result indicates that the property holds.

For example, suppose we wish to verify that the DNN in Fig. 5.6 always outputs a value strictly smaller than 7; i.e., that for any input $x = \langle v_1^1, v_1^2 \rangle$, it holds that $N(x) = v_4^1 < 7$. This is encoded as a verification query by choosing a precondition that does not restrict the input, i.e., $P = (true)$, and by setting $Q = (v_4^1 \geq 7)$, which is the *negation* of our desired property. For this verification query, a sound verifier will return `SAT`, alongside a feasible counterexample such as $x = \langle 0, 2 \rangle$, which produces $v_4^1 = 22 \geq 7$. Hence, the property does not hold for this DNN.

However, these kinds of problems suffer from two limitations: (i) they require a *SAT solver*, which struggle to scale with large DNN; and (ii) it is hard (and in some case even not possible) to work with non-linear activation functions, which is a structural characteristic of the

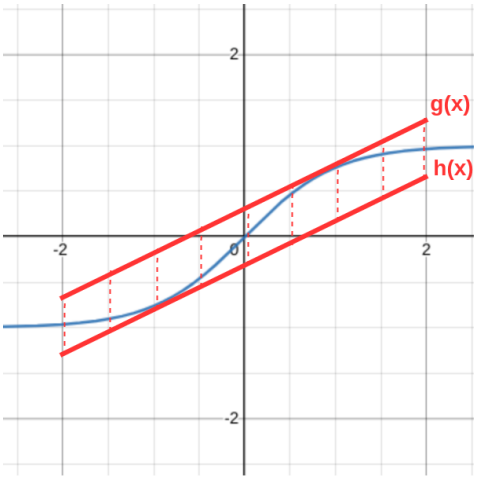



Figure 5.5: Generalization of the *linear relaxation* to different activation functions (tanh in the figure).

DNN. One of the first approaches to deal with this problem is from the work of Bastani et al. [2016]. They propose to reduce the FV task to a *linear programming* (LP) problem, exploiting a *case-splitting* approach to deal with the piece-wise linearity of the non-linear ReLU activation function. Considering $y = ReLU(x)$, the function can be in two linear phases:

1. *Active:* $(x \geq 0) \wedge (y = x)$
2. *Inactive:* $(x < 0) \wedge (y = 0)$

the idea is to perform a case-splitting, enumerating all the possible states of the function. Fig. 5.7 shows an example. If one of the leaves provides a valid assignment, the algorithm returns `SAT`,

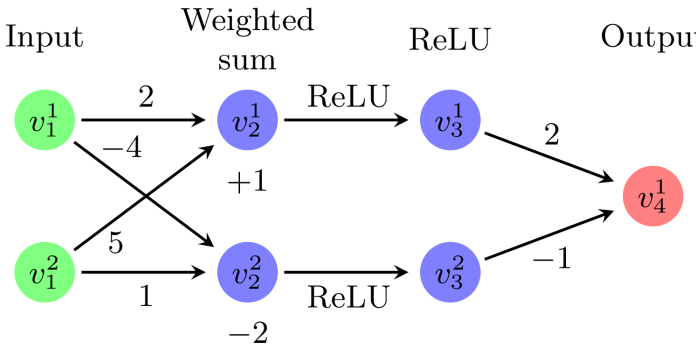



Figure 5.6: A simple neural network with 2 inputs and 1 output.

and the variables assignment is a counterexample that violates the property. Otherwise, to return `UNSAT` it is necessary to explore the whole search tree.

It is clear that this approach suffers from serious scalability limitations on the number of ReLU nodes. However, this approach is sound and complete (which means, in the context of FV for DNN, that the algorithm always returns the correct answer), and has been improved through different algorithmic optimizations and search heuristics [Ehlers, 2017; Lomuscio and Maganti, 2017; Dutta et al., 2017]. An alternative encoding of the problem is based on the *mixed-integer linear programming* (MILP) Tjeng et al. [2018], however, it suffers from the same scalability limitations.

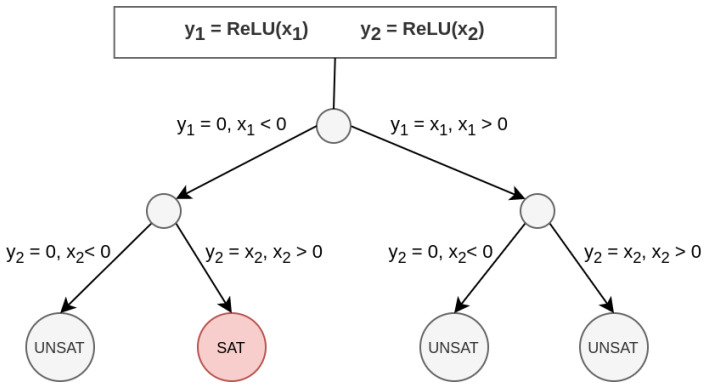



Figure 5.7: High-level overview of the case-splitting approach.

**Reluplex [Katz et al., 2017]** A breakthrough in this field of research has been done by Katz et al. [2017] with *Reluplex*, a sound and complete approach that overcomes the scalability problems of the previous methods. Reluplex is an *SMT-solver*, based on the *simplex* algorithm that exploits a typical concept for the solvers: *laziness*. The idea is to postpone the actions that can cause an explosion in the number of states, which is in contrast with an *eager* approach, such as the case-splitting, that initially enumerates all the possible states. The algorithm follows the structure of the simplex method [Nelder and Mead, 1965], with the addition of three fundamental features:

1. Each ReLU node is represented with two variables, the pre-activation variable $x^w$ and the post-activation variable $x^a$.

2. $x^w$ and $x^a$ are handled as independent variables for the simplex algorithm; this means that they can temporarily violate the ReLU constraint (i.e., $x^a{=}max(x^w,\ 0)$).

3. $x^w$ and $x^a$ are fixed incrementally, and only if necessary, to find a valid assignment (i.e., `SAT`).

The algorithm is sound and complete, however, in some corner cases, the naïve process may not terminate. This happens when the assignment repeatedly violates the ReLU constraints, getting stuck in a loop. In this case, Reluplex performs a case-splitting, expanding the research tree. In recent years, the algorithm has been improved with different optimizations (e.g., bound tightening and back-jumping), the improved version of the algorithm is *Marabou*, presented in the work of Katz et al. [2019], and is widely considered the state of the art in the optimization-based approaches.

# Chapter 6

# VERIFICATION OF DECISIONS (PROVE)

In the previous chapters, we introduced the concept of formal verification for neural networks (FV). However, despite the abundance of both *Deep Reinforcement Learning* (DRL) systems and FV techniques, little work has been published on demonstrating the applicability and usefulness of verification techniques to real DRL controllers, especially when applied to robotic tasks. In this chapter, and from a different perspective also in Chap. 7, we try to bridge the gap. In this chapter, in particular, we propose a novel verification tool, *ProVe*, designed for the formal verification of DRL policies, which typically focus on decision-making problems. We also introduce a novel metric, the *violation rate*, to formally evaluate the performance of a DRL policy from a safety-oriented perspective. Moreover, ProVe can solve the counting version of the verification problem, computing the portion of the input domain that cause a violation of the requirements. This can overcome the limitations of the binary `SAT` or `UNSAT` answer and can be exploited in different ways that we will show in the next sections.

## 6.1 MOTIVATION

Following the recent trend in formal verification for Neural Networks, we propose to design a set of safety properties, that encode different constraints on the behavior of the agent, as a complementary metric for the reward. Ideally, given a safety property and a neural network, a verification framework should either guarantee that the property is always satisfied or return counterexamples [Liu et al., 2019]. The effectiveness of such methodology relies on the estimation accuracy of the output bounds [Wang et al., 2018b] and has been successfully addressed by several recent studies [Wang et al., 2018a, 2021]. One of the main limitations of these approaches concerns the design of the safety properties. State-of-the-art methods have proven to be fast and efficient only if the input domain is strict and the target behavior is well known [Katz et al., 2019]. However, formalizing a set of properties that satisfy these requirements may not be

possible without a deep prior knowledge of the environment, and, forcing the agent to respect such strict requirements, may negatively affect the final policy generated by the DRL algorithm. In contrast, in this paper, we focus on the formal verification of properties that describe the general behavior of the agent (i.e., *behavioral* properties), and that aim at ensuring that the policy makes rational decisions (e.g., if there is an obstacle close to the right never turn right). Crucially, we show that a model that respects such soft constraints is overall safer than a model evaluated only on the long-term reward. Notice that, properties of this form typically require large input domains, as they need to cover a wide variety of possible input configurations for the DNN (i.e., possible situations for the agent). As a consequence, they are rarely respected in the whole input domain and state-of-the-art approaches typically return only `SAT` (i.e., the property is respected) or `UNSAT` (i.e, the property is violated with at least one input configuration), failing to provide useful information on the safety of the model. Against this background, we introduce ProVe a formal verification tool for DNN based on interval analysis [Wang et al., 2018a], designed to verify safety properties for decision-making tasks defined over large input spaces. While previous approaches provide verification tools that aim at verifying whether the bound of an output of the network lies in a given interval, in a DRL context, DNNs typically encode decision-making policies and require the analysis of multiple outputs, considering the relationships among them. To verify this kind of property we need to modify the standard analysis of the output interval, in fact, as detailed in Chap. 5, even in an ideal scenario with a perfect estimation of the output intervals, state-of-the-art tools can not always provide useful information on the relationships among the outputs (i.e., can not exploit the comparison rules of the intervals analysis [Moore, 1963]). To this end, in this paper, we propose a novel approach, that provides an accurate shape estimation of the output functions, by computing an iterative bisection of the input intervals. In detail, for a property that should be evaluated in the global domain $I$, we analyze independently a set of $n$ smaller domain $i_n$ (s.t., $I = \bigcup_{i=1}^{n} i_i$). This paves the possibility to handle properties encoded in the form described above. Moreover, we ca exploit the computation independence of the intervals to encode the process in a parallel fashion (e.g., multi-core CPU or GPU), improving the performances of the standard verification tools and hence handling very large input spaces. Crucially, with our approach, it is possible to compute the size of the domain that violates the safety property, providing a safety metric for the evaluation of a model (*violation rate*). Finally, we empirically evaluate ProVe on different domains including (i) the airborne collision avoidance system (ACAS) [Owen et al., 2019], used in literature as a standard benchmark, and (ii) trajectory generation for the commercial Panda manipulator.

## 6.2 BEHAVIORAL PROPERTIES

Verification of DNN for decision-making requires the comparison between the outputs and this may be difficult to achieve with previous approaches. In particular, Fig. 6.1 visualizes different

scenarios of our output analysis as a 2d graph, to simplify its understanding. Notice that a network with $n > 1$ requires a multi-dimensional graph, however, we assume that each point on the x-axis represents a tuple of $n$ inputs ($x \in X$) in an arbitrary order (as it does not affect the analysis), we represent the outputs ($f(X)$) on the y-axis. Fig. 6.1a shows the typical result of previous verifiers: a generic representation of a single output function. This is also an ideal scenario where the overestimation problem is solved (i.e., the output bounds [a, b] matches the minimum and maximum of the output function). However, Fig. 6.1b clearly shows the limitations of such methods in the verification of decision-making tasks, where $y_0$ and $y_1$ represent the output functions generated by two nodes of a generic network. In particular, it is not possible to infer which output action will be selected as they only compute the output bounds, without considering the shape (and the relationship) of the output functions. The main insight of ProVe is to compute an accurate estimation of the output function shape, subdividing the input area into multiple subareas and computing the previous operations for each subarea. Fig. 6.1c shows that, through this process, it is possible to obtain a better estimation of an output shape. Finally, Fig. 6.1d considers multiple outputs and subareas, allowing ProVe to state that output $y_0$ is the output that the network will choose for the given input area (i.e., $y_0$ always returns a higher value than the other outputs). This analysis of the relationships among the network outputs is necessary to formally verify a decision-making property.

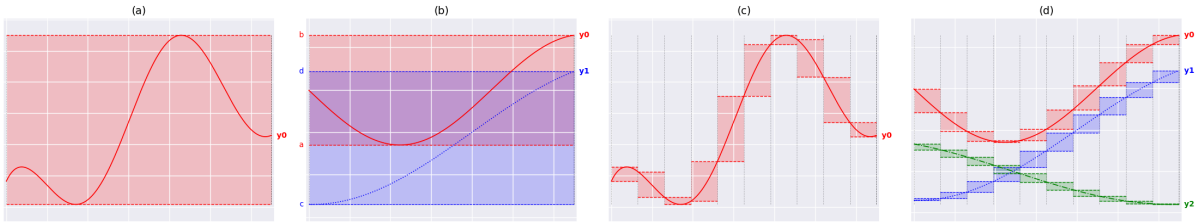



Figure 6.1: Explanatory output analysis of: (a) one output function with one subdivision; (b) decision-making problem with two outputs and one subdivision. (c) Estimation of an output function shape, using multiple subdivisions. (d) Output analysis with three outputs and multiple subdivisions.

### DEFINITION OF SAFETY PROPERTIES FOR DECISION-MAKING

Following the formulation provided by Liu et al. [2019] (and adopted as the standard in the previous state-of-the-art works), a safety property for a neural network formalizes an input-output relationship. In detail, a safety property can be formalized in the following form:

$$\Theta : \text{If } x_0 \in [a_0, b_0] \land ... \land x_n \in [a_n, b_n] \Rightarrow y_j \in [c, d] \tag{6.1}$$

where $x_k \in X$, with $k \in [0, n]$ and $y_j$ is a generic output.

A property in this form aims at verifying if an output of a network lies in a specific interval. This formulation can be applied to many problems related to robotics and deep learning in

general (e.g. the velocity limit of a motor or the probability in a classification task). However, even if it is possible to adapt this formulation to verify simple decision-making properties, it requires manually modifying the input network and introduces overhead in the verification process.

In contrast, we propose a different formulation, specifically designed for decision-making problems:

$$\Theta : \text{If } x_0 \in [a_0, b_0] \land ... \land x_n \in [a_n, b_n] \Rightarrow y_j > y_i \tag{6.2}$$

We refer to these properties as *safe-decision properties* as they can be used to ensure that a given action (e.g., $y_j$) is always preferred over the others for a given input configuration.

Following the insights of the previous section, we exploit this proposition to prove (or deny) a variety of properties in the form of Prop. 6.2. In detail, ProVe compares the computed bounds, verifying if the values of one of them are strictly lower than the others, which means that the DNN never selects the action related to the output with the lower value. As an example, to verify properties for a simplified navigation scenario encoded by a DNN with: (i) inputs $x_i \in [0, 1]$ with $i = 0, .., 3$, representing the normalized distance from an obstacle in the four cardinal directions (1 translates in a distance $\geq 1m$ in that direction), where $x_0$ is the *right* distance and $x_1$ is the *left* distance. (ii) outputs $y_0 = right$, $y_1 = left$, representing the directions where the agent can turn. We could be interested in a property as $\Theta$*: If an obstacle is close to the right and other directions are obstacle-free, always turn left.* Assume we measured the minimum distance from an obstacle ($0.07m$) that allows our autonomous drone to avoid a collision when turning in the opposite direction, supposing the worst case, where the robot is moving at its maximum speed. We can exploit this constraint to formalize the safety property in the form:

$$\Theta : \text{If } x_0 \in [0, 0.07] \land x_1, x_2, x_3 \in \mathcal{D} \Rightarrow y_0 < y_1 \tag{6.3}$$

where $\mathcal{D} = (0.07, 1]$. To verify the relation between two (or more) outputs, ProVe relies on the interval algebra of Moore [1963]. In particular, supposing $y' = [a, b]$ and $y'' = [c, d]$ we have the preposition:

$$b < c \Rightarrow y' < y'' \tag{6.4}$$

To better explain the key problem of previous interval analysis-based approaches, that prevent a direct application to decision-making, in Fig. 6.1 we show a simplified visual example of an output analysis for a decision-making property. In a typical scenario, we often have that $max(y_1) > min(y_0)$, hence we can not assert anything on the property. Fig. 6.1b shows an example of this behavior, where $d \not< a$, (i.e. the bounds overlap). In this scenario verification frameworks can not formally verify the property (i.e., we do not have enough information to state if the property condition is true or false). ProVe directly addresses this problem by computing the propagation for a subset of the input area to obtain a more accurate estimation of the output function shape (Fig. 6.1c). This leads to Fig. 6.1d, where $y_1(x) < y_0(x)$ for any $x \in X$ ($X$ is the set of the possible inputs), which translates, de facto, in $y_1 < y_0$ (i.e. the network

always choose the action represented by $y_0$ inside the input area specified by the property). Furthermore, $y_2 \not\leq y_1$ (the agent can choose $y_2$ in that input domain).

## 6.3 PROPERTY VERIFIER: PROVE

In this section, we address the huge amount of memory and time required to compute and verify the output bounds for decision-making properties. Furthermore, we introduce a violation rate metric to measure the reliability of a model with respect to a property, showing that in some scenarios, we can design a simple controller to ensure the correct behavior of a DNN in the entire input area.

```
# Function that implements the actual algorithm, we assume to have the
# following functions for the support operations:
#    -> generate_matrix
#    -> split_area
#    -> update_violation
#    -> get_output_bound
def ProVe( DNN, areas, _property, eps ):
    mul_matrix  = generate_matrix( DNN )
    areas = split_area( areas, eps, mul_matrix )
    for sub_area in areas:
        output_bounds = get_output_bound( DNN, areas )
    violated_areas = []
    proved_areas = []
    violation_rate = 0
    for bound in output_bounds:
        test = check_property( bound, _property)
        if test is VIOLATED:
            violated_areas.append( bound )
            violation_rate = update_violation()
        if test is PROVED:
            proved_areas.append( bound )
    if len(areas) != 0:
        return ProVe( DNN, areas, _property, eps )
    return violated_areas, violation_rate
```

Listing 6.1: A python-like implementation of ProVe (Property Verifier) in the sequential version.

