






# Enforcing Specific Behaviours via Constrained DRL and Scenario-Based Programming

Davide Corsi<sup>1,\*</sup> , Raz Yerushalmi<sup>2,3,\*</sup> , Guy Amir<sup>3</sup> ,  
Alessandro Farinelli<sup>4</sup> , David Harel<sup>3</sup> , and Guy Katz<sup>2</sup> 

<sup>1</sup> University of California: Irvine, USA  
dcorsi@uci.edu

<sup>2</sup> The Hebrew University of Jerusalem, Israel  
{guyam, guykatz}@cs.huji.ac.il

<sup>3</sup> The Weizmann Institute of Science, Israel  
{raz.yerushalmi, david.harel}@weizmann.ac.il

<sup>4</sup> University of Verona, Italy  
alessandro.farinelli@univr.it

**Abstract.** Deep Reinforcement Learning (DRL) has achieved ground-breaking results in robotics, cyber-physical systems, healthcare, and many other real-world applications in recent years. However, despite their success, DRL controllers’ inherent opacity and unpredictability limit their widespread adoption in many safety-critical scenarios. In such contexts it is crucial to consider additional safety and behavioral requirements pertaining to the deployed agents, in addition to its performance. In this paper, we propose using Scenario-Based Programming (SBP) to define a cost signal that can be optimized together with the standard reward function to enforce additional behaviors in the final agents. To this end, we rely on the constrained DRL framework, particularly on a modified version of Lagrangian-PPO, which we call  $\lambda$ -PPO, designed especially for the multi-step and temporal nature of the SBP requirements. This approach allows us to easily design and enforce the agent’s adherence to these requirements during training without compromising its freedom to explore the state space and converge to an optimal policy, enabling the use of a simple reward function. We have validated our method extensively by experimenting with real robotic platforms in a mapless navigation task, demonstrating the success of our approach. We use SBP to define different types of requirements, including a more predictable behavior, safety properties, and the injection of prior knowledge to drive training.

## 1 Introduction

In recent years, *deep neural networks* (DNNs) have achieved state-of-the-art results in a large variety of tasks, including image recognition [11], game playing [33], protein folding [21], and more. In particular, *deep reinforcement learning*

---

\* Both authors contributed equally.

(DRL) [46] has emerged as a popular paradigm for training DNNs that perform complex tasks through continuous interaction with their environment. Indeed, DRL models have proven remarkably useful in robotic control tasks, such as navigation [25] and manipulation [36, 8], where they often outperform classical algorithms [57]. The success of DRL-based systems has naturally led to their integration as control policies in safety-critical tasks, such as autonomous driving [42], surgical assistance [38], flight control [24], and more. Consequently, 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 safety properties and additional behavioral requirements.

An emerging family of approaches for achieving this goal, known as *Constrained DRL* (CDRL) [40], attempts to simultaneously optimize two functions: the *reward*, which encodes the main objective of the task, and the *cost*, which represents the constraints. Several approaches attempt to tackle this problem with different strategies, ranging from convex optimization [1] to carefully designed penalty functions [27, 54] and model selection [31]. In this paper, we employ the Lagrangian dual relaxation of the constrained optimization problem, following a strategy similar to that used for lag-PPO [45] and RCPO [48]. However, despite their success in many applications, even state-of-the-art methods generally suffer from significant setbacks: (i) it is unclear how to translate these constraints into an effective signal for the training algorithm; (ii) there is no uniform human-readable way of defining the required constraints; and (iii) there is no clear understanding of the meaning of “*cost*” and how to balance between these costs and the performance.

In this paper, we present a novel methodology for addressing these challenges, by enabling domain experts to use formal language to define constraints relating to the agent’s behavior. We developed an approach that pushes the optimization process towards an optimal solution for its primary task, while also adhering to the different constraints. To achieve this goal, we extend and integrate two approaches: our  $\lambda$ -PPO algorithm for the actual training and the *Scenario-Based Programming* (SBP) [10, 17] framework for encoding user-defined constraints. Scenario-based programming is a software engineering paradigm that allows engineers to create a complex system that is 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 eventually combined to run simultaneously and produce cohesive system behavior.

In this work, we demonstrate how such scenarios can 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. Our methodology, however, poses many additional research questions, that we discuss in Sec.4, such as the meaning of the cost function in this context and how to deal with the temporal nature of the SBP requirements. In order to demonstrate the effectiveness of our approach on a

real-world task, we apply it to train a policy for performing robotic *mapless navigation* [53, 47] on the popular Robotis Turtlebot3 platform. Although common DRL-training techniques were shown to give rise to high-performance policies for this task [29], these policies are often unsafe, inefficient, or unpredictable, thus dramatically limiting their potential deployment in real-world systems [30, 31]. In contrast, and as demonstrated in Sec.5, our approach is able to generate trustworthy policies that are both safe and performant.

