Saarland University

Master's Thesis

Classification and Analysis of Safe Reinforcement Learning Algorithms

Submitted by:

Philipp Sauer

Submitted on:

July 22, 2025

Reviewers:

Univ.-Prof. Dr. Verena Wolf

Univ.-Prof. Dr. Dietrich Klakow

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, July 22, 2025

(Unterschrift/Signature)

Abstract

Reinforcement learning algorithms learn to solve tasks in unknown environments via trial-and-error interactions with the environment. When we want to apply our algorithms to problems in the real world, it is not only important to find an optimal solution, but the safety of all actors involved also needs to be guaranteed. Early safe reinforcement learning methods focused on constrained optimization to simultaneously converge to an optimal and safe policy. More recent works have focused on safety shields that aim to identify and block unwanted behavior. Though much research has been done on how to guarantee safety in reinforcement learning, the question of which method is most effective is still open.
In this thesis, we aim to tackle this question by performing an elaborate analysis of the two main classes of algorithms: constrained optimization and safety shields. We classify existing works based on their characteristics and highlight certain advantages and disadvantages of each individual method. Furthermore, we evaluate a subset of algorithms in a diverse set of environments, including the popular but challenging SafetyGymnasium benchmarks. We find that balancing task performance and safety in reinforcement learning is a difficult task and can be greatly affected by the choice of hyperparameters. In addition, the results show that either method can be effective in securing safety depending on the environment and task. To further validate our findings, we also employ PyDSMC for statistical analysis.

Acknowledgments

I would like to express my deepest gratitude to Prof. Dr. Verena Wolf, who provided me the opportunity to pursue this thesis in her area of research. Furthermore, I'm extremely grateful to my supervisor, Timo P. Gros, who always provided constructive feedback, guidance, and enabled a structured and productive work process. Special thanks to Nicola J. Müller and Joshua Meyer, who were always available to answer questions, discuss ideas, and provided invaluable insights. On top of that, I want to thank every single person who was involved in the courses at Saarland University over the last few years. The courses were exceptional, and the learning atmosphere was amazing. It has been a real pleasure to pursue both my Bachelor's and Master's degrees at Saarland University. On a more personal note, I would be remiss in not mentioning the invaluable support of my family, especially my parents, who supported me throughout my studies and my life, and I am forever grateful for their love and support.

Contents

1. Introduction

In recent years, we have seen a rapid increase in the popularity and deployment of artificial intelligence (AI) algorithms in everyday applications. One area of AI that has found widespread adoption in many domains is *reinforcement learning* (RL). RL has shown promising results in a variety of areas, including autonomous driving and robotics, financial applications, healthcare, automation, and gaming.

One key advantage of RL algorithms is their ability to learn directly from their interactions without the need for any data to be collected prior to training, which differs from other supervised learning algorithms. Motivated by how we, as humans, started to learn about the world we live in—by trial and error—RL algorithms exhibit similar behavior: they interact with a potentially unknown environment and learn to distinguish beneficial from adverse actions in order to solve the task at hand. This exploratory approach is what allows them to scale to very large and complex domains, though it also has its drawbacks: it makes the deployment of RL-based agents, especially in safety-critical domains, difficult. For example, in autonomous driving, you definitely want to avoid your car crashing into another car, or in robotics, the robot should not only perform its task but do so while avoiding hurting human beings, other actors, and itself. The problem here is twofold. First, the agent not only has to learn to solve the task, but it also has to develop an understanding of what safe behavior actually means, i.e., what actions it isn't allowed to take at a given point in time. Second, when learning via trial and error in an unknown environment, the agent naturally experiences unsafe behavior. Yet, how can we make sure that the agent remains as safe as possible at all times while learning? These two challenges—the one of learning safety and that of learning safely—are the building blocks of *safe reinforcement learning* (SRL), a subfield of RL that we will explore in this thesis.