Considering the pseudo-code in Listing 6.1, ProVe takes as input: (i) a trained DNN; (ii) the input area encoded as a matrix, (iii) the property to verify, and (iv) a discretization value $\epsilon$. ProVe proceeds by performing an iterative recursive splitting process using a matrix encoding. In more detail, ProVe generates the multiplication matrix for the iterative splitting of the input area, performing the first input split. Then, ProVe propagates the input to compute the output bounds for each unverified sub-areas (a subdivision of the former area, as described in Chap. 5). This is the most computationally demanding part of ProVe (the number of sub-areas is expo-

nential in the recursion depth). However, this procedure is easily parallelizable on GPU, given that the propagation of each sub-area is independent and the computed output bounds can be analyzed individually, sequentially loading on the GPU memory a subset of the former sub-areas. After these preliminary operations, ProVe evaluates the safety property for each output-bound, returning three possible outputs: (i) the property is violated; (ii) the property holds, or (iii) we can not conclude anything on the property in this area (i.e. the bounds overlap as detailed in Chap. 5). In the first two cases, the property is *verified* (proved or denied) and we remove the verified area from the matrix. In the third case, the area matrix is not empty, and we recursively call the algorithm with the remaining unverified sub-area as input. Moreover, during the main loop, the violation rate is constantly kept updated. In general, a property could require an uncountable number of splits to be verified. For this reason, we introduce a discretization value $\epsilon$ to create an upper bound on the possible number of iterations. If the area matrix is empty, we return all the violated areas as counterexamples along with the *violation rate* that represents an overestimation of the probability for a property to fail in a real execution. If the violation rate equals 0, the property is formally verified in the given input area (i.e. the property is true). Finally, Fig 6.2 reports a high-level overview of the main loop.

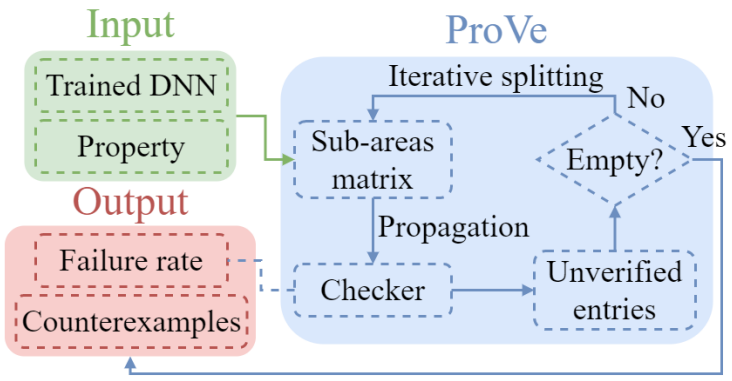



Figure 6.2: High-level overview of ProVe.

**Matrix Encoding** The verification of decision-making DNNs may require a large number of area splitting, hence to use this technique in practical applications the iterative splitting process must be very efficient. The core idea behind ProVe is to exploit matrix operations for the iterative splitting process. Matrix multiplication is a widely used operation for several optimizations and inference tasks, it is known to be highly parallelizable and there exist several dedicated and efficient implementations. [1]. We encode the area splitting by using a matrix $A^0$ of size $m \times 2n$, where $m$ is the number of entries (i.e. the sub-areas to split) and $n$ the number

[1]For Prove we rely on *NumPy*, a *Python* package for scientific computing, https://numpy.org/

of input nodes (i.e. the bounds for each node, which are a couple of values). To split the *k-th* input node, we first generate a *multiplication-matrix* $B_{2n\times 4n}$. These are structured as:

$$A^0_{m\times 2n} = \begin{bmatrix} [a_{0,0}, b_{0,0}] & ... & [a_{0,n}, b_{0,n}] \\ ... & ... & ... \\ [a_{m,0}, b_{m,0}] & ... & [a_{m,n}, b_{m,n}] \end{bmatrix}$$

$$B_{2n\times 4n} = \begin{bmatrix} \widetilde{I_{2n}}' & \widetilde{I_{2n}}'' \end{bmatrix}$$

In detail, the matrix $B$ is formed by two identity matrices. Both these matrices maintain the values of an identity matrix $2n \times 2n$, except for the following elements:

$$\widetilde{I_{2n}}'[2k][2k+1] = 0.5 \qquad \widetilde{I_{2n}}'[2k+1][2k+1] = 0.5 - \epsilon,$$
$$\widetilde{I_{2n}}''[2k][2k] = 0.5 \qquad \widetilde{I_{2n}}''[2k+1][2k] = 0.5 + \epsilon$$

where $\epsilon$ is required to solve the infinite splitting loop that we analyze in the next section. The result of matrix multiplication $A^1 = A^0 \times B$ is a matrix $m \times 4n$, where a row contains two parts of the corresponding input area, split on the desired *k-th* node. Finally, in time $\mathcal{O}(1)$ we reshape the matrix $A^1$ to the desired matrix $A^1_{2m\times 2n}$[2]:

$$A^1_{2m\times 2n} = \begin{bmatrix} [a_{0,0}, b_{0,0}] & ... & [a_{0,n}, b_{0,n}] \\ ... & ... & ... \\ [a_{2m,0}, b_{2m,0}] & ... & [a_{2m,n}, b_{2m,n}] \end{bmatrix}$$

**Input Discretization** The continuous domain of the input space makes it always possible to split an input area, hence ProVe could, in the same specific worst-cases, loop infinitely on the iterative refinement process. To address this, we introduce a discretization value $\epsilon$ to limit the input precision and ensure the convergence of our algorithm in a finite number of steps. Crucially, the $\epsilon$ parameter could be as small as required by the application, however, this will have an impact on the worst-case memory and computation required by the approach. In particular, both the time and space complexity of the algorithm are exponential in $\epsilon$, as we show in the previous analysis. Notice that, $\epsilon$ can be considered as a lever to address the trade-off between analyzing input with higher precision and maintaining the computation manageable. For several applications, and specifically for robotics, $\epsilon$ can be tuned considering the precision of the sensing system. In our mapless navigation task, we set $\epsilon$ to be the precision of the lidar system. With this setting ProVe was able to verify the application within an acceptable amount of time on a standard computational architecture (refer to the evaluation section for more details).

[2]Note that these operations can be performed efficiently by exploiting optimized routines implemented in standard packages for linear algebra such as NumPy.

**Splitting Heuristics** In addition to our matrices encoding and the $\epsilon$ discretization value, we design a heuristic to select which bound to split at each iteration and maximize the entries removal from the matrix at each recursion cycle (i.e., reduce memory and time required for the computation). We propose a comparison between the worst case (i.e. all areas reaches precision $\epsilon$, which is the maximum recursion depth) and three heuristics: (i) *random:* randomly selects one of the available nodes (i.e. nodes with bound size $\geq 0$), (ii) *biggest-first* selects the nodes with the largest bound size and (iii) *best-first* selects the nodes that most influence the output. An estimation of the influence is computed on the first layer of the network, using the absolute weight value of each input node.

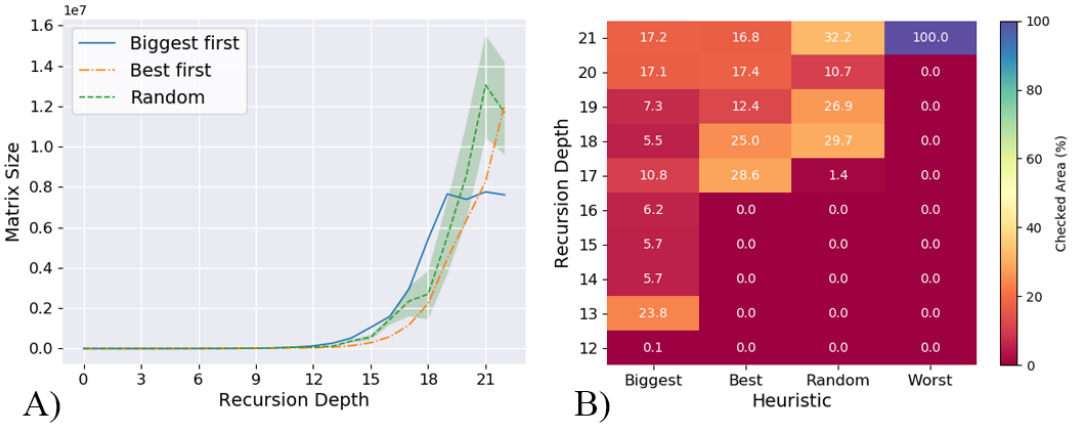



Figure 6.3: Splitting heuristics evaluation overview. A) Growth of the matrix size within the depth of the recursion. B) Distribution of checked areas by our heuristics within the depth of the recursion.

To evaluate the best heuristic, we performed an initial evaluation in a simplified version of the CartPole benchmark Sutton and Barto [2018]. In this task, an agent has to move a cart left and right (which are the two outputs) so that a pole can stand (within a certain angle) as long as possible. In our simplified setting, we only consider the angle and the velocity of the pole as input to provide better graphical visualization of the results. In detail, we performed our evaluation on the property:

$\Theta_{eval}$*: If the pole is falling on the right, the cart must be moved in the same direction to stand the pole.*

Figure 6.3A shows the matrix size (i.e., the memory) required to verify this property (y-axis) with respect to the recursion depth (x-axis). Notice that only the random heuristic presents variance over five statistically independent runs as the other two are deterministic. Clearly, in this explanatory example, the biggest-first is the best heuristic. Figure 6.3B confirms this result, showing the distribution of the checked area for each recursion depth. In detail, biggest-first

finds a sensible amount of areas to remove at iteration 13, while best-first and random strategies find a similar situation $\approx 6$ iterations later. In summary, the biggest-first heuristic shows a memory improvement of over 90% after 21 iterations with respect to the worst case, where all the areas are found at the maximum depth.

ALGORITHM ANALYSIS

In this section, we first provide a convergence proof, followed by the complexity in terms of time and space.

**Property 1.** *Given an input area and a discretization value $\epsilon$, ProVe always converges in a finite number of steps without loss of precision.*

*Proof.* To prove the convergence in a finite number of steps we demonstrate the following properties: (i) there are no infinite loops in the iterative splitting; (ii) splitting an area into two sub-areas covers all the discrete values of the range up to precision $\epsilon$. For the latter, we use $\epsilon$ in the computation of the mean value because the mean of an area could have higher precision with respect to $\epsilon$, producing an infinite loop. This is due to the fact that the discrete values at a higher precision than $\epsilon$ are odd (it is ambiguous to divide it into equally sized groups).

Suppose an $\epsilon = 0.1$ and a split operation of the range [0.0,0.1]; we first compute the mean 0.05, splitting in [0.0,0.05] and [0.05,0.1]. We then round these values to the specified precision, obtaining [0.0,0.1] and [0.1,0.1]. The introduction of $\epsilon$ allows for rounding alternatively up and down in order to avoid infinite recursion. For the former, it is trivially true that we cover all the discrete values of the range up to precision $\epsilon$, because if the mean between the extreme values of the range is at precision $\epsilon$, then the union of the sub-areas returns the original one. Moreover, if the mean has higher precision than $\epsilon$, we round the mean value once to the next bigger value at precision $\epsilon$ and once to the smaller ones. In particular, there is no value at precision $\epsilon$ that is between these two. □

**Time Complexity** Given the number of ranges in input $n$ and the discretization value $\epsilon$, we define the range size as $l$ (fixed to 1 by normalization) and the time cost of the control operation (propagation and property verification) as *op*. The time required for matrix multiplication is $\mathcal{O}(mnp)$ where the input matrices are $m \times n$ and $n \times p$. The structures of the matrices used by ProVe are always in form of $m \times n$ and $n \times 2n$, where $m$ is dependent on the iterations of our algorithm; so the time required for multiplication is $\mathcal{O}(n^2 m)$. For construction, we know that the number of rows ($m$) doubles up at each iteration, i.e., $m = 2^i$ where $i$ is the number of iterations already performed. The time complexity of the algorithm becomes $\mathcal{O}(\sum_{k=1}^{I} n^2 2^{(k-1)})$ that can be written as $\mathcal{O}(n^2 \sum_{k=1}^{I} 2^k 2^{-1})$. Applying Gauss formula $\mathcal{O}(\frac{n^2}{2} 2^{\frac{I(I+1)}{2}})$ where $I$ is the total number of iterations of the algorithm. The total number of iterations required from the algorithm depends on the range size $l$ and the precision $\epsilon$; in fact, each iteration divides an area into two equally sized sub-areas. For a single input range that starts from 0 to 1 we need

$\mathcal{O}(\log_2\lceil\frac{l}{\epsilon}+1\rceil)$ and the total number of operation is $I = \mathcal{O}(n\log_2\lceil\frac{l}{\epsilon}+1\rceil)$. Finally, the time complexity of the algorithm is $\mathcal{O}(op\frac{n^2}{2}2^{\frac{(n\log_2\lceil\frac{1}{\epsilon}+1\rceil)^2}{2}})$. In summary, the time complexity of ProVe is exponential both when $\epsilon$ decreases and $n$ increases.

**Space Complexity** Recalling the notation of the previous section and the fact that at each iteration we double up the number of rows of our matrix, ProVe requires $\mathcal{O}(2^{I^n})$ rows and always $n$ columns for the matrix. The total space required is then $\mathcal{O}(n2^{I^n})$ that is $\mathcal{O}(n\lceil\frac{1}{\epsilon}\rceil^n)$. This analysis further motivates the introduction of the $\epsilon$ discretization value to limit the depth of the recursion and the search for a heuristic, to limit the complexity of verification approaches.

**Completeness of the Method** ProVe is a sound method but, in its naive implementation, it is not complete. However, ProVe can be considered complete under one of these two assumptions: (i) if the discretization parameter $\epsilon$ allows, in the worst case, to completely explore the generated tree and verify independently all the leaves, the method can verify the whole input-domain, returning a discretized but complete answer; or (ii) exploiting techniques from the optimization-based algorithms for verification (refer to Chap. 5 for details), and assuming to have only ReLU activations functions, ProVe can exploit the separation of the ReLU activations in the two linear phases.

## 6.4 THE VIOLATION RATE

As we discussed in Chap. 5, while standard optimization methods can only return `SAT` or `UNSAT`, a key component of our approach is to quantify the number of violations compared to the complete reachable set. We introduce a violation rate metric to infer how a trained DNN performs with respect to the given properties. We define this metric as the percentage of the input area that causes a violation, to compute this value at each step we normalize the size of the area that violates the property with respect to the size of the original input area. Crucially, the violation rate is an upper bound for the actual probability of failure as visually shown in Fig. 6.4. In detail, the left Fig. shows the distribution of the state over 10000 episodes of the CartPole scenario [Sutton and Barto, 2018], while the right one shows the input configurations that cause a violation of a safety property in that environment. Clearly, the states where failures occur are rarely encountered input. This confirms that an empirical evaluation would most probably not encounter those states, ignoring errors that may appear during the deployment in a real-world context.

**Definition 6.4.1** (Violation Rate)**.** *Given a set of behavioral properties* $\Pi$ *with input domain* $\mathcal{X}$*, a neural network function* $f_\theta(x)$ *and the corresponding estimated reachability set* $\Gamma(\mathcal{X}, f_\theta) := \{\mathbf{y} : \mathbf{y} = f_\theta(\mathbf{x}),\ \forall \mathbf{x} \in \mathcal{X}\}$*. Defining* $\mathcal{X}_{UNSAT}$ *as a subset of the original domain* $\mathcal{X}$ *such that* $\Gamma(\mathcal{X}_{UNSAT}, f_\theta) \cap \mathcal{Y} = \emptyset$*, we define the violation rate as follow:* $v = \frac{|\mathcal{X}_{UNSAT}|}{|\mathcal{X}|}$*,*

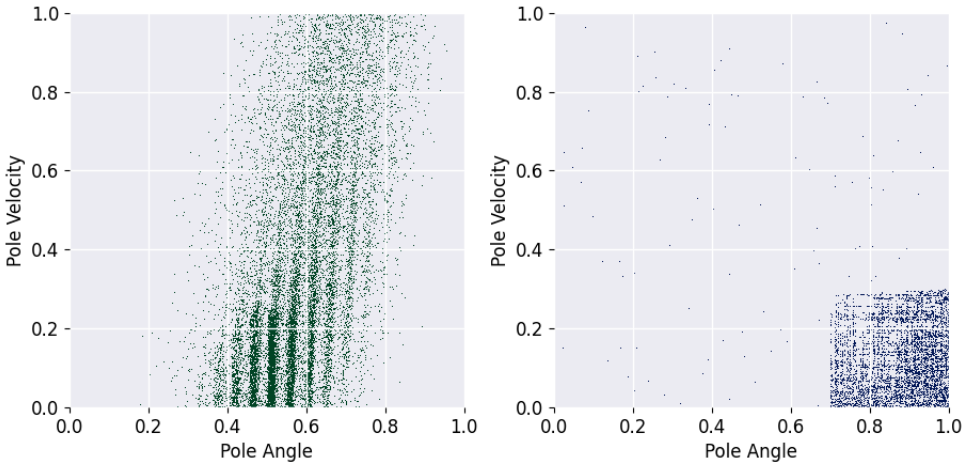



Figure 6.4: Overview of (left) the most frequent input configuration encountered in a typical execution of the *CartPole* environment and (right) the configurations found by ProVe, that cause a property violation.

**Definition 6.4.2** (Safe Rate)**.** *Given a set of behavioral properties* $\Pi$ *with input domain* $\mathcal{X}$*, a neural network function* $f_\theta(x)$ *and the corresponding estimated reachability set* $\Gamma(\mathcal{X}, f_\theta) := \{\mathbf{y} : \mathbf{y} = f_\theta(\mathbf{x}),\ \forall \mathbf{x} \in \mathcal{X}\}$*. Defining* $\mathcal{X}_{SAT}$ *as a subset of the original domain* $\mathcal{X}$*, such that* $\Gamma(\mathcal{X}_{SAT}, f_\theta) \subseteq \mathcal{Y}$*, we define the safe rate as follow:* $s = \frac{|\mathcal{X}_{SAT}|}{|\mathcal{X}|}$.

Finally, in the evaluation section, we show how to exploit the violation rate to design a *simple controller* to check at run-time if the current network input causes a violation, formally guaranteeing the safety related to the desired properties.