## 2 Preliminaries

Deep reinforcement learning (DRL) [26] is a popular paradigm for training deep neural networks [13]. In DRL, 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. Typically, a DRL problem is modeled as a Markov Decision Process (MDP), described by a tuple  $\langle S, A, T, r \rangle$ , where  $S$  is the state space,  $A$  is the actions space,  $T : S \times A \rightarrow S$  is the transition function that encodes the probability  $T(s_{t+1}|s_t, a_t)$  of transitioning from the state  $s_t$  to the next state  $s_{t+1}$  (given an action  $a_t \in A$  at timestep  $t$ ), and  $r : S \times A \rightarrow \mathbb{R}$  is the reward signal. The objective of a standard DRL algorithm is to maximize the *expected reward*, by finding a *policy*, denoted as  $\pi_\theta$ , which maps an observed *environment state*  $s$  to an *action*  $a$ .

In safety-critical tasks, the concept of optimality often goes beyond the maximization of a reward and can involve safety constraints that the agent should adhere to. A *constrained markov decision process* (CMDP) is an alternative framework for sequential decision-making that includes these additional requirements. CMDP extends the standard MDP with an additional signal: the *cost function*, defined as  $C : S \times \mathcal{A} \rightarrow \mathbb{R}$ , whose expected values must remain below a given threshold  $d \in \mathbb{R}$ . CMDP can support multiple cost functions (and consequently their thresholds), denoted by  $\{C_k\}$  and  $\{d_k\}$ , respectively. The set of *valid* policies for a CMDP is defined as:

$$\Pi_C := \{\pi_\theta \in \Pi : \forall k, J_{C_k}(\pi_\theta) \leq d_k\} \quad (2.1)$$

where  $J_{C_k}(\pi_\theta)$  is the expected sum of the  $k^{th}$  cost function over the trajectory and  $d_k$  is the corresponding threshold. Intuitively, the objective is to find a policy function that respects the constraints (i.e., is *valid*) and which also maximizes the expected reward (i.e., is *optimal*).

A formal approach to encode constraints in a classical optimization problem is by using *Lagrange multipliers* and relaxing the constrained problem into the corresponding dual unconstrained version [27, 1]. The optimization problem can then be encoded as follows:

$$J(\theta) = \min_{\pi_\theta} \max_{\lambda \geq 0} \mathcal{L}(\pi_\theta, \lambda) = \min_{\pi_\theta} \max_{\lambda \geq 0} J_R(\pi_\theta) - \sum_K \lambda_k (J_{C_k}(\pi_\theta) - d_k) \quad (2.2)$$

Eq. 2.2 encodes together the reward signal and the constraint monitors for the optimizer.

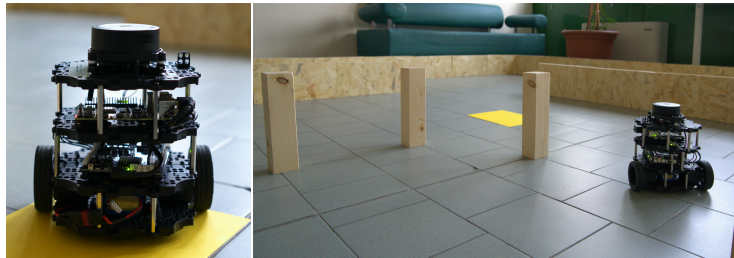
## 2.1 Scenario-Based Programming

Scenario-based programming (SBP) [10, 16] 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. 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. 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 request or wait 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 [16, 17]. For a formal definition of SBP, we refer to the work of Harel et al. [17].

Finally, in Appendix A we show a concrete example of a Scenario Based Program for the control of the temperature and water level in a water tank, inspired by the work of Harel et al. [18].

## 3 Case Study: Mapless Navigation

As a running example, we explain and demonstrate our proposed technique on 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 can rely only on local observations — e.g., from lidar sensors or cameras, and can not access a map of the surrounding environment. 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 difficult to solve. Specifically, the local and partially observable nature



**Fig. 1.** The Robotis Turtlebot3 platform.



of the problem renders learning a successful policy extremely challenging and hard to solve using classical algorithms [37]. Prior work has shown DRL to be among the most successful approaches for tackling this task, often outperforming hand-crafted algorithms [29]. As a platform for our study, we rely on the *Robotis Turtlebot 3* platform (Turtlebot, for short; see Fig. 1), which is widely used in the community [34, 5].

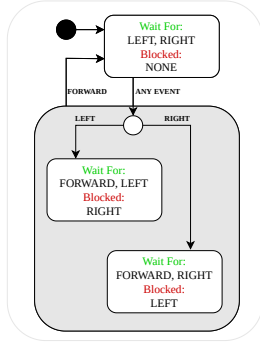
### 3.1 The Primary Reward Function

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 [20], which is compatible with the *Robotic Operating System* (ROS) [39] and allows a fast transfer to the actual platform (*sim-to-real* [56]). We designed a hybrid reward function, which includes a discrete component for the terminal states (“collision” or “target reached”), and a continuous component for the non-terminal states. Formally:

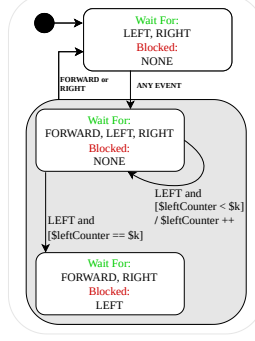
$$R_t = \begin{cases} \pm 1 & \text{terminal states} \\ (dist_{t-1} - dist_t) \cdot \eta - \beta & \text{otherwise} \end{cases} \quad (3.1)$$

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, we used an architecture that was shown to be successful in similar settings [29, 4]: (i) an input layer of nine neurons, including seven for the lidar scans and two for the polar coordinates of the target; (ii) two fully connected hidden layers of 32 neurons each; and (iii) an output layer of three neurons for the discrete actions (i.e., move **FORWARD**, turn **LEFT**, and turn **RIGHT**).