Several SRL approaches already exist that address these exact challenges and can provide a certain level of safety guarantees. These guarantees, however, are algorithm-dependent and often also depend on further assumptions, such as having some limited knowledge of the environment, and range anywhere from providing upper bounds on the number of safety violations to all-time safety with zero violations throughout training.

In this thesis, we consider the two main classes of SRL algorithms: those that directly embed safety into the agent's policy via *constrained optimization*, and those that leverage *shielding* as a means to achieve safety. In the context of SRL, the shield is a separate mechanism with the sole purpose of reducing the amount of unsafe behavior that the agent exhibits by either blocking or correcting unsafe actions preemptively. For example, consider an agent driving towards an intersection and another car approaching from the right. The agent might choose to accelerate to reach its destination faster, but the shield would forbid such risky behavior as it would likely result in a crash and force the agent to decelerate. Constrained optimization methods, on the other hand, rely on the agent's policy to handle both objectives simultaneously. In this scenario, based on the information available, the agent itself has to come to the conclusion that a small deceleration is required, although it conflicts with the goal of reaching the target location as quickly as possible.

Contributions As we just saw, these two classes of SRL algorithms have a fundamental design difference: shielding algorithms delegate the safety aspect of the agent to a separate (learned) module, while constrained optimization algorithms force the agent to strike a balance between both objectives. Hence, as part of this thesis, we compare these two classes of SRL algorithms with the aim of answering the following questions:

- What are the requirements, advantages, and disadvantages of each individual algorithm?
- Is one of the two classes of SRL algorithms more effective in preventing unsafe behavior?
- How does the task performance compare?
- Does the modular design benefit the learning process of the agent?

Overall, we tackle these questions by providing an overview and classification of existing SRL algorithms, followed by an extensive comparison of popular constrained optimization and shielding algorithms on state-of-the-art benchmarks.

Thesis Structure We start this thesis by introducing reinforcement learning fundamentals and terminology in Chapter 2 and proceed with an elaboration of important SRL concepts in Chapter 3. In the next section 4, we present relevant existing works in the area of SRL with a focus on constrained optimization and shielding. Furthermore, we provide a classification of these approaches in the same chapter. Chapter 5 discusses the benchmarks we use for our evaluation and the subset of algorithms we consider in our experiments. The next three chapters contain the main results of this thesis. In Chapter 6, we present our overall approach to hyperparameter tuning and our findings on our way to discover promising hyperparameters for the chosen algorithms. These hyperparameters constitute the basis for the evaluation runs in the next chapter 7. Lastly, we also derive statistical guarantees for a subset of interesting metrics in Chapter 8 for each of the considered algorithms and draw our conclusions in Chapter 9. Lots of additional material regarding our setup and extended results can be found in the appendices.

2. Theoretical Background

2.1 Markov Decision Process

Reinforcement learning builds on the concept of *Markov decision processes* (MDPs) [1]. Let $d_S, d_A \in \mathbb{N}$. An MDP is described as a six-tuple: $M = (S, A, r, P, \mu, \gamma)$, where $S \subseteq \mathbb{R}^{d_S}$ is a set of states, $A \subseteq \mathbb{R}^{d_A}$ the set of actions the agent can take, $r : S \times A \times S \to \mathbb{R}$ a reward signal, $P : S \times S \times A \to [0, 1]$ a transition probability function, $\mu : S \to [0, 1]$ the initial state probability distribution, and $\gamma \in [0, 1]$ a discount factor. The set of states S and the set of actions A are task-dependent and can be either discrete or continuous. The RL agent is modeled as a policy π_θ, where θ is a learnable vector of parameters. Unless clear from context, we omit the subscript θ. We consider stochastic policies $\pi : A \times S \to [0, 1]$. $\pi(\cdot|s)$ provides the probability distribution over actions in state s, and $\pi(a|s)$ the probability of choosing action a.
An episode (often also called a trajectory) $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, a_{T-1}, r_T, s_T$ describes a single finite run of the agent in the MDP. Starting at initial state $s_0 \sim \mu(\cdot)$, the agent selects and executes an action $a_0 \sim \pi(\cdot|s_0)$ using its policy, observes the next state $s_1 \sim P(\cdot|s_0, a_0)$ and reward $r_1 = r(s_0, a_0, s_1)$. This process of environment interaction then repeats until the agent hits the time limit of T interaction steps. In episodic MDPs, the episode additionally terminates after $T' \leq T$ steps whenever the agent finishes its task, e.g., by reaching the goal in a driving simulation. By setting the reward of the remaining $T - T'$ steps to 0, we again end up with a T-step episode. We write shorthand $\tau \sim \pi$ to denote a trajectory in which all actions were sampled according to π. Let