## 6.5 RESULTS

We empirically evaluate ProVe on two domains: ACAS, trajectory generation for a robotic manipulator, and mapless navigation. For these domains, we train the DNN by using the Rainbow algorithm [Hessel et al., 2018]. Data are collected on a commercial desktop computer (equipped with a GPU NVIDIA2070, an 8-core CPU, and 16gb RAM), with a *c++* code based on CUDA11.

**ACAS XU Comparison** The airborne collision avoidance system (ACAS), designed to prevent collision between aircraft, has been widely adopted by previous verification approaches [Katz et al., 2017; Wang et al., 2018b]. This system consists of a set of networks, each one with five inputs: (i) distance between ownship and intruders, (ii) heading of ownship with respect to an intruder, (iii) heading of the intruder relative to ownship, (iv) speed of ownship and (v) speed of the intruder; and five outputs to encode the action to take to avoid the collision: (i) clear of

conflict (which means that no action is needed), (ii) weak right, (iii) strong right, (iv) weak left and (v) strong left. For our evaluation, we rely on the *HorizontalCAS* implementation of Owen et al. [2019], which provides a set of trained networks and an extensive dataset of safety-critical situations.

In the previous sections, we introduced the problem of the formalization of the safety properties. Designing a set of properties that cover all the possible violations could be hard and requires a deep prior knowledge of the environment. In some cases (e.g., real-world applications or robotics) this process is unfeasible. To support our claims, in Table 6.1 we show that, even the 15 properties of ACAS, used as standard metric to evaluate the safety in the state-of-the-art works [Wang et al., 2018a; Katz et al., 2017], are not informative enough to guarantee collision avoidance. In contrast, we propose to design a small set of *behavioral* safety properties, that aims to ensure that the agent makes rational decisions, instead to verify the complete set of unsafe possible actions. We exploited ProVe to obtain the *violation rate* on only two fundamental properties:

$\theta_L$**:** If there is an intruder close on the left, never turn left.
$\theta_R$**:** If there is an intruder close on the right, never turn right.

Table 6.1 shows our results. We compare six models that achieve the maximum cumulative reward for the task, but with significantly different *violation rates*. To obtain an estimation of the real collision we rely on the safety-critical configurations provided by Owen et al. [2019], comparing the behavior of our models with the correct actions of the dataset. More in detail, from the multitude of trained models that achieve similar rewards, we select a subset with a violation rate ranging from 50 to 4.8 (our best result) with a step of 10. Notice that, in contrast to the standard formulation, for our *behavioral* properties it is difficult to obtain a *violation rate* of zero, having to cover a huge amount of possible situations. Moreover, we believe that this is not an objective. In some limit cases, take dangerous decisions could provide benefits in the long term.

Table 6.1: Comparison between the real collision probability and the violation rate of our *behavioral* properties. The table also shows the number of collisions for a model that respects all the original 15 ACAS properties.

| **Model** | **Properties** | **V. Rate (%)** | **Collision (%)** |
|---|---|---|---|
| *acas_50* | $\theta_L$, $\theta_R$ | $\approx 50$ | 14.21 |
| *acas_40* | $\theta_L$, $\theta_R$ | $\approx 40$ | 9.34 |
| *acas_30* | $\theta_L$, $\theta_R$ | $\approx 30$ | 8.22 |
| *acas_20* | $\theta_L$, $\theta_R$ | $\approx 20$ | 5.04 |
| *acas_10* | $\theta_L$, $\theta_R$ | $\approx 10$ | 3.71 |
| *acas_05* | $\theta_L$, $\theta_R$ | $\approx 5$ | 0.72 |
| *acas_og* | $\theta_0$, ..., $\theta_{15}$ | $\emptyset$ | 0.56 |

Our results show a strong correlation between the *violation rate* and the real collision estimation, highlighting the limits of the reward for the evaluation of safety-critical tasks and the relevance of our novel metric. Finally, Table 6.1 shows that the models that obtain the lowest violation rate on our properties (*acas_05*) result in a similar collision number to the one that respects all the 15 original properties (*acas_og*). This further motivates our claims and the hypothesis that our *behavioral* properties, designed without a deep prior knowledge of the environment, provide a good indicator of the safety of a model.

**Mapless Navigation** In mapless navigation a robot must reach a given target without a map of the surrounding environment, using only local observation to avoid obstacles. This is a challenging scenario for DRL that has attracted significant attention in recent literature [Zhang et al., 2017; Wahid et al., 2019]. In detail, we consider a Turtlebot3 navigating with constant linear velocity, which is a widely used platform in several previous works focusing on DRL for navigation [Tai et al., 2017, 2016]. Here, we use Unity as a simulator [Juliani et al., 2018] (previous work [Marchesini et al., 2019] demonstrates that this is an efficient and realistic simulation of the behavior of the robot).

In our setup, the network has twelve inputs: (i) the laser sensor is used to collect sparse 11-dimensional scan values $x_0, .., x_{10}$ normalized $\in [0, 1]$ and sampled between -90 and 90 *deg* in a fixed angle distribution. The lidar sensor precision is the manufacturer specification used to set the $\epsilon$ discretization value; (ii) the heading of the target with respect to the robot heading ($x_{11}$ normalized $\in [0, 1]$); and three outputs for the angular velocity (i.e., [-90, 0, 90] deg/s).

We evaluate the robustness of the network with three different properties, that represent important behaviors that the agent must respect to be considered reliable in navigating a polygonal map without collision with the surrounding walls.

$\Theta_{T,0}$**:** If the target is in front of the robot and no obstacle is detected, go straight.
$\Theta_{T,1}$**:** If there is an obstacle close to the left never turn left.
$\Theta_{T,2}$**:** If there is an obstacle close to the right never turn right.

**Trajectory Generation for a Robotic Manipulator** We chose safety verification for robotic manipulators because they are critical assets for industrial environments. Our task is to rotate each joint of the robot to generate a real-time trajectory to reach a target (a similar task is considered in the work of Gu et al. [2017] and in the work of Marchesini et al. [2019]). In our setup, the input layer of the network contains 9 nodes normalized in the range $[0, 1]$: (i) one for each considered joint and (ii) the last three to encode the target coordinate. We use 12 nodes in the output layer: each joint is represented by 2 nodes to decide if it should move $\omega$ degrees clockwise or anti-clockwise. This encoding of the output allows a straightforward verification process for our tool (i.e., one node represents only one specific action). Furthermore, operating the manipulator in the Cartesian space, we can use $\omega$ as the $\epsilon$ value. Hence, to formally verify if the manipulator operates inside its work-space, we consider properties in the following form:

if the current angle of a joint $j_i$ is equal to one of its domain limits, whatever the configuration of the other joints and whatever is the position of the target, the robot must not rotate $j_i$ in the wrong direction (i.e. an action that rotates $j_i$ causes the robot to exit from the work-space). Considering the network architecture and our task formalization, we design safety properties as:

Table 6.2: Execution time comparison between Neurify and ProVe on the standard properties of the ACAS Xu dataset. For each group of properties, the table shows the mean computation time (seconds) and the average speed up (x). Given the deterministic nature of the considered approaches, the variations between different runs on the same hardware are negligible, therefore we reported only the average values.

| **Property** | **Neurify (sec)** | **ProVe (sec)** | **Gain (X)** |
|---|---|---|---|
| $\Theta_{A,1}$ | 1037.37 | 345.12 | 3.01 |
| $\Theta_{A,2}$ | 19352.12 | 339.72 | 56.08 |
| $\Theta_{A,3}$ | 1359.89 | 159.22 | 8.54 |
| $\Theta_{A,4}$ | 113.62 | 132.31 | 0.86 |
| $\Theta_{A,5}$ | 22.07 | 5.16 | 4.27 |
| $\Theta_{A,6}$ | 4.91 | 11.5 | 0.42 |
| $\Theta_{A,7}$ | 1278.7 | 64.84 | 19.72 |
| $\Theta_{A,8}$ | 412.29 | 11.21 | 36.78 |
| $\Theta_{A,9}$ | 643.8 | 42.97 | 14.98 |
| $\Theta_{A,10}$ | 60.01 | 10.08 | 5.95 |
| $\Theta_{A,11}$ | 0.51 | 5.79 | 0.08 |
| $\Theta_{A,12}$ | 8.82 | 8.86 | 0.99 |
| $\Theta_{A,13}$ | 34.76 | 10.34 | 3.36 |
| $\Theta_{A,14}$ | 23.74 | 8.09 | 2.93 |
| $\Theta_{A,15}$ | 1136.55 | 8.22 | 16.61 |
| **Total:** | **25526.52** | **1163.43** | **21.94** |

**$\Theta_{P,0}$: If the first joint current rotation is close to the left limit of the workspace, never rotate that joint on the left.**

Property $\Theta_{P,0L}$ represents a configuration where the angle of joint $j_0$ equals its limit on the left (i.e., a normalized value 1) and whatever values the other inputs of the network assume, the output value corresponding to the action *rotate left*, must be lower than at least one of the others. For each joint $j_i$ we consider two properties, one for the left limit ($\Theta_{P,iL}$) and one for the right limit ($\Theta_{P,iR}$).

**Discussion** In order to collect statistically significant data, we performed different training phases for each task, using different random seeds [Cédric et al., 2019]. We report the mean and standard deviation (smoothed over one hundred episodes) for each task, considering (i)

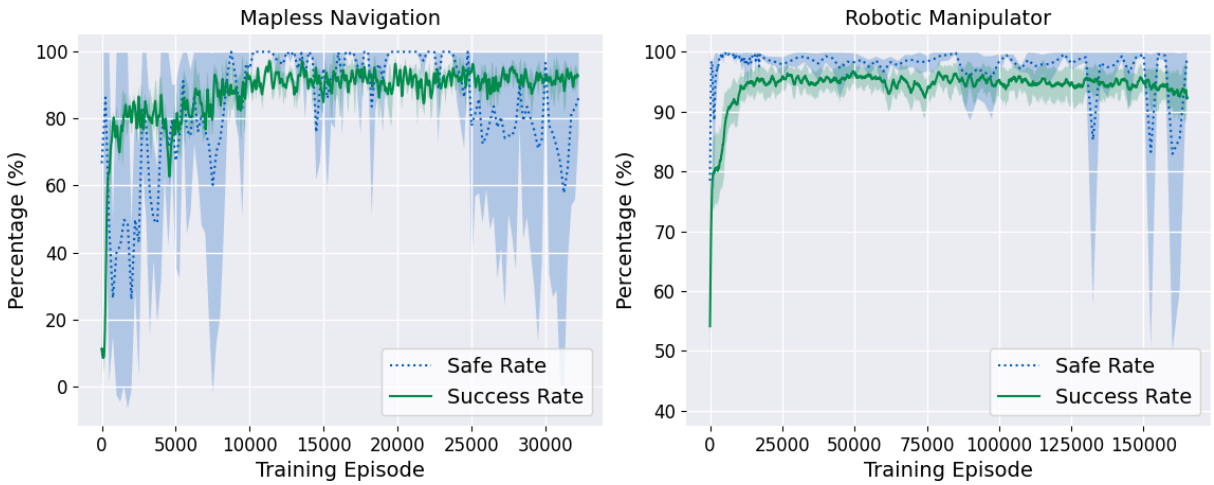



Figure 6.5: Comparison between the success rate and the safe rate in our environments.

the success rate (i.e., the number of successful episodes over 100 sequential steps) and (ii) the safe rate (the complementary metric of the violation rate, used for a clearer visualization of the graphs).

Fig. 6.5 shows that the *violation rate* is not directly correlated to the success rate (or cumulative reward). In particular, in the early stage of the training, the safe rate is surprisingly high (i.e., the model is overall safe), we motivate that because the agent has not yet learned the task, consequently, it tends to stay still or move around the same point. Afterward, the safe rate starts to follow the trend of the success rate, reaching the best values, this is the fundamental phase, where the agent starts to learn the policy with a good generalization for unseen situations. Moreover, Fig. 6.5 highlights the problem described in the previous sections, the safe rate becomes unstable. A multitude of models with the same performance for the standard metric obtains significantly different values from the safety point of view. Moreover, our results show an overall drop in the safety of the models in the last stage of the training, in general, the agents start to learn shorter paths, taking more dangerous actions to maximize the reward. This further motivates the need to use additional evaluation metrics, before the deployment in a real-world scenario. Finally, Table 6.2 shows that ProVe can also be applied to the standard properties, providing a performance comparison between ProVe and Neurify [Wang et al., 2018a]. On average, ProVe achieves a speedup up of *20x* over Neurify. In particular, we found a huge improvement on the most time-demanding properties (i.e. $\theta_1$, $\theta_2$, $\theta_{15}$) while we obtain slightly worse performance on the simplest ones. We motivate this as our approach requires loading all the data on the GPU memory before the operations, and this is a bottleneck on simple (and fast to verify) properties.

**Simple Controller** Due to the low violation rate of our properties, it is possible to design a simple controller to guarantee the correct behavior of the network. To illustrate this, we describe the process to decide whether the controller can be designed for the trajectory generation environment. From the manipulator data sheet, a step of 2 degrees (i.e. the $\omega$ value of our controller) requires $\approx$ 0.01s to be executed by the arm and, with the violation rate presented in Fig. 6.5, a complete search through the array of the sub-areas that cause a violation, always requires less than 0.01s. Ideally, this means that we can verify if the input state leads to a violation at each iteration, without lags in the robot operations (notice that this depends on the hardware). In contrast, we computed that with a violation rate of $\approx$ 12%, a complete search requires approximately 1.02s, making our solution unfeasible without operational lags. Clearly, the application of a simple controller to guarantee the correct behavior under a certain property is limited by many factors, such as the nature of the task and the characteristic of the agent.

# Chapter 7

# TIME-DEPENDENT PROPERTIES

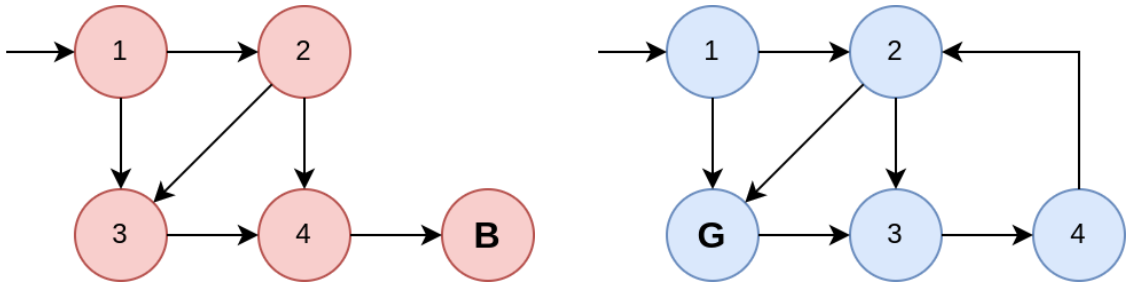
In the previous chapters, we introduced the concept of formal verification for neural networks (FV). In particular, in Chap. 5 we described the general objective of an FV algorithm, providing an intuition on the different approaches for this class of methods, while in Chap. 6 we moved in the direction of the formal verification for Deep Reinforcement Learning (DRL), focusing on the decision-making problem and introducing a novel approach to compute the violation rate, an evaluation metric for the safety of DRL-based systems. However, previous methods focus on the verification of safety properties that encode only input-output relations, and this presents different limitations when applied in a DRL context: (i) the focus is only on local properties, which means that the properties can describe only the current state of the system; (ii) it considers only the immediate consequences of an action; (iii) it is based on single invocations of the DNN, whereas DRL policies involve *sequences* of invocations and interactions with the environment; and (iv) this encoding requires a high prior knowledge from the users, that must be aware of the long term consequence of the past actions.

To overcome these limitations, a novel family of verification strategies has been developed for the verification of *long-term properties* (or *time-dependent properties*) [Amir et al., 2021; Eliyahu et al., 2021]. In our work, we propose to extend these approaches to a robotic DRL context, where sequences of actions and interactions with the environment characterize the agent's behavior (in Part III, we show a concrete case study). We combined FV approaches with different methodologies from the model-checking community, subdividing the properties into two classes of problems: *safety* and *liveness*. These two classes are sufficiently expressive to encode a large variety of requirements in many settings and, specifically for the primary goal of this dissertation, for DRL systems [Baier and Katoen, 2008].

- A *safety property* encodes that *nothing bad happens in the system* [Eliyahu et al., 2021], Fig. 7.1 shows an example; more formally a safety property is described by a function $B(x)$ that returns true if $x$ is a *bad* state. For example, in the surgical robot case studies, we

analyzed in this thesis, a bad state can be a configuration where the end effector touches the patient. Given $B(x)$, a violating run of the system is a finite sequence of states $x_0, ..., x_n$ such that, starting from an initial state $x_0$, ends up in a state $x_n$ such that $B(x_n)$ is true. Following this definition, the system of Fig. 7.1(a) is unsafe because the sequence $1, 2, 4, B$ ends up in the unsafe state $B$.

- A *liveness property* encodes that *good thing eventually happen* [Eliyahu et al., 2021]. Fig. 7.1 shows an example; more formally, a liveness property is described by a function $G(x)$ that returns true if $x$ is a *good* state, i.e., a state where the system is supposed to be at some point in the execution. Following the previous example of the surgical robot, a good state can be where the robot completed the task. Finally, given $G(x)$, a violating run of the system is a sequence of states $x_0, ..., x_n$ such that the sequence is a loop and does not exist a state $x_i$ of the sequence where $G(x)$ is true. Following this definition, the system of Fig. 7.1(b) is unsafe because the sequence $1, 2, 3, 4, 2, 3, 4, ...$ ends up in a loop without reaching the state $B$.




(a) Safety Property (b) Liveness Property

Figure 7.1: Examples of violated *safety* and *liveness* properties.

In the next sections, we show how to encode these kinds of problems in a formal fashion and how to translate this encoding into a formal verification query for a standard formal verifier. In our examples, we rely on Marabou [Katz et al., 2019], but other tools can be used as a backend. A crucial challenge to applying these approaches in a robotic context is related to the nonlinearity of these systems. A multi-step verification tool inherits the limitations of an optimization-based FV algorithm and can thus handle only linear constraints to describe the relation between to states of the system (i.e., the transition model). Apparently, this limitation prevents to merge of these two worlds (i.e., robotics and multi-step verification); however, in our work, we introduce two strategies to overcome these limitations: the *overapproximation* of the constraints and the *design-for-verification* approach. In this chapter, we provide a theoretical analysis of the approach while in Part III, we exploit this method in a real-world robotic problem, showing some practical examples of formal encoding. To summarize, in this chapter, we made

the following contribution in the direction of allowing the adoption of the multi-step formal verification in a robotic context: (i) we provide a formalism to encode multi-step requirements in a reactive robotic system; (ii) we show how to relax the requirements, and how these linear properties can be adapted in a robotic context; and (iii) we explain how the results of the analysis can be used in DRL contexts.