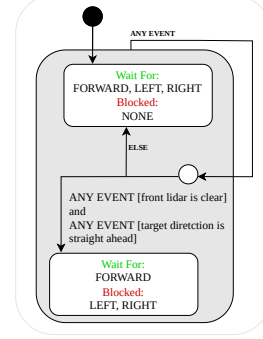
Using this reward function and network topology we were able to train agents that achieved high performance — i.e., obtained a success rate of approximately 95%, where “success” means that the robot reached its target without colliding into walls or obstacles. Analyzing the trained agents further, we observed that even DRL agents that achieved a high success rate might still demonstrate undesirable behavior in different scenarios. One such behavior is a sequence of back-and-forth turns, which 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 and demonstrate how we can remove these undesirable behaviors without altering the defined reward function (Eq. 3.1).



**Fig. 2.** A visualization of the first scenario: *avoid back-and-forth rotation*



**Fig. 3.** A visualization related to the second scenario: *avoid left turns larger than 180°*



**Fig. 4.** A visualization of the third scenario: *avoid turning when clear*

### 3.2 A Rule-Based Approach

To suppress undesired behaviors without changing the primary reward function (and potentially compromising the learning process), we propose integrating a scenario-based program into the DRL training process. More concretely, we propose to create specific scenarios to rule out each of the undesirable behaviors mentioned above. This is achieved by creating a mapping between each possible action  $a_t \in [\text{Move FORWARD, Turn LEFT, Turn RIGHT}]$  of the DRL agent and a dedicated event  $e_{a_t} \in [\text{SBP\_MoveForward, SBP\_TurnLeft, SBP\_TurnRight}]$  within the scenario-based program. These events allow the various scenarios to keep track of and react to the agent's actions. Similarly to the work of Yerushalmi et al. [52], 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 (*super-step*), until it returns to a steady state and then waits for another external event.

Considering again our running example (i.e., the Turtlebot mapless navigation case study), we create scenarios for discouraging the three undesirable behaviors we had previously observed. Scenario *avoid back-and-forth rotation* (Fig. 2) seeks to prevent in-place, back-and-forth turns by the robot. Scenario *avoid turns larger than 180°* (Fig. 3) seeks to prevent left turns in angles that are greater than 180° (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  $\text{LeftCounter}$  are variables. Each turn rotates the robot by 30°, and so we set  $k = 7$ . Scenario *avoid turning when clear* (Fig. 4) seeks to force the agent to move toward 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.

## 4 $\lambda$ -PPO: From Scenarios to Constrained DRL

In order to integrate the requirements expressed via SBP in a CDRL training process, we define the cost function to correspond to violations of scenario constraints. Intuitively, whenever the agent selects an action that is mapped to a *blocked* SBP event (during training) we should increase the *cost*. However, obtaining a differentiable function for the training process, that is also trajectory dependent, is not straightforward. To this end, we propose the following binary (indicator) function:

$$c_k(s_t, a, s_{t+1}) = I(\text{the tuple } \langle s_t, a, s_{t+1} \rangle \text{ is blocked by the } k^{th} \text{ rule}) \quad (4.1)$$

Intuitively, summing the values of  $c_k$  over a training episode yields the number of violations to the  $k^{th}$  scenario rule during a single trajectory; this value, if normalized over the number of steps, can be seen as a probability of having a violation during an episode. Crucially, the threshold can be interpreted as a bound for the probability of violating a requirement.

**Interpretation of the cost function.** Before introducing the main optimizations that constitute  $\lambda$ -PPO, it is important to clarify how we interpret the cost function for the evaluation. Following the insights presented in the large-scale analysis from Corsi et al. [9], we believe that generating a policy that achieves a true “*zero cost*” is almost impossible. For this reason, in this paper, we treat the cost function as a budget function and our goal is to minimize the expected probability of encountering unwanted situations above the given required threshold, rather than completely avoiding these behavior.