## 7.1 ENCODING

Before defining the formal encoding of the time-dependent properties, which we refer to as *model checker* representation, it is important to clarify a concept: this encoding is slightly different with respect to the standard definition of a formal verification problem. A state in this context is not the same state that constitutes the input of the DNN but can be a single state, an action, a state-action pair, or even a combination of these elements. For example, if the requirement is not to repeat the same action $k$ times, a state for the model checker represents a counter for the number of repetitions. Notice that there is always a strict relation between the model checker and the formal verification representations because a fast translation is required to run the query on a formal verification tool (e.g., Marabou [Katz et al., 2019]).

A model checking query for a time-dependent property requires a tuple $\langle S, I, T, p \rangle$, where:

- $S$ is the state space for the system. The state space is strongly related to the properties we aim to verify. For example, in some cases, it overlaps the state of the underlying MPD, or in others, it encodes a pair action-state.

- $I$ is a predicate to represent the initial states, $I(x)$ returns true if $x \in S$ and $x$ is an initial state. This is an important requirement because, for example, if a sequence of states that ends up in an unsafe state is only reachable starting from a state which is not initial, the system can be considered safe.

- $T$ is the transition relation function. Notice that this is different from a transition model of an MDP. $T(x, x')$ is a function that specifies if $x'$ can be reached from $x'$ in a single step in the system.

- $p$ is the property to verify, this predicate contains the requirements for the analysis, e.g., $B(x)$ for a safety property and $G(x)$ for a liveness property.

Finding a violation for a time-dependent property is the task of deciding if a sequence of action that respects some requirements exists. More formally, in the case of a *safety property*:

$$\exists x_1, ..., x_k \,.\, I(x_1) \wedge (\bigwedge_{i=1}^{k-1} T(x_i, x'_{i+1})) \wedge (\bigvee_{i=1}^{k} B(x_i)) \tag{7.1}$$

and, in the case of a *liveness property*:

$$\exists x_1, ..., x_k \,.\, I(x_1) \wedge (\bigwedge_{i=1}^{k-1} T(x_i, x'_{i+1})) \wedge (\bigwedge_{i=1}^{k} \neg G(x_i)) \wedge (\bigvee_{i=1}^{k-1} x_k = x_i) \tag{7.2}$$

Eq.7.1 and Eq.7.2 encode the model checking problem. Before the formal verification step, it is necessary to convert this formulation into the tuple $\langle N, P, Q \rangle$ that represents the verification query and exploit a solver-based verification tool. The idea is to duplicate the DNN to generate a new larger DNN composed of $k$ copies of the original one (where $k$ is the number of steps in the future we are considering for the verification query). Each input of the DNN is a variable, and the relation between the output and the next state, described by the relation function $T(x, x')$, is encoded using an implication constraint ($\rightarrow$), typically supported by state of the art SMT-solvers.

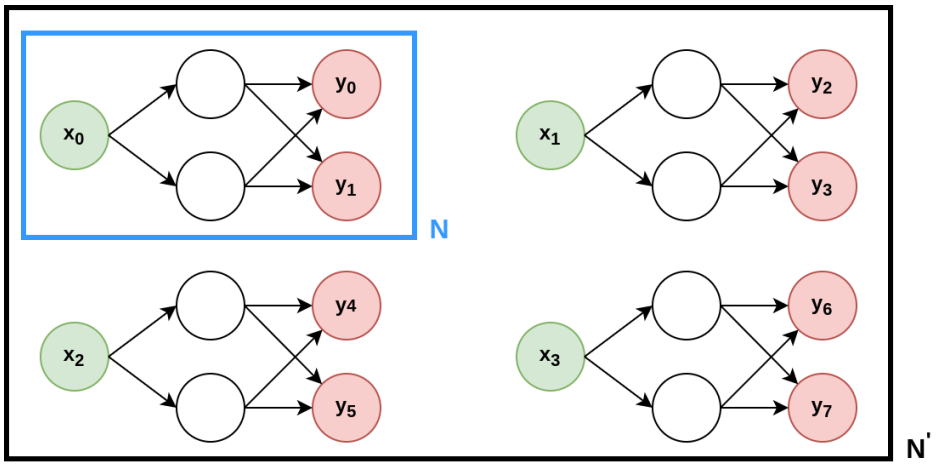



Figure 7.2: Generation of the larger neural network for the time-dependent verification. The original neural network has 1 input and 2 and is represented in the blue box ($N$). The larger network consists of 4 copies of the original one and is represented in the black box ($N'$).

Fig 7.2 provides a toy example for a safety property. Suppose we have a reactive system with a counter that constitutes the state, and only two actions are allowed: (i) increment the counter and (ii) decrement the counter. The controller is encoded with a DNN with 1 input and 2 outputs, and the initial state is when the counter is set to 0. The safety property is to avoid a state where the counter is greater than 3. Supposing that the action represented by $y_0$ increments the counter, and the action represented by $y_1$ decrements the value, the controller always selects the node with the highest value. We also suppose to set the limit for the number

of steps to analyze $k = 4$. The model-checking query can be formalized as follows:

$$\exists x_1, x_2, x_3, x_4 \,.\, I(x_1) \wedge (\bigwedge_{i=1}^{3} T(x_i, x'_{i+1})) \wedge (\bigvee_{i=1}^{4} B(x_i)) \tag{7.3}$$

where $I(x)$ returns true if the counter is 0, $T(x, x^{'})$ return true in two cases: (i) if the action is *increment counter* then $x^{'} = x + 1$ or (ii) if the action is *decrement counter* then $x^{'} = x - 1$, and $B(x)$ returns true if the counter is greater than 2. In such configuration, the formal verification query $\langle N^{'}, P, Q \rangle$, where $N^i$ consists in $k$ copies of the original network $N$ (Fig 7.2), is in the following form:

$$P : (x_0 = 0) \wedge (y_0 > 0 \rightarrow x_1 = x_0 + 1) \wedge (y_2 > y_3 \rightarrow x_2 = x_1 + 1) \wedge (y_4 > y_5 \rightarrow x_3 = x_2 + 1)$$

$$Q : (y_6 > y_7)$$

this query can be directly run on a verification engine, if the FV process return `SAT` it means that a counterexample has been found, i.e., a specific sequence of input/output that violates the property exists (in the toy example, it is possible that the counter is greater than 3). In contrast, if the FV tool return `UNSAT`, such configuration does not exist, and then the property is respected. This statement is true if we assume that $T(x, x^{'})$ describes exactly the transition function. In general, this function is difficult to obtain, the idea is to use an *abstraction*; $T(x, x^{'})$ can be an over-approximation of the actual transition function. Using an abstraction, the method is still sound but loses its completeness. This means that if the process return `UNSAT` the system is probably safe; however if it returns `SAT` the configuration may not be feasible and must be tested. Crucially, for safety-critical systems, a conservative approach is preferable; therefore, in our experiments (exhaustively described in Part III), we always look for `UNSAT` configurations. Finally, several techniques and additions derived from the model-checking community have been developed to improve performance and scalability when applied to real-world applications. Among the others, we mention the *k-induction*, *invariant inference*, and *abstraction*. We refer to the work of Amir et al. [2021] for more details.

## 7.2 APPLICATION TO ROBOTICS

In the introduction of this chapter, we introduced the main challenge for the application of time-dependent FV in a robotic context: *the non-linearity of the systems*. To exploit the multi-step verification, the first step is to encode the transition between the states of the system through a safety property in the standard form (e.g., Eq. 7.1 or Eq. 7.2). However, to be verified with an optimization-based method, the constraints must be linear (or piece-wise linear)[1]. While this is

[1]This limitation is inherited from the verification framework used as backend, which means that our method can take advantage of future development in this direction.

not a hard requirement in an abstract context (e.g., chess or videogame), it becomes crucial in a real-world robotic context, where the dynamic is typically non-linear, making the codification of the transition model for the FV queries untractable. In order to solve this problem, we propose to exploit an abstraction of the transition model. The intuition is to introduce a linear relaxation of the transition model to *"overapproximate"* the set of states that can be reached from an initial state with only one step. Exploiting this technique, we can encode the complex non-linear behavior of the robot. However, the verification becomes *non-complete*, which means that *if the FV process says that the robot is safe, this is formally true*, but if a counterexample is found, it can be a false positive (i.e., it does not respect the actual non-linear transition model). In general, given the conservative nature of formal verification, we argue that it can be considered a good result; in fact, the algorithm never returns *safe* if the robot does not respect the requirements, which is the most dangerous scenario. On the other hand, the abstraction should be a good approximation of the actual transition model. Otherwise, the algorithm is always prone to return false-positive counterexamples (e.g., in the borderline case, the transition model allows any transition, resulting in a *always usanfe* answer).

While completely avoiding the linear relaxation of the transition model is practically not possible in a robotic context, different techniques can be exploited to mitigate the previously discussed overhead. A possible option is to exploit a linearization technique (e.g., first-order Taylor expansion) to generate a piece-wise approximation of the transition model to locally set an upper and a lower bound to the transition function. Although this approach can limit the introduced overhead, it intrinsically requires an approximation and can not completely solve the problem; nevertheless, this approach should be further investigated, and we plan to work on this topic as a future direction for this work. Here, we propose an alternative solution that can drastically reduce the number of relaxation required, the *"design-for-verification"* approach. The intuition is to directly model the neural network controller to guarantee linear constraints between consecutive steps. For example, by fixing the linear velocity of a robot and supposing to have as input its coordinate, we obtain a linear constraint between the state at time $t$ and the state at time $t+1$ for a single movement action (i.e., if the robot selects the action *forward*, the $x$ coordinate will be increased by the linear velocity multiplied by the decision frequency). However, This approach does not allow for complete avoidance of abstraction but can help to reduce the number of non-linear transitions. In the next chapter, we will provide a concrete example where this approach allows us to find formally safe models before deployment.

## 7.3 EXPLOITING THE VERIFICATION

Now that we presented a method to formally verify *time-dependent properties*, the following step is to understand how to exploit these results to improve the reliability of the trained DRL policies. The primary motivation is shared with the standard FV methods, which we already discussed in Chap. 5, and it is to guarantee some requirements specified by expert users before

the deployment. However, as we discussed in Chap. 6, in a DRL context, formally guaranteeing that a policy *never* violates the properties is a strict requirement, especially because the multi-step verification is based on the SAT solver methods where the answer is always binary (`SAT` or `UNSAT`) and a violation rate can not be easily computed.

An available option is to exploit the counterexamples found during the verification process to enrich the memory buffer and improve the training performance from a safety perspective. However, (i) this approach can not be applied an on-policy algorithm (e.g., REINFORCE Sutton et al. [1999] and its derivations) because the memory buffer can contain only experiences collected with the current policy (a fundamental condition for the policy gradient theorem, consult Chap. 2 for details); and, (ii) in the case of an off-policy algorithm (e.g., DQN Mnih et al. [2013]) where we are allowed to manipulate the memory buffer, there are no guarantees about the policy improvements.

Nevertheless, from our experiments, and recalling the theoretical results of Chap. 6, we found that a high cumulative reward does not trivially imply respect for the safety constraints, and thus we believe that the result of a more formal analysis should be exploited to complement the evaluation phase before the deployment. In this direction, we propose two alternative methods to exploit the FV that do not directly affect the training loop.

1. *Termination Criteria:* the idea is to use this formal verification process to decide when the training can be considered terminated. The idea is to perform an FV step when the policy provides good performance (e.g, reward above a given value) and, if the FV returns `UNSAT` (i.e., the policy is safe), the training can be concluded. Otherwise, the policy must be improved and the loop continues.

2. *Model Selection:* the idea is to collect a set of models from the training loop, ideally from the final stages of the training, and use the FV to filter the unsafe models to obtain the policies that: (a) obtained good performance in training (e.g., reward-wise); and (b) does not violate the safety requirements (i.e., the verification returns `SAT`).

Although both these approaches can be helpful and do not imply theoretical problems when applied to a DRL loop, from our experiments, we noticed that the *model selection* has more impact. The reason is that using this analysis as termination criteria requires temporary pausing the training process, which introduces significant overhead and requires multiple invocations of the verification tool (e.g., Marabou Katz et al. [2019]). Crucially, this can not be done in parallel to the training. Model selection, in contrast, does not interfere with the training. Given some standard criteria (for example, the reward should be above a given threshold), all the successful models can be stored. Each of them requires only a single invocation of the verifier to filter the safe and unsafe policies. Therefore a good solution is to proceed with a multi-step pipeline: (i) perform the training until convergence and saving a set of good models; (ii) perform the model selection on these models using the formal verification, and (iii) among the *survived* models, perform an additional empirical testing phase behavioral oriented, to find the best policy

between the models that we found to be provably safe. In Part III) of this thesis, we show how this approach is useful in a real-world robotic problem, providing a set of well-performing and provably safe models.

# Part III

# Application to Robotics

# Chapter 8

# THE MAPLESS NAVIGATION CASE STUDY

In this chapter, we apply the main contributions of this thesis to a real-world robotic case study: the *Mapless Navigation* problem. The objective of this chapter is to show a concrete example of how to generate agents that respect a set of given requirements, are provably safe, and at the same time, obtain excellent results in solving the task. In this section, we make the following contributions: (i) we present and analyze a real-world problem, formulating the safety and behavioral requirements; (ii) exploiting the method proposed in Chap. 4, we show how to perform constrained training, injecting domain expert knowledge to enforce behavioral requirements in the generated policy; and (iii) we perform a model selection phase, exploiting the formal verification approaches proposed in Chap. 6 and Chap. 7, to filter the provably safe agents; here we consider both the single-step verification and the time-dependent properties. We refer to the requirements for the constrained training as *soft constraints*, which means that we tolerate some deviations from these expected behaviors as long as they stay below a given threshold. In contrast, we consider the requirements for the formal verification as *hard constraints*, which must be formally proved with a verification algorithm before the deployment. Finally, we show our results, demonstrating that, following the pipeline presented in this dissertation, the resulting agents are safe and reliable without compromising the performances. Crucially, we perform our experiments on a realistic simulator (developed with the Unity3D engine and the Robotic Operating System[1]) before the deployment of the actual robotic platform.

## 8.1 PROBLEM DEFINITION AND REQUIREMENTS

We explain and demonstrate our proposed techniques applying our pipeline to the *mapless navigation* problem, in which a robot is required to reach a given target efficiently while avoiding collision with obstacles. Unlike in classical planning, the robot is not given a map of its surround-

[1]Refer to Chapter 2 for details about the simulation.

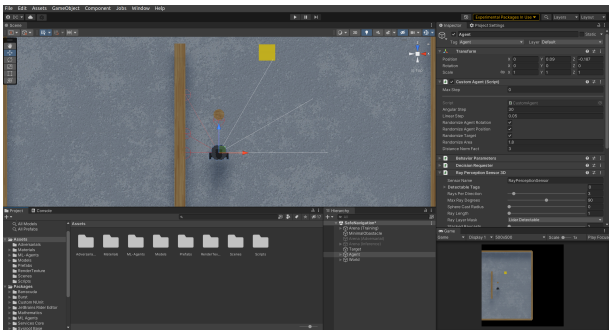
Figure 8.1: The Robotis Turtlebot3 platform.

ing environment and can rely only on local observations (e.g., from lidar sensors or cameras). Thus, a successful agent needs to be able to adjust its strategy dynamically as it progresses toward its target. Mapless navigation has been studied extensively and is considered particularly challenging to solve. Specifically, the local nature of the problem renders learning a successful policy extremely challenging and hard to solve using classical algorithms [Pfeiffer et al., 2018]. Prior work has shown DRL approaches to be among the most successful for tackling this task, often outperforming hand-crafted algorithms [Marchesini and Farinelli, 2020].