### 4.1 Optimized Lagrangian-PPO

In Section 2 we proposed to relax the Lagrangian constrained optimization problem into an unconstrained, *min-max* version thereof. Taking the gradient of Equation 2.2, and some algebraic manipulation, we derive the following two simultaneous problems:

$$\nabla_{\theta} \mathcal{L}(\pi, \lambda) = \nabla_{\theta} (J_R(\pi) - \sum_K \lambda_k J_{C_k}(\pi)) \quad (4.2)$$

$$\forall k, \quad \nabla_{\lambda_k} \mathcal{L}(\pi, \lambda) = -(J_{C_k}(\pi) - d_k) \quad (4.3)$$

However, the naïve application of this approach has shown strong instability and the proclivity to optimize only the cost in our experiments, limiting the exploration and resulting in poorly performing agents. To overcome these problems, we introduce two key optimizations that proved crucial in obtaining the results we present in the next section, in addition to a set of improvements to stabilize the training process.

**Reward Multiplier.** the standard update rule for the policy in a Lagrangian method is given in Equation 4.2. 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:

$$\nabla_{\theta} \mathcal{L}(\pi, \lambda) = \nabla_{\theta} (\alpha \cdot J_R(\pi) - \sum_K \lambda_k J_{C_k}(\pi)) \quad (4.4)$$

**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. 4.2), potentially not pushing the policy towards high-performance regions. 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:

$$\forall k, \lambda_k = \frac{\tilde{\lambda}_k}{2(\sum_K \tilde{\lambda}_k)} \quad \alpha = 1 - \sum_K \lambda_k \quad (4.5)$$

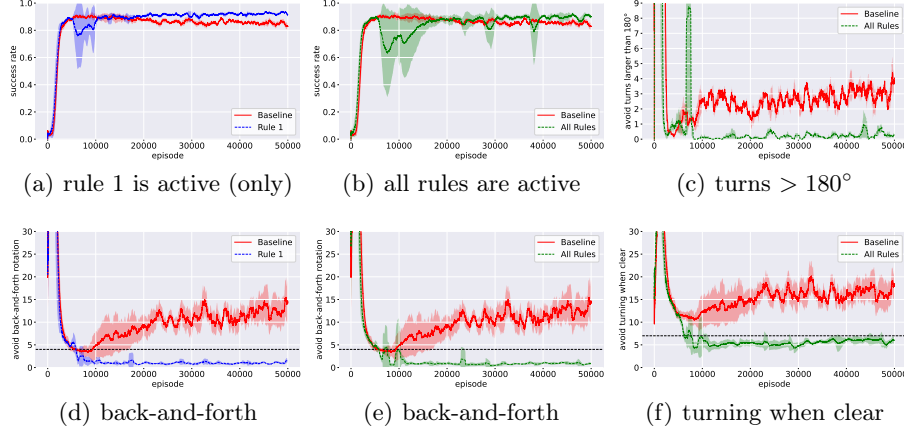
## 4.2 State Space Expansion and Convergence

In the previous sections, we explained the benefits of using SBP for encoding trajectory-based properties to effectively characterize intricate behaviors and their combinations. To achieve this objective, it is crucial to combine actions, states, and connections between them in a time-dependent manner. For example, if we need to encode the behavior of never turning left four times in a row, we can use a scenario-based property. However, optimizing the cost function generated by this scenario without caution may result in violating the properties of the Markov Decision Process (MDP), where transitions between states should only depend on the previous state and action. We leave it for future work to address that tension, in order to allow the use of arbitrary SBP behaviors.

We note that our approach builds on well-established algorithms such as PPO and other constrained reinforcement learning methods (e.g., CPO, LAG-PPO), which are known to have theoretical convergence guarantees under certain conditions, as discussed in previous work [43, 40, 1]. However, since our method relies on gradient-based optimization, the convergence guarantees are inherently limited by the stochasticity involved in the training process. Therefore, while our approach inherits the general convergence properties of the underlying framework, the practical learning process may vary during stochastic execution, especially given the complex, multi-step nature of our constraints.

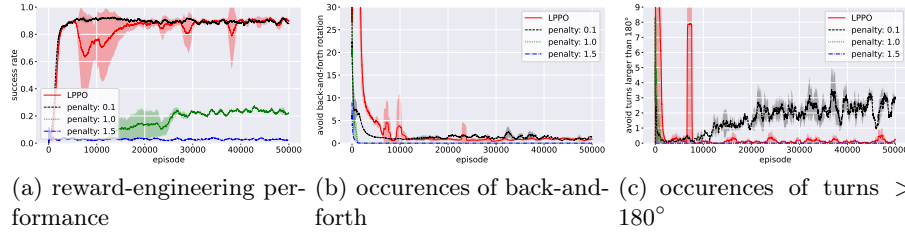
## 5 Evaluation

We performed the 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



**Fig. 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.

seeds for each algorithm, reporting the mean and standard deviation for each learning curve, following the guidelines for a fair comparison of Colas et al. [7]. Fig. 5 depicts a detailed comparison between policies trained with a standard end-to-end PPO [43] (the baseline), and those trained using our constrained method with the injection of rules. In particular, (a) and (d) show the results of policies trained with just *avoid back-and-forth rotation* added as a constraint, while (b), (c), (e), and (f) show the result with all the rules active. The subfigure (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%. Subfigure (d) shows a comparison of the frequency of undesired behavior occurrences between the baseline and our policies, where the frequency diminishes *almost completely*. Subfigures (c), (e), and (f), compare the frequency of the occurrence of undesired behaviors between the baseline and the policies trained with all rules active. Using the baseline, the frequencies of the three behaviors are respectively around 13, 3, and 17 per episode. The undesired behaviors are removed *almost completely* for the policies trained with our approach. We note that the undesired behavior addressed by the rule *avoid turns larger than  $180^\circ$*  is quite rare in general; and so the statistics reported in (c) were collected over the final 100 training episodes. The results clearly show that our method is able to 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 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 Section 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.

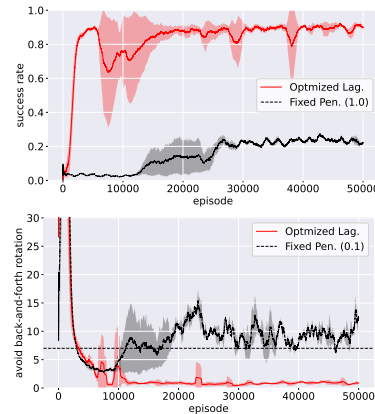


**Fig. 6.** A comparison between results achieved by our approach, denoted by *LPPPO* (red line), with those achieved with a reward-engineering approach.

### 5.1 Comparison with Naïve Penalty-Based Approach

To further validate our approach, we performed a comparison against a reward-shaping approach that penalizes the undesired behavior at each occurrence. Figure 6 shows a set of experiments with different reward penalties (e.g., 0.1, 1.0, or 1.5) compared to our approach (the red line). These results highlight the challenge of finding the correct balance for the penalty, as it necessitates non-trivial parameter tuning. In contrast, our constrained approach offers easier tuning and guarantees a more general and flexible solution for such problems. This finding is consistent with the conclusions of Kamran et al. [22] and Roy et al. [41] in their recent works on the subject.

In Yerushalmi et al. [52], the authors proposed integration between SBP and DRL using a reward-shaping 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 aforementioned work, which results in high-performing agents and fewer rule violations. In Fig. 7, we provide an extensive comparison between the results of our approach and those of Yerushalmi et al. [52]. As shown in the figure, 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 using a penalty value of 0.1, of Yerushalmi et al. [52], 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 differ-



**Fig. 7.** Comparison between our approach and a reward-shaping method.

entiate between optimizing the main reward and minimizing the costs. This differentiation presents significant advantages, including: (i) Allows the setting of constraint thresholds independently for each rule/property and the handling of multiple such constraints in the same way, unlike reward-shaping methods that only allow a global minimization to zero of the total cost. (ii) Separates reward maximization from cost minimization, simplifying the reward engineering task. (iii) 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. (iv) Introduces novel numerical optimizations to the training phase, resulting in a more stable algorithm with a higher cumulative reward.

Additional results can be found in Appendix B where we analyze the trained policies exploiting the exciting field of *verification for neural network* to formally show how our approach can improve the reliability of the trained agents.

## 5.2 Discussion and Limitations

While our method shows strong performance in the scenarios we evaluated, we recognize that there might be limitations to some CMDPs. One potential limitation is CMDPs with highly complex or conflicting constraints, where the cost functions may create trade-offs with rewards that are difficult to optimize. In such cases, finding a feasible policy that satisfies all constraints while optimizing for performance can lead to suboptimal exploration and convergence behavior. This could result in overly conservative policies where the agent prioritizes constraint satisfaction over the reward function, thereby limiting its ability to find high-performing policies. Another challenge may arise in environments with dynamic or non-stationary constraints. Since our approach assumes that constraints remain consistent during training, sudden changes in constraints or their thresholds could disrupt the learning process, requiring the algorithm to adapt to new conditions, which is not explicitly handled by our current method. Finally, our method, like other gradient-based algorithms, may struggle in CMDPs with sparse rewards or constraints that are rarely violated. In such scenarios, the agent may not receive sufficient feedback to effectively balance constraint satisfaction with reward maximization.

## 6 Related Work

To the best of our knowledge, this is the first work that combines scenario-based programming into training a constrained deep reinforcement learning system — specifically, in a robotic environment. While there are related works, the closest one is the study by Yerushalmi et al. [52], which proposed the integration of SBP and DRL using a reward-engineering approach. However, this approach inherits the limitations of incorporating a fixed penalty into the reward function. In another recent work on constrained reinforcement learning [41], 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. Additionally, they did not employ a framework specifically designed for constraint encoding, such as SBP. Moreover, previous works focused on game development and synthetic environments, which differ from our robotic domains and present distinct challenges, such as safety and efficiency, are not considered crucial requirements. In this paper, we adopted Lagrangian PPO as the basic building block for our algorithm. However, it is important to note that the adoption of SBP as a framework to design the requirements is agnostic to the optimization algorithm of choice. In this vein, we believe that alternative approaches to solving a CMDP should be explored. In particular, the alternative family of algorithms that rely on convex optimization, such as CPO [1], CUP [51], or FOCOPS [55]. An alternative family of approaches to guarantee safety and additional requirements exploits *safety shields*. Approaches from this family try to enforce respect for the constraints via hardcoded shielding methods [44, 49]. Although these approaches guarantee the respect of the requirements by construction, they rely heavily on prior knowledge and often restrict the agent’s ability to learn original strategies for problem-solving. In the case study work from Kamran et al. [22], the authors show that restricting the search space often produces over-conservative behavior that can potentially lead to a stalemate of the system.