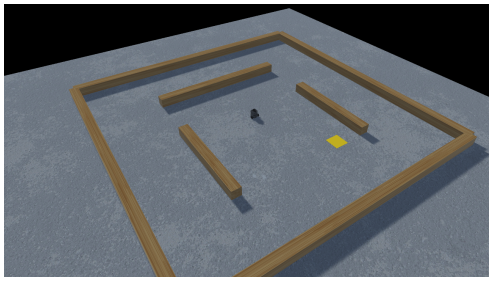
As a platform for our study, we relied on the *Robotis Turtlebot 3* platform (Turtlebot, for short; see Fig. 8.1), which is widely used in the community [Nandkumar et al., 2021; Amsters and Slaets, 2019]. The Turtlebot is capable of horizontal navigation and is equipped with lidar sensors for detecting nearby obstacles. In order to train DRL policies for controlling the Turtlebot, we built a simulator based on the *Unity3D* engine [Juliani et al., 2018] (Fig. 8.2 shows some screenshots of the simulator). Unity3D is compatible with the *Robotic Operating System* (ROS) [Quigley et al., 2009] and allows a fast transfer to the actual platform (*sim-to-real* [Zhao et al., 2020]). We used a hybrid reward function, which includes a discrete component for the terminal states ("collision", or "reached target"), and a continuous component for the non-terminal states. Formally:

$$R_t = \begin{cases} \pm 1 & \text{terminal states} \\ (dist_{t-1} - dist_t) \cdot \eta - \beta & \text{otherwise} \end{cases} \tag{8.1}$$

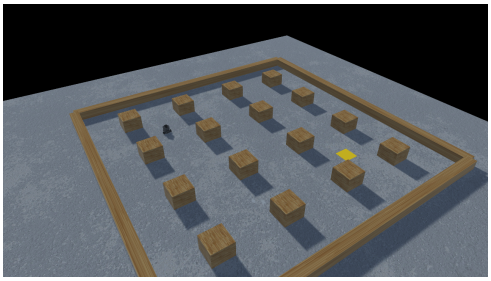
Where $dist_k$ is the distance from the target at time $k$; $\eta$ is a normalization factor; and $\beta$ is a penalty, intended to encourage the robot to reach the target quickly (in our experiments, we empirically set $\eta = 3$ and $\beta = 0.001$). Additionally, in terminal states, we increase the reward by 1 if the target is reached, or decrease it by 1 in case of collision. For our DNN topology,



(a) Unity3D engine



(b) basic environment

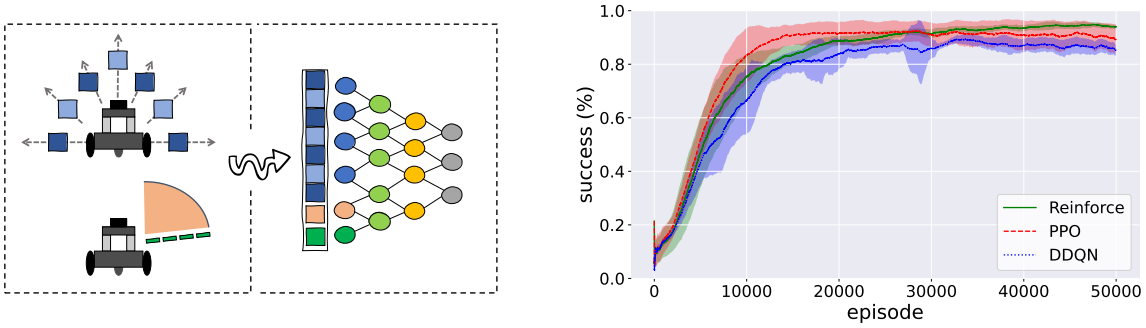


(c) additional environment

Figure 8.2: The *Unity3D* game engine with some examples of our simulation environments.

we used an architecture that was shown to be successful in a similar setting [Marchesini and Farinelli, 2020]:

- An input layer with nine neurons. These include seven neurons representing the Turtlebot's lidar readings. The additional, non-lidar inputs include one neuron representing the relative angle between the robot and the target, and one neuron representing The robot's distance from the target. A scheme of the inputs appears in Fig. 8.3a.
- Two subsequent fully-connected layers, each consisting of 16 neurons, followed by a ReLU activation layer. In Marchesini and Farinelli [2020] the authors focused on a DNN with a total of 64 hidden neurons. For our experiments, we trained DNNs of various sizes and eventually focused on DNNs with a total of 32 hidden neurons, as they achieved similar accuracy and allowed us to expedite verification.
- An output layer with three neurons, each corresponding to a different (discrete) action that the agent can choose to execute in the following step: move `FORWARD`, turn `LEFT`, or turn `RIGHT`. It has been shown that discrete controllers achieve excellent performance in robotic navigation, often outperforming the continuous controllers in a large variety of tasks [Marchesini and Farinelli, 2020].



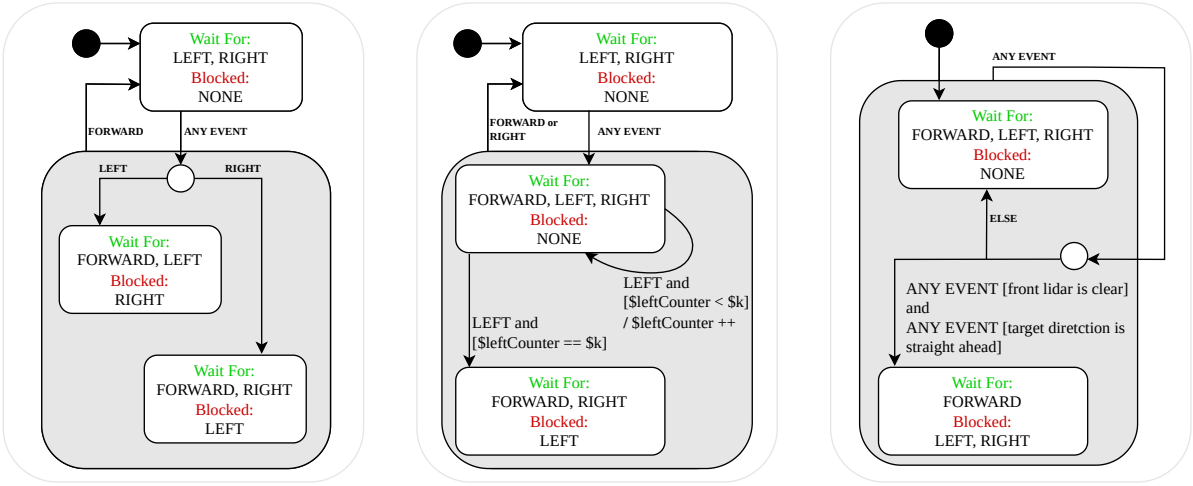

(a) The DRL controller

(b) Average success rates

Figure 8.3: (a) The DRL controller used for the robot in the case study; the DRL model has nine input neurons: seven lidar sensor readings (blue), one input indicating the relative angle (orange) between the robot and the target, and one input indicating the distance (green) between the robot and the target. (b) The average success rates of models trained by each of the three DRL training algorithms, per training episode. The plot also indicates the standard deviation for each algorithm.

Using some of the training algorithms mentioned in Chap 2, we trained a collection of DRL agents to solve the Turtlebot mapless navigation problem. We ran a stochastic training process, and therefore we obtained varied agents; of these, we only kept those that achieved a success rate of at least 96% during training, where "success" means that the robot reached its target without colliding with walls or obstacles. A total of 780 models were selected, consisting of 260 models per each of the three training algorithms. More specifically, for each algorithm, all 260 models were generated from 52 random seeds. Each seed gave rise to a family of 5 models, where the individual family members differ in the number of training episodes used for training them. Fig. 8.3b shows the trained models' average success rate, for each algorithm used. We note that PPO was generally the fastest to achieve high accuracy. However, all three training algorithms successfully produced highly accurate agents.

**Soft Constraints - Behavioural Requirements** Analyzing the trained agents further, we observed that even DRL agents that achieved a high success rate may demonstrate highly undesirable behavior in different scenarios. One such behavior is a sequence of back-and-forth turns, that causes the robot to waste time and energy. Another undesirable behavior is when the agent makes a lengthy sequence of right turns instead of a much shorter sequence of left turns (or vice versa), wasting, again, time and energy. A third undesirable behavior that we observed is that the agent might decide not to move forward toward a target that is directly ahead, even when the path is clear. Our goal was thus to use our approach to remove these undesirable behaviors.




(a) *avoid back-and-forth rotation* (b) *avoid turns larger than 180°* (c) *avoid turning when clear*

Figure 8.4: A visualization of the three scenarios. Figure (b) refers to the *Left turns part* only. 'Wait For' and 'Blocked' in the state-blob indicates events that the scenario waits for or blocks, respectively. The events `SBP_MoveForward, SBP_TurnLeft and SBP_TurnRight` are represented respectively, by FORWARD, LEFT, RIGHT.

**Hard Constraints - Safety Requirements** All of our trained models achieved very high success rates, and so, at face value, there was no reason to favor one over the other. However, as we show next, a verification-based approach can expose multiple subtle differences between them. As our evaluation criteria, we define some fundamental properties of interest that derive from the main goals of the robotic controller: (i) reaching the target; and (ii) avoiding collision with obstacles. Using verification, we use these criteria to identify models which may fail to fulfill their goals, i.e., because they collide with various obstacles, are overly conservative, or may enter infinite loops without reaching the target. Here, we remark that all the selected policies never collided during an empirical evaluation analysis, however, some of them are susceptible to specific adversarial corner cases (refer to Chap. 5 for more details about this concept).

## 8.2 SCENARIO BASED CONSTRAINED DRL

Following the approach presented in Chap. 4, we integrated a scenario-based program into the DRL training process, in order to remove the aforementioned undesirable behaviors. More concretely, we created specific scenarios to rule out each of the three aforementioned undesirable behaviors we observed. To accomplish this, we created a mapping between each possible action $a_t \in$ {Move `FORWARD`, Turn `LEFT`, Turn `RIGHT`} of the DRL agent and a dedicated event $e_{a_t} \in$ {SBP_MoveForward, SBP_TurnLeft, SBP_TurnRight} within the scenario-based program.

These events allow the various scenarios to keep track of (and react to) the agent's actions. We refer to these $e_{a_t}$ events as *external events*, indicating that they can only be triggered when requested from outside the SB program proper. By convention, we assume that after each triggering of a single, external event, the scenario-based program executes a sequence of internal events (a *super-step* [Yerushalmi et al., 2022]), until it returns to a steady state and then waits for another external event. The novelty of our approach is in the strategy by which we use scenarios to affect the training process. Specifically, we define the DRL cost function to correspond to violations of scenario constraints by the DRL agent. Whenever the agent selects an action that is mapped to a *blocked* SBP event, we increase the *cost*. This approach is described in detail in Chap 4, and constitutes a general and scalable method for injecting explicit constraints (expressed, e.g., by scenarios) directly into the policy optimization process.

Considering our Turtlebot mapless navigation case study, we created scenarios for discouraging the three undesirable behaviors we had previously observed. The scenarios are visualized in Fig. 8.4, using an amalgamation of Statecharts and SBP graphical notation languages [Harel, 1987; Marron et al., 2018]. The Python implementation of the three scenarios used in this paper is shown below: the code for *avoid back-and-forth rotation* appears in Listing 8.1, the code for *avoid turns larger than 180°* appears in Listing 8.2, and the code for *avoid turning when clear* appears in Listing. 8.3.

```
def SBP_avoidBackAndForthRotation():
    blockedEvList = []
    waitforEvList = [BEvent("SBP_MoveForward"),
                     BEvent("SBP_TurnLeft"),
                     BEvent("SBP_TurnRight")]
    while True:
        lastEv = yield {waitFor: waitforEvList, block: blockedEvList}
        if lastEv != BEvent("SBP_TurnLeft")
            and lastEv != BEvent("SBP_TurnRight"):
            blockedEvList = []
        else:
            blocked_ev = BEvent("SBP_TurnRight")
                if lastEv == BEvent("SBP_TurnLeft")
                else BEvent("SBP_TurnLeft")
            # Blocking!
            blockedEvList.append(blocked_ev)
```

Listing 8.1: The Python implementation of scenario *avoid back-and-forth rotation*. The code waits for any of the possible events: *SBP_MoveForward, SBP_TurnLeft* and *SBP_TurnRight*. Upon receiving *SBP_TurnLeft*, it blocks *SBP_TurnRight*, and upon receiving *SBP_TurnRight*, it blocks *SBP_TurnLeft*. Upon receiving *SBP_MoveForward*, it clears any blocking.

```
def SBP_avoid_k_consecuative_turns():
    k = 7
    counter = 0
    prevEv = None
```

```
    blockedEvList = []
    waitforEvList = [BEvent("SBP_MoveForward"), BEvent("SBP_TurnLeft"), \\
     BEvent("SBP_TurnRight")]
    while True:
        lastEv = yield {waitFor: waitforEvList, block: blockedEvList}
        if prevEv is None or lastEv == BEvent("SBP_MoveForward") \\
         or prevEv != lastEv:
            prevEv = lastEv
            counter = 0
            blockedEvList = []
        else:
            if counter == k - 1:
                # Blocking!
                blockedEvList.append(lastEv)
            else:
                counter += 1
```

Listing 8.2: The Python implementation of a scenario that blocks turning in the same direction more than k consecutive times. Each turn action rotates the robot by 30°, and so we set k to be 7.

```
def SBP_avoid_turning_when_clear():
    blockedEvList = []
    waitforEvList = [BEvent("SBP_MoveForward"), BEvent("SBP_TurnLeft"), \\
     BEvent("SBP_TurnRight")]
    while True:
        lastEv = yield {waitFor: waitforEvList, block: blockedEvList}
        state = lastEv.data['state']
        if state[3] > MINIMAL_FWD_CLEARANCE and \\
         state[2] > MINIMAL_CLEARANCE and \\
         state[4] > MINIMAL_CLEARANCE and \\
         abs(FWD_DIR - state[-2]) < FWD_DIR_TOLERANCE:
            blockedEvList.extend([BEvent("SBP_TurnLeft"), \\
             BEvent("SBP_TurnRight")])
        else:
            blockedEvList = []
```

Listing 8.3: The Python implementation of a scenario that blocks turning if the target is straight ahead and the path toward it is clear. The event carries data with it, which includes readings from the seven lidar sensors (with state[3] being the front-heading sensor. State[-2] is the direction to the target).

1. Scenario *avoid back-and-forth rotation* (Fig. 8.4a) seeks to prevent in-place, back-and-forth turns by the robot, to conserve time and energy.

2. Scenario *avoid turns larger than 180°* (Fig. 8.4b) seeks to prevent left turns in angles that are greater than 180°, to conserve time and energy (the right-turn case is symmetrical).

A forward slash indicates an action that is performed when a transition is taken; square brackets denote guard conditions, and $k and $leftCounter are variables. Each turn rotates the robot by 30°, and so we set $k = 7$.

3. Scenario *avoid turning when clear* (Fig. 8.4c) seeks to force the agent to move towards the target when it is ahead, and there is a clear path to it. This is performed by blocking any turn actions when this situation occurs. Triggered events carry data, which can be referenced by guard conditions.

**Evaluation** Fig. 8.5 depicts a comparison between policies trained with a standard end-to-end PPO [Schulman et al., 2017] (the baseline), and those trained using our constrained method with the injection of rules. In Figs. 8.5(a) and 8.5(d), we show results of policies trained with just *avoid back-and-forth rotation* added as a constraint. Fig. 8.5(a) shows that the success rate of the baseline stabilizes at around 87%, while the success rate of our improved policies stabilizes at around 95%. Fig. 8.5(d) then compares the frequency of undesired behavior occurrences between the baseline, at about 13 per episode, and our policies, where the frequency diminishes *almost completely*. Next, for Fig. 8.5(b) we show results of policies trained with all three of our added rules; we note that the success rate for these policies stabilizes around 95%, compared to 87% for the baseline. Finally, in Figs. 8.5(c), (e), and (f), we compare the frequency of the occurrence of undesired behaviors between the baseline and the policies trained with all rules active. Using the baseline, the frequency of the three behaviors is about 13, 3, and 17 per episode. The undesired behaviors are removed *almost completely* for the policies trained with our additional rules and method. We note that the undesired behavior addressed by the rule *avoid turns larger than 180°* is quite rare in general; and so the statistics reported in Fig. 8.5(c) were collected over the final 100 episodes of training.

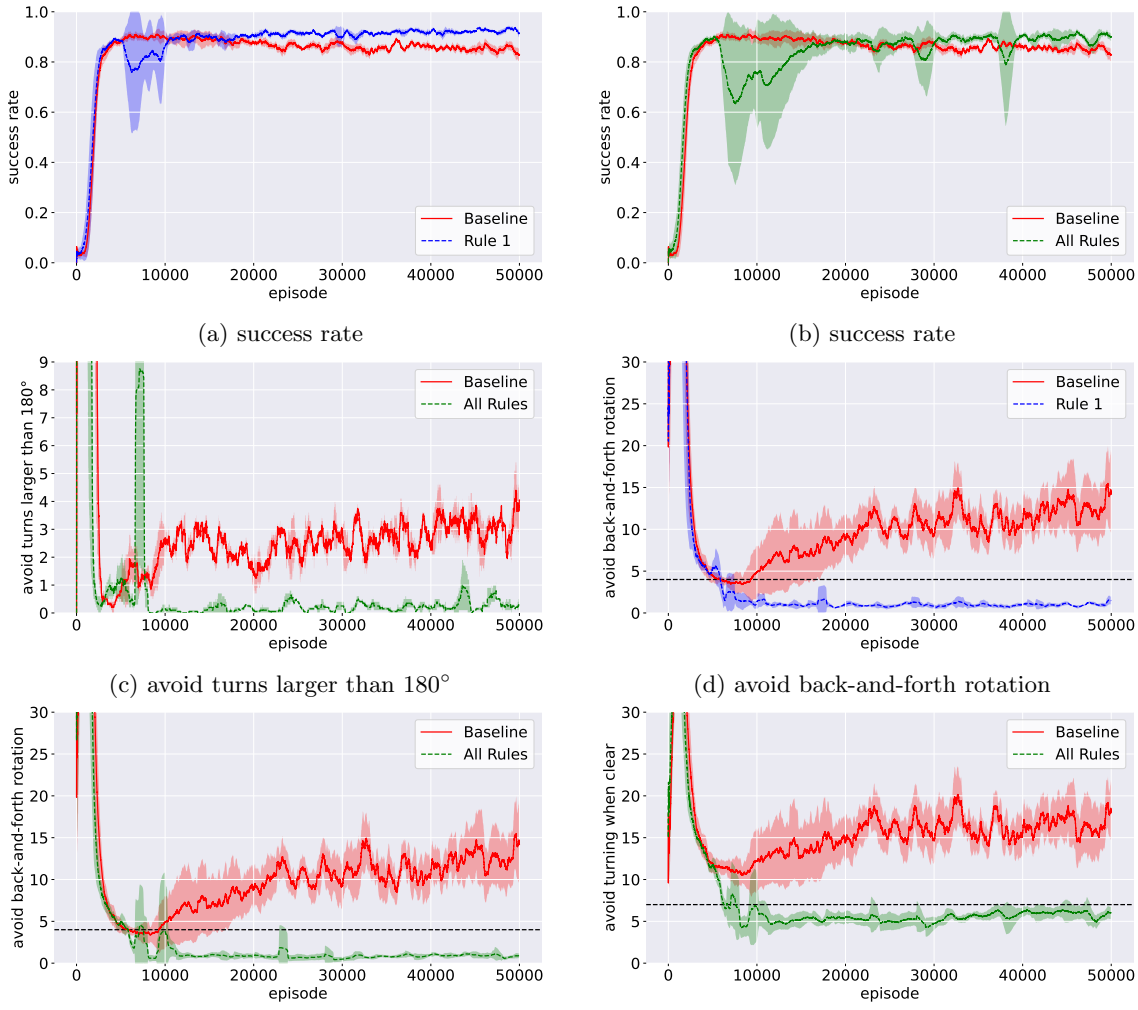
The results clearly show that our method can train agents that respect the given constraints, without damaging the main training objective — the success rate. Moreover, it also highlights the scalability of our method, i.e., performing well when single or multiple rules are applied. Reviewing Fig 8.5(b), comparing the baseline's success rate with our method's success rate when all rules are applied together with all the optimizations presented in Chap. 4, shows a clear advantage. Excitingly, our approach even led to an improved success rate, suggesting that the contribution of expert knowledge can drive the training to better policies. This showcases the importance of enabling expert-knowledge contributions, compared to end-to-end approaches in the context of mapless navigation.