## 7 Conclusion

In this paper, we presented a novel and generic approach for incorporating subject-matter-expert knowledge directly into the DRL learning process, allowing us to achieve user-defined safety properties and behavioral requirements. We showed how to encode the desired behavior as constraints for the DRL algorithm and improve a state-of-the-art algorithm with various optimizations. Importantly, we define properties comprehensibly, leveraging scenario-based programming to encode them into the training loop. We apply our method to a real-world robotic problem, namely mapless navigation, and show that our method can produce policies that respect all the constraints without adversely affecting the main objective of the optimization. Future work will focus on exploring the scalability of our approach, particularly in systems with a large number of constraints. Although constrained-based methods typically scale better than reward-shaping approaches [41], further investigation is needed to fully understand the computational and performance trade-offs in large-scale environments. Moreover, we plan to extend our work to different robotics environments, including navigation in more complex domains (e.g., air and water), manipulation (e.g., grasping), and medical applications, where safety is a crucial requirement.



## Bibliography

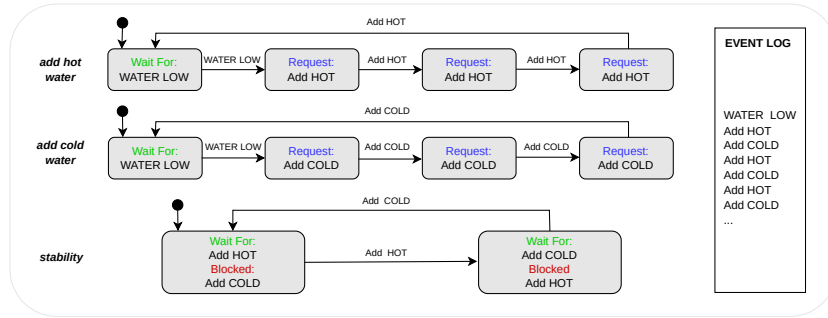
- [1] J. Achiam, D. Held, A. Tamar, and P. Abbeel. Constrained Policy Optimization. In *Proc. 34th Int. Conf. on Machine Learning (ICML)*, 2017.
- [2] G. Alexandron, M. Armoni, M. Gordon, and D. Harel. Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking. In *Proc 36th Int. Conf. on Software Engineering (ICSE)*, 2014.
- [3] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
- [4] G. Amir et al. Verifying learning-based robotic navigation systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2023.
- [5] R. Amsters and P. Slaets. Turtlebot 3 as a Robotics Education Platform. In *Proc. 10th Int. Conf. on Robotics in Education (RiE)*, 2019.
- [6] E. Bacci et al. Verifying Reinforcement Learning Up to Infinity. In *Proc. 30th Int. Joint Conf. on Artificial Intelligence*, 2021.
- [7] C. Colas et al. A hitchhiker’s guide to statistical comparisons of reinforcement learning algorithms. *arXiv preprint arXiv:1904.06979*, 2019.
- [8] D. Corsi, E. Marchesini, and A. Farinelli. Formal Verification of Neural Networks for Safety-Critical Tasks in Deep Reinforcement Learning. In *Proc. 37th Conf. on Uncertainty in Artificial Intelligence (UAI)*, 2021.
- [9] D. Corsi, G. Amir, G. Katz, and A. Farinelli. Analyzing adversarial inputs in deep reinforcement learning. *arXiv preprint arXiv:2402.05284*, 2024.
- [10] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Journal on Formal Methods in System Design (FMSD)*, 2001.
- [11] J. Du. Understanding of object detection based on cnn family and yolo. In *Journal of Physics: Conference Series*, 2018.
- [12] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2021.
- [13] I. Goodfellow et al. *Deep Learning*. MIT Press, 2016.
- [14] M. Gordon et al. Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th ACM Conf. on Innovation and Technology in Computer Science Education (ITCSE)*, 2012.
- [15] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.
- [16] D. Harel and R. Marelly. *Come, Let’s Play: Scenario-Based Programming using LSCs and the Play-Engine*. Springer Science & Business Media, 2003.
- [17] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM (CACM)*, 55(7):90–100, 2012.
- [18] D. Harel et al. Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems*, 2012.