## 8.3 FORMAL VERIFICATION AND MODEL SELECTION

In this section, we discuss the safety requirements that we considered as *hard-constraints* for the "verification-based model selection". As discussed in the previous section we subdivided these requirements based on the two main goals of the robotic controller: (i) reaching the target; and

(ii) avoiding collisions. In addition, we consider an alternative way to use the model selection and the formal verification to filter some desirable behaviors, we finally discuss our *design for verification* approach.

**Collision Avoidance** Collision avoidance is a fundamental and ubiquitous safety property for navigation agents Clarke et al. [2018]. In the context of Turtlebot, our goal is to check whether




(a) success rate

(b) success rate

(c) avoid turns larger than 180°

(d) avoid back-and-forth rotation

(e) avoid back-and-forth rotation

(f) avoid turning when clear

Figure 8.5: A comparison between the baseline policies to policies trained using our approach. The black dotted line states the threshold ($d_k$) we considered for the $k^{th}$ rule.

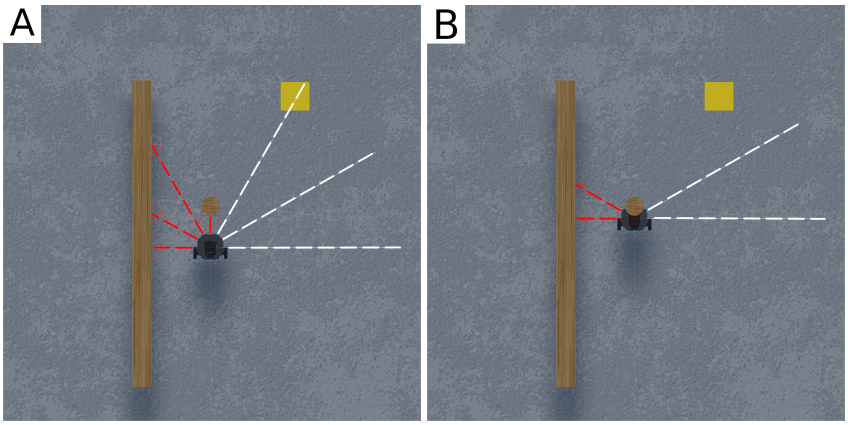



Figure 8.6: Example of a single-step collision. The robot is not blocked on its right and can avoid the obstacle by turning (panel A), but it still chooses to move forward and hence collides (panel B).

there exists a setting in which the robot is facing an obstacle and chooses to move straight ahead — even though it has at least one other viable option, in the form of a direction in which it is not blocked. In that situation, it is clearly preferable to choose turning `LEFT` or `RIGHT` instead of choosing to move `FORWARD` and colliding. See Fig. 8.6 for an illustration.

Given that turning `LEFT` or `RIGHT` produces an in-place rotation (i.e., the robot does not change its position), the only action which can cause a collision is `FORWARD`. In particular, a collision can happen when an obstacle is directly in front of the robot, or slightly off to one side (just outside the front lidar's field of detection). More formally, we consider the safety property *"the robot does not collide at the next step"*, with three different types of collisions:

- `FORWARD COLLISION`: the robot detects an obstacle straight ahead, but makes a single step forward and collides with the obstacle.
- `LEFT COLLISION`: the robot detects an obstacle up ahead and slightly shifted to the left (using the lidar beam that is 30° to the left of the one pointing straight ahead), but makes a single step forward and collides with the obstacle. The shape of the robot is such that in this setting, a collision is unavoidable.
- `RIGHT COLLISION`: the robot detects an obstacle up ahead and slightly shifted to the right, but makes a single step forward and collides with the obstacle.

Recall that in mapless navigation, all observations are local — the robot has no sense of the global map, and can encounter any possible obstacle configuration (i.e., any possible sensor reading). Thus, in encoding these properties, we considered a single invocation of the DRL agent's DNN, with the following constraints:

1. All the sensors that are not in the direction of the obstacle receive a lidar input indicating that the robot can move either `LEFT` or `RIGHT` without risk of collision. This is encoded by lower-bounding these inputs.

2. The single input in the direction of the obstacle is upper-bounded by a value matching the representation of an obstacle, close enough to the robot so that it will collide if it makes a move `FORWARD`.

3. The input representing the distance to the target is lower-bounded, indicating that the target has not yet been reached (encouraging the agent to make a move).

The exact encoding of these properties is based on the physical characteristics of the robot and the lidar sensors.

**Infinite Loops** Whereas collision avoidance is the natural safety property to verify in mapless navigation controllers, checking that progress is eventually made towards the target is the natural liveness property. Unfortunately, this property is difficult to formulate due to the absence of a complete map. Instead, we settle for a weaker proxy and focus on verifying that the robot does not enter infinite loops (which would prevent it from ever reaching the target).

Unlike the case of collision avoidance, where a single step of the DRL agent could constitute a violation, here we need to reason about multiple consecutive invocations of the DRL controller, in order to identify infinite loops. This, again, is difficult to encode due to the absence of a global map, and so we focus on *in-place* loops: infinite sequences of steps in which the robot turns `LEFT` and `RIGHT`, but without ever moving `FORWARD`, thus maintaining its current location ad infinitum. Our queries for identifying in-place loops encode that: (i) the robot does not reach the target in the first step; (ii) in the following $k$ steps, the robot never moves `FORWARD`, i.e., it only performs turns; and (iii) the robot returns to an already-visited configuration, guaranteeing that the same behavior will be repeated by our deterministic agents. The various queries differ in the choice of $k$, as well as in the sequence of turns performed by the robot. Specifically, we encode queries for identifying the following kinds of loops:

- `ALTERNATING LOOP`: a loop where the robot performs an infinite sequence of $\langle$ `LEFT`, `RIGHT`, `LEFT`, `RIGHT`, `LEFT`, ...$\rangle$ moves. A query for identifying this loop encodes $k = 2$ consecutive invocations of the DRL agent, after which the robot's sensors will again report the exact same reading, leading to an infinite loop. An example appears in Fig. 8.7. The encoding uses the "sliding window" principle, on which we elaborate later.

- `LEFT CYCLE`, `RIGHT CYCLE`: loops in which the robot performs an infinite sequence of $\langle$`LEFT`, `LEFT`, `LEFT`, ...$\rangle$ or $\langle$`RIGHT`, `RIGHT`, `RIGHT`, ...$\rangle$ operations. Because the Turtlebot turns at a 30° angle, this loop is encoded as a sequence of $k = 360°/30° = 12$ consecutive invocations of the DRL agent's DNN, all of which produce the same turning action (either

`LEFT` or `RIGHT`). Using the sliding window principle guarantees that the robot returns to the same exact configuration after performing this loop, indicating that it will never perform any other action. We also note that all the loop-identification queries include a condition for ensuring that the robot is not blocked from all directions. Consequently, any loops that are discovered demonstrate a clearly suboptimal behavior. An example appears in Fig. 8.8.

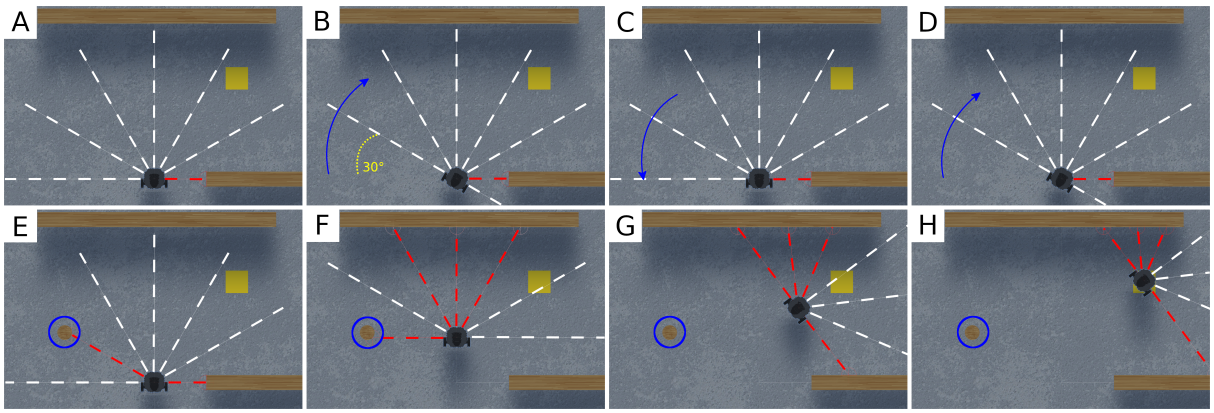



Figure 8.7: An example of a simulated Turtlebot entering a 2-step loop. The white and red dashed lines represent the lidar beams (white indicates "clear", and red indicates that an obstacle is detected). The yellow square represents the target position; the blue arrows indicate rotation. In the first row, from left to right, the Turtlebot is stuck in an infinite loop, alternating between right and left turns. Given the deterministic nature of the system, the agent will continue to select these same actions, ad infinitum. In the second row, from left to right, we present an almost identical configuration, but with an obstacle located 30° to the robot's left (circled in blue). The presence of the obstacle changes the input to the DNN, and allows the Turtlebot to avoid entering the infinite loop; instead, it successfully navigates to the target.

**Specific Behavior Profiles.** In our experiments, we noticed that the safety policies, i.e., the ones that do not cause the robot to collide displayed a wide spectrum of different behaviors when navigating to the target. These differences occurred not only between policies that were trained by different algorithms, but also between policies trained by the same reward strategy, indicating that these differences are, at least partially, due to the stochastic realization of the DRL training process.

Specifically, we noticed high variability in the length of the routes selected by the DRL policy in order to reach the given target: while some policies demonstrated short, efficient, paths that passed very close to obstacles, other policies demonstrated a much more "conservative" behavior, by selecting longer paths, and avoiding getting close to obstacles (an example appears in Fig. 8.9).

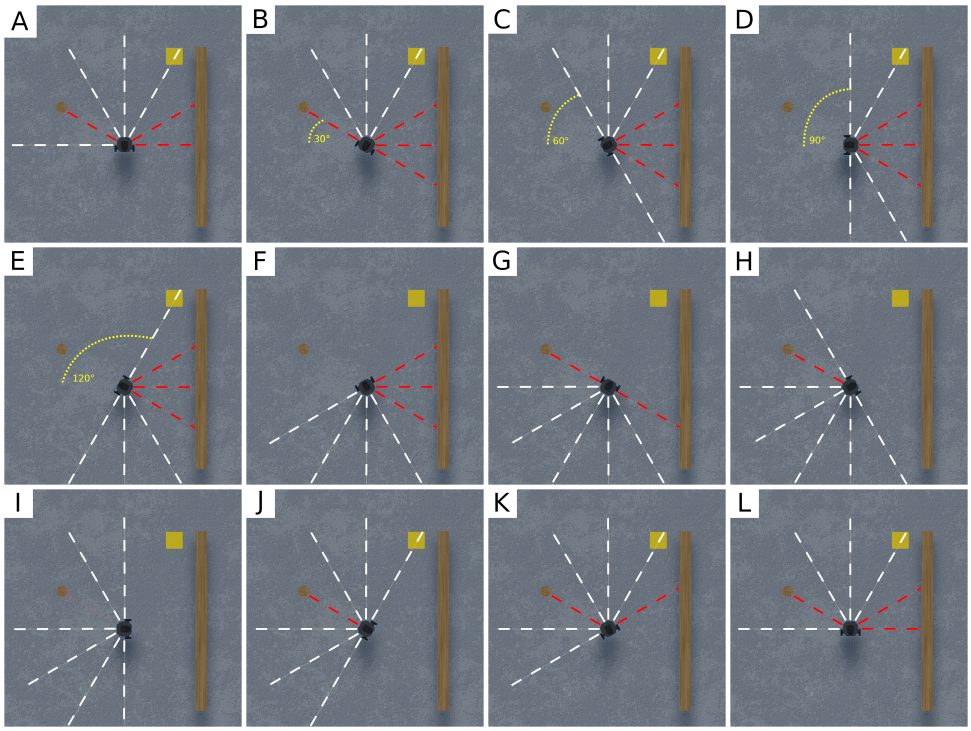



Figure 8.8: A 12-step trajectory of an infinite `RIGHT CYCLE`. The white and red dashed lines represent the lidar scan (white indicates a clear path; red indicates an obstacle is present), while the yellow square represents the target position towards which the robot navigates. The yellow dotted line represents the angular step the robot performs in the first six iterations. Given that the spacing between each lidar scan is the same as the angular step size (30°), at each time step it is possible to encode the sliding window for the state. The trajectory of a `LEFT CYCLE` is symmetric.

Thus, we used our verification-driven approach to quantify how "conservative" the learned DRL agent is in the mapless navigation setting. Intuitively, a highly conservative policy will keep a significant safety margin from obstacles (possibly taking a longer route to reach its destination), whereas a "braver" and less conservative controller would risk venturing closer to obstacles. In the case of Turtlebot, the preferable DRL policies are the ones that guarantee the robot's safety (with respect to collision avoidance), and demonstrate a high level of bravery — as these policies tend to take shorter, optimized paths (see path A in Fig. 8.9), which lead to reduced energy consumption over the entire trail.

Bravery assessment is performed by encoding verification queries that identify situations in which the Turtlebot *can* move forward, but its control policy chooses not to. Specifically, we encode single invocations of the DRL model, in which we bound the lidar inputs to indicate that the Turtlebot is sufficiently distant from any obstacle and can safely move forward. We

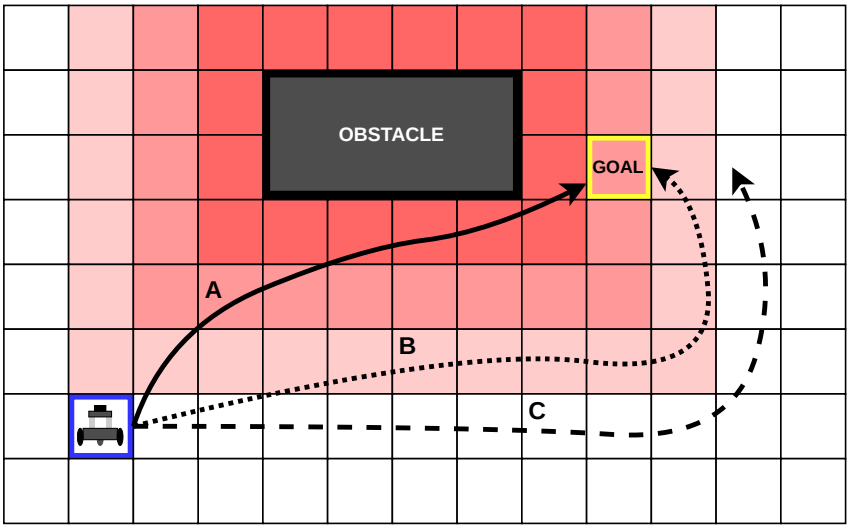



Figure 8.9: Comparing paths selected by policies with different *bravery* levels. Path $A$ takes the Turtlebot close to the obstacle (red area) and is the shortest. Path $B$ maintains a greater distance from the obstacle (light red area) and is consequently longer. Finally, path $C$ maintains such a significant distance from the obstacle (white area) that it is unable to reach the target.

then use the verifier to determine whether, in this setting, a `FORWARD` output is possible. By altering and adjusting the bounds on the central lidar sensor, we control how far away the robot perceives the obstacle to be. If we limit this distance to large values and the policy will still not move `FORWARD`, it is considered conservative; otherwise, it is considered brave. By conducting a binary search over these bounds Amir et al. [2021], we can identify the shortest distance from an obstacle, for which the policy orders the robot to move `FORWARD`, and this value's inverse serves as a bravery score for that policy.

**Design for Verification: Sliding Windows** A significant challenge that we faced in encoding our verification properties, especially those that pertain to multiple consecutive invocations of the DRL policy, had to do with the local nature of the sensor readings that serve as input to the DNN. Specifically, if the robot is in some initial configuration that leads to a sensor input $x$, and then chooses to move forward reaching a successor configuration in which the sensor input is $x'$, some connection between $x$ and $x'$ must be expressed as part of the verification query (i.e., nearby obstacles that exist in $x$ cannot suddenly vanish in $x'$). In the absence of a global map, this is difficult to enforce.

In order to circumvent this difficulty, we used the *sliding window* principle, which has proven quite useful in similar settings Eliyahu et al. [2021]; Amir et al. [2021]. Intuitively, the idea is to focus on scenarios where the connections between $x$ and $x'$ are particularly straightforward

to encode — in fact, most of the sensor information that appeared in $x$ also appears in $x'$. This approach allows us to encode multistep queries, and is also beneficial in terms of performance: typically, adding sliding-window constraints reduces the search space explored by the verifier, and expedites solving the query. In the Turtlebot setting, this is achieved by selecting a robot configuration in which the angle between two neighboring lidar sensors is identical to the turning angle of the robot (in our case, 30°). This guarantees, for example, that if the central lidar sensor observes an obstacle at a distance $d$ and the robot chooses to turn `RIGHT`, then at the next step, the lidar sensor just to the left of the central sensor must detect the same obstacle, at the same distance $d$. More generally, if at time-step $t$ the 7 lidar readings (from left to right) are $\langle l_1, \ldots, l_7 \rangle$ and the robot turns `RIGHT`, then at time-step $t+1$ the 7 readings are $\langle l_2, l_3, \ldots, l_7, l_8 \rangle$, where only $l_8$ is a new reading. The case for a `LEFT` turn is symmetrical. An illustration appears in Fig. 8.7. By placing these constraints on consecutive states encountered by the robot, we were able to encode complex properties that involve multiple time-steps, e.g., as in the case of infinite loop properties discussed previously.