- [19] X. Huang et al. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, 2017.
- [20] A. Juliani et al. Unity: A General Platform for Intelligent Agents, 2018. Technical Report. <https://arxiv.org/abs/1809.02627>.
- [21] J. Jumper et al. Highly accurate protein structure prediction with alphafold. *Nature*, 2021.
- [22] D. Kamran et al. A modern perspective on safe automated driving for different traffic dynamics using constrained reinforcement learning. In *IEEE 25th International Conference on Intelligent Transportation Systems*, 2022.
- [23] G. Katz et al. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification*, 2017.
- [24] W. Koch et al. Reinforcement learning for uav attitude control. *ACM Transactions on Cyber-Physical Systems*, 2019.
- [25] J. Kulhánek et al. Vision-based navigation using deep reinforcement learning. In *2019 European Conference on Mobile Robots (ECMR)*, 2019.
- [26] Y. Li. Deep Reinforcement Learning: An Overview, 2017. Technical Report. <http://arxiv.org/abs/1701.07274>.
- [27] Y. Liu, J. Ding, and X. Liu. Ipo: Interior-Point Policy Optimization under Constraints. In *Proc. 34th AAAI Conf. on Artificial Intelligence*, 2020.
- [28] Z. Lyu et al. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, 2020.
- [29] E. Marchesini et al. Discrete Deep Reinforcement Learning for Mapless Navigation. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 2020.
- [30] E. Marchesini et al. Benchmarking Safe Deep Reinforcement Learning in Aquatic Navigation. In *Conf on Intelligent Robots and Systems (IROS)*, 2021.
- [31] E. Marchesini et al. Exploring Safer Behaviors for Deep Reinforcement Learning. In *Proc. 35th AAAI Conf. on Artificial Intelligence*, 2021.
- [32] A. Marron et al. Embedding scenario-based modeling in statecharts. In *MODELS workshops*, 2018.
- [33] V. Mnih et al. Playing Atari with Deep Reinforcement Learning, 2013. Technical Report. <https://arxiv.org/abs/1312.5602>.
- [34] C. Nandkumar, P. Shukla, and V. Varma. Simulation of Indoor Localization and Navigation of Turtlebot 3 using Real Time Object Detection. In *Conf. on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, 2021.
- [35] G. Nanfack, P. Temple, and B. Frénay. Learning customised decision trees for domain-knowledge constraints. *Pattern Recognition*, 2023. URL <https://www.sciencedirect.com/science/article/pii/S0031320323003114>.
- [36] H. Nguyen and H. La. Review of deep reinforcement learning for robot manipulation. In *Third IEEE International Conference on Robotic Computing (IRC)*, 2019.
- [37] M. Pfeiffer, S. Shukla, M. Turchetta, C. Cadena, A. Krause, R. Siegwart, and J. Nieto. Reinforced Imitation: Sample Efficient Deep Reinforcement Learning for Mapless Navigation by Leveraging Prior Demonstrations. *IEEE Robotics and Automation Letters*, 3(4):4423–4430, 2018.

- [38] A. Pore et al. Safe Reinforcement Learning using Formal Verification for Tissue Retraction in Autonomous Robotic-Assisted Surgery. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2021.
- [39] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, 2009.
- [40] A. Ray, J. Achiam, and D. Amodei. Benchmarking Safe Exploration in Deep Reinforcement Learning, 2019. Technical Report. <https://cdn.openai.com/safexp-short.pdf>.
- [41] J. Roy, R. Girgis, J. Romoff, P. Bacon, and C. Pal. Direct behavior specification via constrained reinforcement learning. *arXiv preprint arXiv:2112.12228*, 2021.
- [42] A. Sallab, M. Abdou, E. Perot, and S. Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017.
- [43] J. Schulman et al. Proximal Policy Optimization Algorithms, 2017. Technical Report. <http://arxiv.org/abs/1707.06347>.
- [44] K. Srinivasan, B. Eysenbach, S. Ha, J. Tan, and C. Finn. Learning to be safe: Deep rl with a safety critic. *arXiv preprint arXiv:2010.14603*, 2020.
- [45] A. Stooke et al. Responsive Safety in Reinforcement Learning by Pid Lagrangian Methods. In *Proc. 37th Int. Conf. on Machine Learning*, 2020.
- [46] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.
- [47] L. Tai, G. Paolo, and . Liu. Virtual-to-Real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2017.
- [48] C. Tessler, D. Mankowitz, and S. Mannor. Reward constrained policy optimization. In *International Conference on Learning Representations*, 2018.
- [49] B. Thananjeyan et al. Recovery rl: Safe reinforcement learning with learned recovery zones. *IEEE Robotics and Automation Letters*, 2021.
- [50] S. Wang et al. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.
- [51] L. Yang et al. Constrained update projection approach to safe policy optimization. *Advances in Neural Information Processing Systems*, 2022.
- [52] R. Yerushalmi et al. Scenario-Assisted Deep Reinforcement Learning. In *Conf. on Model-Driven Engineering and Software Development*, 2022.
- [53] J. Zhang et al. Deep reinforcement learning with successor features for navigation across similar environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.
- [54] L. Zhang et al. Penalized proximal policy optimization for safe reinforcement learning. *Int. Joint Conf. on Artificial Intelligence*, 2022.
- [55] Y. Zhang, Q. Vuong, and K. Ross. First order constrained optimization in policy space. *Advances in Neural Information Processing Systems*, 2020.
- [56] W. Zhao, J. Queralta, and T. Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020.
- [57] K. Zhu and T. Zhang. Deep reinforcement learning based mobile robot navigation: A review. *Tsinghua Science and Technology*, 2021.