## 8.4 RESULTS

Next, we ran verification queries with the aforementioned properties, in order to assess the quality of our trained DRL policies. The results are reported below. In many cases, we discovered configurations in which the policies would cause the robot to collide or enter infinite loops. All verification queries ran on a distributed cluster of HP EliteDesk machines, running at 3.00 GHz, and with a 32 GB memory. We used the sound and complete *Marabou* verification engine Katz et al. [2019] as our backend verifier (although other verification engines could be used, as well). The *Marabou* engine supports DNNs with ReLU layers, max-pooling, convolution, absolute value, and sign layers; and also supports sigmoids and softmax constraints.

**Model Selection** In this set of experiments, we used verification to assess our trained models. Specifically, we used each of the three training algorithms (DDQN, Reinforce, PPO) to train 260 models, creating a total of 780 models. For each of these, we verified six properties of interest: three collision properties (`FORWARD COLLISION`, `LEFT COLLISION`, `RIGHT COLLISION`), and three loop properties (`ALTERNATING LOOP`, `LEFT CYCLE`, `RIGHT CYCLE`). This gives a total of 4680 verification queries. We ran all queries with a `TIMEOUT` value of 12 hours and a `MEMOUT` limit of $2G$; the results are summarized in Table 8.1. The single-step collision queries usually terminated within seconds, and the 2-step queries encoding an `ALTERNATING LOOP` usually terminated within minutes. The 12-step cycle queries, which are more complex, usually ran for a few hours. 9.6% of all queries hit the `TIMEOUT` limit (all from the 12-step cycle category), and none of the queries hit the `MEMOUT` limit.

Our results exposed various differences between the trained models. Specifically, of the 780 models checked, 752 (over 96%) violated at least one of the single-step collision properties.

| | LEFT COLLISION | | FORWARD COLL. | | RIGHT COLL. | |
|---|---|---|---|---|---|---|
| Algorithm | SAT | UNSAT | SAT | UNSAT | SAT | UNSAT |
| DDQN | 259 | 1 | 248 | 12 | 258 | 2 |
| Reinforce | 255 | 5 | 254 | 6 | 252 | 8 |
| PPO | 196 | 64 | 197 | 63 | 207 | 53 |

| | ALTERNATING LOOP | | LEFT CYCLE | | RIGHT CYCLE | | INSTABILITY |
|---|---|---|---|---|---|---|---|
| Algorithm | SAT | UNSAT | SAT | UNSAT | SAT | UNSAT | # alternations |
| DDQN | 260 | 0 | 56 | 77 | 56 | 61 | 21 |
| Reinforce | 145 | 115 | 5 | 185 | 120 | 97 | 10 |
| PPO | 214 | 45 | 26 | 198 | 30 | 198 | 1 |

Table 8.1: Results of the policy verification queries. We verified six properties over each of the 260 models trained per algorithm; SAT indicates that the property was violated, whereas UNSAT indicates that it held (to reduce clutter, we omit TIMEOUT and FAIL results). The rightmost column reports the stability values of the various training methods. We define a family of models to be *unstable* in the case where a property holds in the family but ceases to hold in another model from the same family with a higher number of training iterations. Intuitively, the model "forgot" a desirable property as training progressed.

These 752 collision-prone models include *all* 260 DDQN-trained models, 256 Reinforce models, and 236 PPO models. Furthermore, when we conducted a model filtering process based on all six properties (three collision types and three types of infinite loops), we discovered that 778 models out of the total of 780 (over 99.7%!) violated at least one property. The PPO algorithm trained the only 2 models that passed our filtering process. Further analyzing the results, we observed that PPO models tended to be safer than those trained by other algorithms: they usually had the fewest violations per property. However, there are cases in which PPO proved less successful. For example, our results indicate that PPO-trained models are more prone to enter an ALTERNATING LOOP than those trained by Reinforce. Specifically, 214 (82.3%) of the PPO models have been found to enter this undesired state, compared to 145 (55.8%) of the Reinforce models. We also point out that, similar to the case with collision properties, *all* DDQN models violated this property. Finally, when considering 12-step cycles (either LEFT CYCLE or RIGHT CYCLE), 44.8% of the DDQN models entered such cycles, compared to 30.7% of the Reinforce models, and just 12.4% of the PPO models. In computing these results, we computed the fraction of violations (SAT queries) out of the number of queries that did not time out or fail, and aggregated SAT results for both cycle directions. Interestingly, we observed a bias toward violating a certain subcase of various properties in some cases. For example, in the case of entering complete cycles — although 125 (out of 520) queries on Reinforce models indicated that the agent might enter a cycle in either direction, in 96% of these violations the agent entered a RIGHT CYCLE. This bias is not present in models trained by the other algorithms, where the violations are roughly evenly divided between cycles in both directions.

We find that our results demonstrate that different "black-box" algorithms generalize very

differently with respect to various properties. In our setting, PPO produces the safest models, while DDQN tends to produce models with a higher number of violations. We note that this does not necessarily indicate that PPO-trained models perform better but are more robust to corner cases. Using our filtering mechanism, it is possible to select the safest models among the available, seemingly equivalent candidates. Next, we used verification to compute the bravery score of the various models. Using a binary search, we computed for each model the minimal distance a dead-ahead obstacle needs to have for the robot to move forward. The search range was $[0.18, 1]$ meters, and the optimal values were computed up to a 0.01 precision. Almost all binary searches terminated within minutes, and none of them hit the `TIMEOUT` threshold. By first filtering the models based on their safe behavior, then by their bravery scores, we can find the few models that are safe (do not collide), and not overly conservative. These models tend to take efficient paths and may come close to an obstacle, but without crashing with it. We also point out that over-conservativeness may significantly reduce the success rate in specific scenarios, such as cases where the obstacle is close to the target. Specifically, of the only two models that survived the first filtering stage, one is considerably more conservative than the other. The braver model may move forward when the distance is slightly over 0.42 meters, while the over-conservative model never moves forward in cases where a similar obstacle is closer than 0.88 meters.

| | *Scenario 1* | | | *Scenario 2* | | | *Scenario 3* | | |
|---|---|---|---|---|---|---|---|---|---|
| ALGO | `SAT` | `UNSAT` | `TIMEOUT` | `SAT` | `UNSAT` | `TIMEOUT` | `SAT` | `UNSAT` | `TIMEOUT` |
| Baseline | 60 | 0 | 0 | 51 | 0 | 9 | 60 | 0 | 0 |
| SBP | 22 | 38 | 0 | 0 | 41 | 19 | 9 | 34 | 17 |

Table 8.2: Results of the formal verification queries over a total of 120 trained DNNs, for each of the three properties in question. The first row shows the results of the 60 baseline policies and the second row shows the results of the 60 policies trained by our method, with all rules active.

**Combining the Approaches** As an additional means of proving the effectiveness of our methods, we ran formal verification on the models trained with our constrained scenario-based approach (SPB). To conduct a fair comparison, we selected only models that passed our success cutoff value (85%). For each of these models, we ran three verification queries — each checking whether the model violates a given property (`SAT`), or abides by it for all inputs (`UNSAT`). We note that a verifier might also fail to terminate, due to `TIMEOUT` or `MEMOUT` errors. Each query ran with a `TIMEOUT` value of 36 hours, and a `MEMOUT` value of 6 GB. Table 8.2 summarizes the results of our experiments. These results show a *significant* change of behavior between DNNs trained with the baseline algorithm and those trained by our method. Indeed, we see that the latter policies much more often completely abide by the specific rules, and are consequently far more reliable.

# Chapter 9

# CONCLUSION AND FUTURE DIRECTION

This thesis presented a set of methodologies that combines training and verification to generate safe, predictable, and trustworthy agents through a Safe Deep Reinforcement Learning process. In this conclusive chapter, we recap our contributions and discuss the possible future research directions.

**Constrained DRL** Deep Reinforcement learning approaches rely on a simple paradigm. An agent interacts with the environment through a trial-and-error process, making mistakes and learning from them to maximize a reward signal. These approaches have shown groundbreaking results in a large variety of tasks, such as robotics, game playing, and autonomous driving. However, one of the significant strengths of DRL, i.e., it requires only a reward signal to learn, can be, at the same time, a weakness. An agent typically explores the environment deeply to learn how to accomplish a task efficiently, in some cases generating policies that do not match human expectations. This training paradigm, combined with the exploitation of deep neural networks as artificial brains for the agents, often leads to the generation of *unpredictable policies*. This approach presents some significant drawbacks. First, unpredictability is usually considered undesirable and potentially unsafe in a safety-critical context (i.e., where human safety or expensive equipment can be involved). Second, an end-to-end unsupervised approach does not allow the injection of behavioral suggestions that are not limited to safety; but can involve additional requirements, such as human-friendly behaviors, management of optional subtasks, and more. Third, providing prior knowledge into the training loop is crucial to obtaining more reliable and efficient intelligent agents. At the same time, we believe that allowing the agent a proper degree of freedom to learn new and unforeseen strategies to accomplish a task is necessary and one of the main strengths of DRL that should not be neglected.

In Part I of the thesis we addressed these problems by improving the classical unconstrained optimization of DRL, whose objective is the pure maximization of the reward function, to a constrained optimization approach, where the maximization of the reward signal is subject to

adherence to some given constraints. Crucially, we are not interested in strictly matching all the requirements. In contrast, we aim to guarantee a tolerance to the number of violations to avoid substantial limitations to the exploration of new strategies. To reach this goal, in the first part of the thesis, we faced two problems: (i) how to encode the requirements; and (ii) how to perform constrained optimization in the DRL context. We proposed to encode the requirements in the form of "scenarios" through a formal language, Scenario-Based Programming (SBP). To provide an intuition, from a high-level perspective, a "scenario" is a sort of finite state machine that describes the agent's desired (or undesired) behavior. We then provided a method to translate these rules into a cost signal for the learning process. Keeping this value below a given threshold becomes the constraint for our optimization process. To close the loop, we propose to exploit the lagrangian relaxation of constrained optimization problems to provide an approach that concurrently maximizes the reward function while limiting the probabilities of violations to the given rules below a tolerance value.

**Formal Verification of DRL Systems** Another crucial challenge in safety-critical contexts is to formally guarantee the respect of the requirements before the deployment in a real-world context, outside a controlled environment. The black-box nature of the deep neural networks (DNN), the complex structure, and the non-linearity inherent in these tools make difficult the understanding and analysis of these systems. Providing formal guarantees about these autonomous systems' behavior has been considered only a theoretical problem and practically not feasible for years. However, the research in this area has recently significantly increased, providing DNN verification tools that scale well on state-of-the-art neural networks [Katz et al., 2019; Wang et al., 2021].

In Part II of the thesis, we showed how to apply and extend these approaches to certify DRL agents, focusing on robotic systems. In particular, we presented a methodology for formally verifying safety properties involving sequences of actions and interactions with the environment. Previous methods aim at verifying input-output relations, which is a strong limitation for DRL agents, where an agent sequentially interacts with the environment performing sequences of actions. Furthermore, we proposed a method to formally quantify the number of violations of these properties, generalizing the satisfiability neural network verification problem to a counting one. We finally presented a novel algorithm to compute a specific metric, the violation rate, to formally evaluate the safety of the system.

## FUTURE DIRECTIONS

Throughout the thesis, we presented a complete pipeline to inject prior knowledge, improve safety, and provide formal guarantees on the behavior of the systems to obtain reliable and trustworthy intelligent agents. We validated our approaches by applying our techniques to the complex robotics problem of mapless navigation, showing that our methodologies are general and

potentially extendable to other domains; we finally show that our techniques produce policies that respect a set of requirements without adversely affecting the main objective of the optimization. In conclusion, we believe that this thesis is a further step toward the widespread adoption of intelligent systems in real-world contexts and even for safety-critical tasks. Moreover, the proposed approaches pave the opportunity of extending the work in numerous ways.

**Iterative policy improvement via Constrained DRL** Throughout this thesis, we exploited constrained reinforcement learning to inject prior knowledge into the generated policies, specifically by designing handcrafting rules that suggest a set of desired behaviors in the training process. A natural extension is to inject expertise through *demonstrations*, to obtain a novel policy improvement approach via Constrained DRL. A possible approach is to exploit a specific metric that measures the *distance* between the demonstration and the agent's behavior and treats this value as a cost function to minimize for the constrained optimization, obtaining agents that perform similarly to the demonstrations. The crucial advantage of this approach is the parameter *threshold* that represents a tolerance, which allows the agent sometimes to ignore the demonstration if useful for the maximization of the reward. As an alternative approach, this *distance* metric can also be measured in discrete action spaces, for example, by counting the number of times the agent selects an action that differs from the demonstration in the same (or similar) conditions. Normalizing this value by the number of steps in an episode, we estimate the probability for the agent of choosing a different action. The threshold for the constrained optimization problem represents a bound for this frequency. Moreover, thinking of this approach in an iterative fashion, we obtain a safe incremental update of the policy. For example, suppose to have a safe human demonstration that we call $\pi^*$, which is suboptimal but safe by construction. Performing our approach with a strict tolerance threshold for changes, we obtain an improved policy $\pi^0$ with a strong bound on disruptive changes. Following the intuition of PPO [Schulman et al., 2017], it is unlikely to obtain catastrophic changes by limiting the differences between two consecutive policy updates. The new policy can be formally checked to guarantee its safety, and then the process can be repeated to obtain $\pi^1$. By iterating this process, we obtain an improved policy $\pi^t$ at each step, slightly different from the previous safe policy $\pi^{t-1}$, which has been formally analyzed. These solutions should be further analyzed from a theoretical and an empirical perspective. However, these processes highlight the versatility of constrained reinforcement learning, which can be extended to solve a large variety of problems. A final remark on this topic is that it is not limited to human demonstrations. For example, as a starting policy $\pi^*$, a suboptimal planner or even another trained DNN can be used for this approach.

**Managing the Risk Tolerance** Managing risk tolerance is a fundamental challenge for the safety-critical problems we addressed throughout the thesis. In our proposed lagrangian PPO, we exploited a threshold to modulate the frequency of violations that we are willing to accept. In

our approach, we estimate the adherence to our behavioral requirements based on the expected value of the corresponding cost functions. In the recent work from Yang et al. [2022b]), the authors highlight that considering a constraint respected when the expected value of the cost function is just below the desired threshold is, in fact, an optimistic practice. The intuition is that, by definition, the expected value represents the mean of a distribution, and thus, even if the mean of the distribution is below the given limit, the probability of violating it can still be high. Given that, a more conservative approach, such as estimating the conditional Value-at-Risk [Yang et al., 2021], should be preferred for safety-critical tasks (i.e., considering a parameterized quantile instead of the mean of the distribution). Following this intuition, a possible future direction is to exploit this concept in our algorithms to modulate the priority of the desired behavior. For example, one could rank the requirements and provide a more strict tolerance to the most dangerous actions and a low priority to the behavioral requirements.

**Alternative Languages for Defining the Safety Requirements** Another interesting challenge is improving the description language for the safety requirements (i.e., our rules). From our experiments and case studies, SBP has shown an excellent expressiveness level, and the generated rules have proven to be easy to write, read and understand; moreover, *scenario based programming* (SBP) can encode both positive and negative requirements (i.e., actions to perform and to avoid). However, our optimization approach aims to be general and agnostic to the rules' encoding. Investigating different formal languages to describe the requirements is an exciting direction to increase the potential application domains of our constrained optimization method. A fundamental requirement for the description language is that it must be sound with the policy gradient theorem by depending only on the current trajectory; the analysis of Chap. 4 shows that SBP respects this requirement, but adopting alternative languages requires further studies. A possible alternative language is *statecharts* [Harel, 1987], which can be more intuitive given its graphical nature, but also classical state-based automata or other Turing-complete description languages should be considered and analyzed.

**Formal Verification of Multi-Agent Systems** Future directions also involve the formal verification part of the thesis. For example, the proposed method for verifying our time-dependent properties (i.e., multi-step verification) could be extended for the formal verification of multi-agent systems. The approach presented in Chap. 7 consists in multiplying the neural network $n$ times, where $n$ is the number of steps of the properties we aim to verify. The network copies are joined together, where each output at step $t$ directly influences the input at step $t+1$ to build a bigger and interconnected network. This DNN, built as a chain of smaller networks, can then be verified with a standard verification tool (e.g., Marabou [Katz et al., 2019]). Following this intuition, a possible future direction is to encode the neural networks that control different agents in a similar way, where the output of an agent directly affects the input of the others (and vice versa). Exploiting this configuration, instead of having a copy of the DNN for

each time step in a sequence, we obtain a bigger network built with the neural controllers of each agent; relying on this structure, it is possible to verify properties that involve the relations between a set of agents. For example, suppose two agents are moving straight in parallel at a distance $m$; an agent is allowed to turn toward the center but, to avoid a collision, they can not perform this action together. This approach could be a key asset to devising reliable decentralized multi-agent reinforcement learning approaches.

**Approximate Verification** Even for state-of-the-art formal verification methods, scalability is a crucial challenge. An exciting possible future direction is to exploit the formal verification results to drive the training loop. However, in this scenario, a verification tool should be called multiple times, introducing overhead and making the training process unfeasible. A possible solution to investigate is the introduction of an approximated verification method to compute a specific metric, such as the violation rate introduced in Chap. 6. The idea is to obtain an estimation of the actual number of violations that can drive the training toward safer regions; this process is not intended as a replacement for the formal verification but, instead, as a helpful tool to increase the probability of obtaining safe policies before the model selection phase. A trivial approximation approach is sampling. Of course, the precision of this solution is strongly related to the number of samples and the size of the input space. Our intuition is that the input space can be reduced, through an iterative approach, until it becomes easily manageable by a formal method. In this setup, providing some formal bounds to the confidence interval is a crucial challenge.

# LIST OF FIGURES

# LIST OF TABLES

# LISTINGS

# Acknowledgements

Firstly, I want to say thanks to my advisor, Prof. Alessandro Farinelli, whose support, encouragement, and belief in my abilities have allowed me to pursue my academic dreams. Without his guidance and mentorship, I would not be where I am today. I would like to extend my sincere thanks to my co-author and friend Luca Marzari. I might have abandoned one of our most important works without his unyielding belief that anything is possible. I owe a debt of gratitude to Prof. Guy Katz, who hosted me at The Hebrew University of Jerusalem during one of the most productive periods of my life. His guidance and mentorship completely changed my vision of research and academia, motivating me to strive harder to achieve my goals. I am deeply grateful to the amazing people from his lab, including Raz, Idan, Omri, and Raya, for showing me how a strong team can work together to achieve extraordinary results. Among them, a special mention is for Guy Amir, I'm honored to say that he is today more than a colleague and a co-author, he is especially a friend.