## A Scenario-Based Programming: Concrete Example

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. 8 uses that representation to depict a simple SB program that controls the temperature and water level in a water tank (borrowed from [18]). 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, 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. SBP is an attractive choice for the incorporation of domain-expert knowledge, expressing safety-related and other constraints into a DRL agent training process, due to its support of the event-blocking idiom, as well as being formal and fully executable [14, 2], e.g., enabling formal verification of the constraints model. However, the approach presented in this paper is not specific to SBP and can be easily extended to use other frameworks and methods to express safety-related and other constraints, as long as these enable constraint violation monitoring during the agent learning process, such as in the works of Harel [15] and Marron et al. [32], as well as decision trees [35].



**Fig. 8.** A scenario-based program for controlling a water tank. The small black circle indicates the initial state.

## B Deep Neural Networks and Formal Verification of DNNs

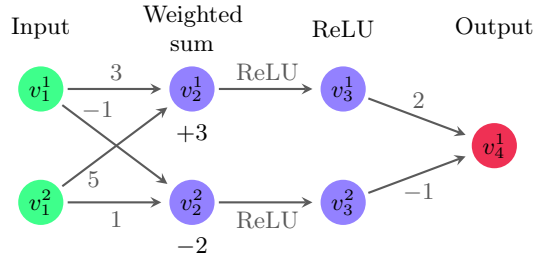
A deep neural network (DNN) [13] is a computational, directed graph, that includes various types of layers — in which each layer includes various nodes (“neurons”). After receiving a value through the first layer (the “input” layer), the network propagates the value through the computational layers (the “hidden”

layers), until reaching the network’s final layer (the “output” layer). The outputs generated by the final layer can either be a classification label, or a regression value, depending on the DNN in question, and its training process. In each of the hidden layers, the computation is based on the *type* of activation characterizing the neurons of each layer. For example, in the common *ReLU* (*rectified linear unit*) layer, each neuron  $y$  calculates the value  $y = \text{ReLU}(x) = \max(x, 0)$ , for a value  $x$  of the matching neuron in the preceding layer. Additional layer types include weighted-sum layers, and various layers with non-linear activations (such as *sign* activations, *max-pooling*, and more). Here, we focus on *feed-forward* DNNs, i.e., networks in which each layer is connected exclusively to its subsequent layer. An example of a toy, feed-forward DNN appears in Fig. 9.

A DNN verification algorithm receives the following inputs [23]: a trained DNN  $N$ , a precondition  $P$  on the DNN’s inputs, and a postcondition  $Q$  on  $N$ ’s output. The precondition is used to limit the input assignments to inputs of interest or to express some assumption the user has regarding the environment (e.g., that an image-recognition DNN will only be presented with certain pixel values). The postcondition typically encodes the *negation* of the behavior we would like  $N$  to exhibit on inputs that satisfy  $P$ . Then, the verification algorithm searches for an input  $x'$  that satisfies the given conditions (i.e.,  $P(x') \wedge Q(N(x'))$ ), and returns exactly one of the following outputs:

1. **SAT**, indicating the query is satisfiable. Due to the postcondition  $Q$  encoding the negation of the required property, this result indicates that the wanted property is violated in some cases. Modern verification engines also supply a concrete input  $x'$  that satisfies the query, and hence, a valid input that triggers a bug, such as an incorrect classification; or
2. **UNSAT**, indicating that there does not exist such an  $x'$ , and thus — that the desired property always holds.

For example, suppose we wish to guarantee that for all non-negative inputs  $x = \langle v_1^1, v_1^2 \rangle$ , the DNN in Fig. 9 always outputs a value strictly smaller than 40; i.e., that that  $N(x) = v_4^1 < 40$ . This property can be encoded as a verification query consisting of a precondition that restrict the inputs to the desired range,



**Fig. 9.** A toy DNN.

ALGO	back-and-forth rotation			turns larger than 180°			turning when clear		
	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 1.** 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.

i.e.,  $P = (v_1^1 \geq 0) \wedge (v_1^2 \geq 0)$ , and by setting  $Q = (v_4^1 \geq 40)$ , which is the *negation* of the desired property. In this case, a sound verifier will return **SAT**, alongside a feasible counterexample such as  $x = \langle 2, 3 \rangle$ , which produces the output  $v_4^1 = 48 \geq 40$  when fed to the DNN. Hence, the property does not always hold. Originally, DNN verification engines were designed to verify the correct behavior of feed-forward DNNs [23, 50, 28, 19]. However, in recent years, the verification community has also designed verification methods tailored for DRL systems [8, 6, 12, 3, 4]. These methods include techniques for encoding multiple invocations of the agent in question when interacting with a reactive environment over multiple time-steps.

## Results

As an additional means of proving the effectiveness of our method, we ran formal verification queries relating to the aforementioned undesirable behaviors. In order to conduct a fair comparison, we selected only models that passed our success cutoff value (85%); and 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 1 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.