On a personal level, I must mention Enrico, Matteo, Andrea, and Tommaso; without your friendship, I could never have finished this journey while preserving my mental health. I also want to express my heartfelt appreciation to my girlfriend, Marcella. I can not express how I'm grateful to have your unconditioned support in every single moment of my life, you are the only person in this world with whom I can be completely myself. As an only child, I cannot know the bond of having a sister, but I imagine it would be similar to the connection I feel towards my wonderful cousins, Silvia and Ilaria. Thank you for your time, encouragement, and the brilliant conversations and discussions we always share.

The first special thank goes to Enrico Marchesini. I can not believe that I met you nine years ago on the first day of our Bachelor's. Since then, you have played a multitude of roles in my life: a fellow student in our course, my study partner for every exam, a fierce competitor during our Master's program, a friend, my best friend, an advisor during the early stages of my PhD, a co-author, a rival again as a researcher, and today a model to follow. But now I want to thank you for the most important thing, being a sincere friend whom I can always trust. I also want to extend my gratitude to my best friend, Tommaso Saturnia, we're used to saying that between us we don't need to say anything. I'll make an exception. You're one the smartest person I've ever met and our discussions on every possible topic constitute a fundamental part of my personal growth; you have shown me that there is much more than science, if today I love art, books, and movies is largely because of you. Your incredible success as a musician and composer has always inspired me to push my limits and pursue my dreams, and I am grateful for your constant motivation.

Finally, I apologize for the language, but I would like to add in Italian a last special thanks to my parents: *Grazie mamma e grazie papà. Ogni singola goccia di inchiostro che si trova su questa tesi è merito vostro. Mi avete dato l'opportunità di studiare, sempre lasciandomi i miei tempi e i miei spazi, dandomi tutto il supporto possible e senza mai chiedere nulla in cambio. Nemmeno nelle mie migliori fantasie potrei immaginare dei genitori migliori di voi. Grazie.*

# BIBLIOGRAPHY


Achiam, J. (2018). Openai spinning up. *GitHub, GitHub repository*.

Achiam, J., Held, D., Tamar, A., and Abbeel, P. (2017). Constrained policy optimization. In *International conference on machine learning*.

Alexandron, G., Armoni, M., Gordon, M., and Harel, D. (2014). Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking. In *Proc 36th Int. Conf. on Software Engineering (ICSE)*.

Altman, E. (1998). Constrained markov decision processes with total cost criteria: Lagrangian approach and dual linear program. *Mathematical methods of operations research*.

Amir, G., Corsi, D., Yerushalmi, R., Marzari, L., Harel, D., Farinelli, A., and Katz, G. (2022). Verifying learning-based robotic navigation systems. *arXiv preprint arXiv:2205.13536*.

Amir, G., Schapira, M., and Katz, G. (2021). Towards scalable verification of deep reinforcement learning. In *2021 formal methods in computer aided design (FMCAD)*.

Amsters, R. and Slaets, P. (2019). Turtlebot 3 as a Robotics Education Platform. In *Proc. 10th Int. Conf. on Robotics in Education (RiE)*, pages 170–181.

Attanasio, A., Scaglioni, B., Leonetti, M., Frangi, A., Cross, W., Biyani, C., and Valdastri, P. (2020). Autonomous Tissue Retraction in Robotic Assisted Minimally Invasive Surgery - A Feasibility Study. *IEEE Robotics and Automation Letters*.

Baier, C. and Katoen, J. (2008). *Principles of model checking*. MIT press.

Banerjee, C., Mukherjee, T., and Pasiliao Jr, E. (2019). An empirical study on generalizations of the relu activation function. In *Proceedings of the 2019 ACM Southeast Conference*.

Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., and Criminisi, A. (2016). Measuring neural net robustness with constraints. In *Conference on Neural Information Processing Systems*.

Berry, M., Mohamed, A., and Yap, B. (2019). *Supervised and unsupervised learning for data science*. Springer.

Bertsekas, D. (2014). *Constrained optimization and Lagrange multiplier methods*. Academic press.

Binns, R. (2018). Fairness in machine learning: Lessons from political philosophy. In *Conference on Fairness, Accountability and Transparency*.

Birjali, M., Kasri, M., and Beni-Hssane, A. (2021). A comprehensive survey on sentiment analysis: Approaches, challenges and trends. *Knowledge-Based Systems*.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. (2016). End to End Learning for Self-Driving Cars. Technical Report. http://arxiv.org/abs/1604.07316.

Briot, J. and Pachet, F. (2020). Deep learning for music generation: challenges and directions. *Neural Computing and Applications*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Casadio, M., Komendantskaya, E., Daggitt, M., Kokke, W., Katz, G., Amir, G., and Refaeli, I. (2022). Neural network robustness as a verification property: A principled case study. In *International Conference on Computer Aided Verification*.

Clarke, E., Henzinger, T. A., Veith, H., and Bloem, R. (2018). *Handbook of model checking*. Springer.

Corsi, D. (2022). Basicrl. https://github.com/d-corsi/BasicRL.

Corsi, D., Marchesini, E., and Farinelli, A. (2021). Formal Verification of Neural Networks for Safety-Critical Tasks in Deep Reinforcement Learning. In *37th Conf. on Uncertainty in Artificial Intelligence (UAI)*.

Corsi, D., Yerushalmi, R., Amir, G., Farinelli, A., Harel, D., and Katz, G. (2022). Constrained reinforcement learning for robotics via scenario-based programming. *arXiv preprint arXiv:2206.09603*.

Cédric, C., Olivier, S., and Pierre-Yves, O. (2019). A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms.

Damm, W. and Harel, D. (2001). Lscs: Breathing life into message sequence charts. *Formal methods in system design*.

Deng, L. and Liu, Y. (2018). *Deep Learning in Natural Language Processing*. Springer.

Dutta, D., Jha, S., Sanakaranarayanan, S., and Tiwari, A. (2017). Output range analysis for deep neural networks. *arXiv preprint arXiv:1709.09130*.

Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*.

Eliyahu, T., Kazak, Y., Katz, G., and Schapira, M. (2021). Verifying learning-augmented systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.

EU (2022). A european approach to artificial intelligence.

Fogel, D. (1995). Toward a new philosophy of machine intelligence. *IEEE Evolutionary Computation*.

Fogel, D. (2006). *Evolutionary computation: toward a new philosophy of machine intelligence*. John Wiley & Sons.

Garcıa, J. and Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*.

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., and Vechev, M. (2018). AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*.

Goodfellow, I., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. (2013). An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*.

Goodfellow, I., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*.

Gordon, M., Marron, A., and Meerbaum-Salant, O. (2012). Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th ACM Annual Conf. on Innovation and Technology in Computer Science Education (ITCSE)*.

Gronauer, S. (2021). Bullet safety gym. https://github.com/SvenGronauer/Bullet-Safety-Gym.

Gronauer, S. (2022). Bullet-safety-gym: A framework for constrained reinforcement learning. Technical report, mediaTUM.

Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2017). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018a). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*.

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

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2012a). Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*.

Harel, D. and Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming using LSCs and the Play-Engine*. Springer Science & Business Media.

Harel, D., Marron, A., and Weiss, G. (2012b). Behavioral programming. *Communications of the ACM*.

He, T., Zhao, W., and Liu, C. (2023). Autocost: Evolving intrinsic cost for zero-violation reinforcement learning. *arXiv preprint arXiv:2301.10339*.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*.

Hu, H., Jia, X., He, Q., Fu, S., and Liu, K. (2020). Deep reinforcement learning based agvs real-time scheduling with mixed rule for flexible shop floor in industry 4.0. *Computers & Industrial Engineering*.

Jiang, P., Ergu, D., Liu, F., Cai, Y., and Ma, B. (2022). A review of yolo algorithm developments. *Procedia Computer Science*.

Juliani, A., Berges, V., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A General Platform for Intelligent Agents. Technical Report. https://arxiv.org/abs/1809.02627.

Kamran, D., Simão, T., Yang, Q., Ponnambalam, C., Fischer, J., Spaan, M., and Lauer, M. (2022). A modern perspective on safe automated driving for different traffic dynamics using

constrained reinforcement learning. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*.

Katz, G., Barrett, C., Dill, D., Julian, K., and Kochenderfer, M. (2017). Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*.

Katz, G., Huang, D., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al. (2019). The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*.

Kazak, Y., Barrett, C., Katz, G., and Schapira, M. (2019). Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*.

Khadka, S. and Tumer, K. (2018). Evolution-guided policy gradient in reinforcement learning. *Advances in Neural Information Processing Systems*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lai, S., Xu, L., Liu, K., and Zhao, J. (2015). Recurrent Convolutional Neural Networks for Text Classification. In *Proc. 29th AAAI Conf. on Artificial Intelligence*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*.

Lehman, J., Chen, J., Clune, J., and Stanley, K. (2018). Safe mutations for deep and recurrent neural networks through output gradients. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

Liu, C., Arnon, T., Lazarus, C., Barrett, C., and Kochenderfer, M. (2019). Algorithms for verifying deep neural networks. In *Foundations and Trends in Optimization*.

Liu, Y., Ding, J., and Liu, X. (2020). Ipo: Interior-point policy optimization under constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Lomuscio, A. and Maganti, L. (2017). An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks. Technical Report. http://arxiv.org/abs/1706.07351.

Lyu, Z., Ko, C. Y., Kong, Z., Wong, N., Lin, D., and Daniel, L. (2020). Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044.

Marchesini, E. (2022). *Enhancing Exploration and Safety in Deep Reinforcement Learning*. University of Verona.

Marchesini, E., Corsi, D., Benfatti, A., Farinelli, A., and Fiorini, P. (2019). Double deep q-network for trajectory generation of a commercial 7dof redundant manipulator. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*.

Marchesini, E., Corsi, D., and Farinelli, A. (2020). Genetic soft updates for policy evolution in deep reinforcement learning. In *International Conference on Learning Representations*.

Marchesini, E., Corsi, D., and Farinelli, A. (2021a). Benchmarking Safe Deep Reinforcement Learning in Aquatic Navigation. In *Proc. IEEE/RSJ Int. Conf on Intelligent Robots and Systems (IROS)*.

Marchesini, E., Corsi, D., and Farinelli, A. (2021b). Exploring Safer Behaviors for Deep Reinforcement Learning. In *Proc. 35th AAAI Conf. on Artificial Intelligence (AAAI)*.

Marchesini, E. and Farinelli, A. (2020). Discrete Deep Reinforcement Learning for Mapless Navigation. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*.

Marron, A., Hacohen, Y., Harel, D., Mülder, A., and Terfloth, A. (2018). Embedding scenario-based modeling in statecharts. In *MODELS workshops*.

Martín, H. and Lope, J. (2009). Learning autonomous helicopter flight with evolutionary reinforcement learning. In *International Conference on Computer Aided Systems Theory*.

Marzari, L., Corsi, D., Marchesini, E., and Farinelli, A. (2022). Curriculum learning for safe mapless navigation. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*.

Meng, T. and Khushi, M. (2019). Reinforcement learning in financial markets. *Data*.

Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. Technical Report. https://arxiv.org/abs/1312.5602.

Moerland, T., Broekens, J., and Jonker, C. (2020). Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*.

Moore, E. (1963). Interval arithmetic and automatic error analysis in digital computing. In *Stanford University*.

Moosavi-Dezfooli, S., Fawzi, A., Fawzi, O., and Frossard, P. (2017). Universal Adversarial Perturbations. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1765–1773.

Nandkumar, C., Shukla, P., and Varma, V. (2021). Simulation of Indoor Localization and Navigation of Turtlebot 3 using Real Time Object Detection. In *Proc. Int. Conf. on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*.

Nelder, J. and Mead, R. (1965). A simplex method for function minimization. *The computer journal*.

OpenAI (2021). Gym. https://www.gymlibrary.dev/.

Owen, M., Panken, A., Moss, R., Alvarez, L., and Leeper, C. (2019). Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*.

Pan, X., You, Y., Wang, Z., and Lu, C. (2017). Virtual to real reinforcement learning for autonomous driving. *arXiv preprint arXiv:1704.03952*.

Pfeiffer, M., Shukla, S., Turchetta, M., Cadena, C., Krause, A., Siegwart, R., and Nieto, J. (2018). Reinforced Imitation: Sample Efficient Deep Reinforcement Learning for Mapless Navigation by Leveraging Prior Demonstrations. *IEEE Robotics and Automation Letters*.

Pore, A., Corsi, D., Marchesini, E., Dall'Alba, D., Casals, A., Farinelli, A., and Fiorini, P. (2021). Safe reinforcement learning using formal verification for tissue retraction in autonomous robotic-assisted surgery. In *International Conference on Intelligent Robots and Systems (IROS)*.

Pore, A., Finocchiaro, M., Dall'Alba, D., Hernansanz, A., Ciuti, G., Arezzo, A., Menciassi, A., Casals, A., and Fiorini, p. (2022). Colonoscopy Navigation using End-to-End Deep Visuomotor Control: A User Study. *arXiv preprint arXiv:2206.15086*.

Pulina, L. and Tacchella, A. (2010). An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., et al. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*.

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

Roy, J., Girgis, R., Romoff, J., Bacon, P., and Pal, C. (2021). Direct Behavior Specification via Constrained Reinforcement Learning. Technical Report. https://arxiv.org/abs/2112.12228.

Ruan, X., Ren, D., Zhu, X., and Huang, J. (2019). Mobile robot navigation based on deep reinforcement learning. In *2019 Chinese control and decision conference (CCDC)*.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. Technical Report. http://arxiv.org/abs/1707.06347.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D., Singh, S., Precup, D., and Sutton, R. (2021). Reward is enough. *Artificial Intelligence*.

Simão, T., Jansen, N., and Spaan, M. (2021). Alwayssafe: Reinforcement learning without safety constraint violations during training. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*.

Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. Technical Report. http://arxiv.org/abs/1409.1556.

Slawinski, P., Taddese, A., Musto, k., Sarker, S., Valdastri, P., and Obstein, K. (2018). Autonomously controlled magnetic flexible endoscope for colon exploration. *Gastroenterology*.

Srinivasan, K., Eysenbach, B., Ha, S., Tan, J., and Finn, C. (2020). Learning to be safe: Deep rl with a safety critic. *arXiv preprint arXiv:2010.14603*.

Stooke, A., Achiam, J., and Abbeel, P. (2020). Responsive safety in reinforcement learning by pid lagrangian methods. In *International Conference on Machine Learning*.

Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*. MIT press.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2013). Intriguing Properties of Neural Networks. Technical Report. http://arxiv.org/abs/1312.6199.

Tai, L., Li, S., and Liu, M. (2016). A deep-network solution towards model-less obstacle avoidance. In *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*.

Tai, L., Paolo, G., and Liu, M. (2017). Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Tamar, A., Di Castro, D., and Mannor, S. (2016). Learning the variance of the reward-to-go. *The Journal of Machine Learning Research*.

Temeltas, H. and Kayak, D. (2008). Slam for robot navigation. *IEEE Aerospace and Electronic Systems Magazine*.

Tessler, C., Mankowitz, D. J., and Mannor, S. (2019). Reward constrained policy optimization. In *7th International Conference on Learning Representations, ICLR, 2019*.

Thananjeyan, B., Balakrishna, A., Nair, S., Luo, M., Srinivasan, K., Hwang, M., Gonzalez, J., Ibarz, J., Finn, C., and Goldberg, K. (2021). Recovery rl: Safe reinforcement learning with learned recovery zones. *IEEE Robotics and Automation Letters*.

Tjeng, V., Xiao, K., and Tedrake, R. (2018). Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. In *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI)*.

Wahid, A., Toshev, A., Fiser, M., and Lee, T. (2019). Long range neural navigation policies for the real world. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018a). Efficient formal safety analysis of neural networks. In *Conference on Neural Information Processing Systems*.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018b). Formal security analysis of neural networks using symbolic intervals. In *USENIX Security Symposium*.

Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C., and Kolter, J. (2021). Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *Advances in Neural Information Processing Systems*.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*.

Weng, L., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D., and Dhillon, I. (2018). Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*.

Yang, Q., Simão, T., Jansen, N., Tindemans, S., and Spaan, M. (2022a). Training and transferring safe policies in reinforcement learning. In *AAMAS 2022 Workshop on Adaptive Learning Agents*.

Yang, Q., Simão, T., Tindemans, S., and Spaan, M. (2021). Wcsac: Worst-case soft actor critic for safety-constrained reinforcement learning. In *AAAI*.

Yang, Q., Simão, T., Tindemans, S., and Spaan, M. (2022b). Safety-constrained reinforcement learning with a distributional safety critic. *Machine Learning*.

Yang, T., Rosca, J., Narasimhan, K., and Ramadge, P. (2020). Projection-based constrained policy optimization. *arXiv preprint arXiv:2010.03152*.

Yang, Y. (2023). Safety gymnasium. https://github.com/OmniSafeAI/safety-gymnasium.

Yerushalmi, R., Amir, G., Elyasaf, A., Harel, D., Katz, G., and Marron, A. (2022). Scenario-assisted deep reinforcement learning. In *Proc. 10th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*.

Zamora, I., Lopez, N., Vilches, V., and Cordero, A. (2016). Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint arXiv:1608.05742*.

Zhang, H., Weng, T., Chen, P., Hsieh, C., and Daniel, L. (2018). Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*.

Zhang, J., Springenberg, J., Boedecker, J., and Burgard, W. (2017). Deep reinforcement learning with successor features for navigation across similar environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Zhao, W., Queralta, J., and Westerlund, T. (2020). Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*.

Zhu, W., Xie, L., Han, J., and Guo, X. (2020). The application of deep learning in cancer prognosis prediction. *Cancers*.

Zhu, Y., Mottaghi, R., Kolve, E., Lim, J., Gupta, A., Fei-Fei, L., and Farhadi, A. (2017). Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*.