$$G_t(\tau) = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-t-1} r_T \tag{2.1}$$

denote the discounted, accumulated reward (also called discounted return) of the trajectory from time step t onwards. The smaller the value of γ, the less emphasis is put on later rewards, with $\gamma = 0$ considering only the immediate next-step reward and $\gamma = 1$ yielding an undiscounted return in which later rewards are not diminished. Making use of this definition, we define the objective $J(\theta)$ of the RL agent as the expected discounted return over all possible trajectories under the current policy, i.e.,

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[G_0(\tau)], \tag{2.2}$$

and the optimal policy

$$\pi^*_\theta = \arg\max_\theta J(\theta) \tag{2.3}$$

maximizes $J(\theta)$. Furthermore, $V_\pi(s) = \mathbb{E}_{\tau\sim\pi}[G_0(\tau) \mid s_0 = s]$ denotes the state value (or value function), which measures the expected discounted return in state s by following policy π, and $Q_\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[G_0(\tau) \mid s_0 = s, a_0 = a]$, the state-action value (or state-action function), additionally conditions on choosing action a in state s. Combining these two measures yields the advantage function $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$.

2.2 Constrained Markov Decision Process

A *constrained Markov decision process* (CMDP) [2] is an extension of the traditional MDP model. It extends the MDP six-tuple M to an eight-tuple $M' = (S, A, r, c, P, \mu, \gamma, \chi)$, where S, A, r, P, μ, and γ are defined as before, $c : S \to \mathbb{R}$ is a cost function, and $\chi \in \mathbb{R}$ denotes a cost threshold. c is task-specific and can be used to model unsafe behavior: a binary value of {0,1} can be used to, for example, indicate whether a state is safe or unsafe, or non-zero values can simply be used to record violations of any constraints.
In CMDPs, whenever the agent executes an action and visits a new state, it observes the cost associated with executing that action on top of the reward. Thus, trajectories take the form $\tau = s_0, a_0, r_1, c_1, s_1, a_1, r_2, c_2, s_2, \ldots, a_{T-1}, r_T, c_T, s_T$, and we analogously define the state, state-action, and advantage functions $V^C_\pi(s)$, $Q^C_\pi(s,a)$, and $A^C_\pi(s,a)$ to measure expected discounted cost returns, and cost objective $J^C(\theta)$ by replacing rewards with costs.
CMDPs have a constrained objective that depends on both objectives $J(\theta)$ and $J^C(\theta)$. We still want to find the optimal parameters θ that maximize $J(\theta)$, while at the same time upper-bounding the cost objective $J^C(\theta)$ by threshold χ. More formally, we have

$$\begin{aligned} \max_\theta \quad & J(\theta), \\ \text{s.t.} \quad & J^C(\theta) \le \chi. \end{aligned} \tag{2.4}$$

The overall safety guarantees then depend on the task at hand. As an example, let c be a binary cost signal with $c(s) = 0$ if visiting s is safe and $c(s) = 1$ otherwise. With $J^C(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G^C_0(\tau)]$ and $\gamma = 1$, we effectively limit the probability of unsafe states being visited by χ. Hence, χ serves as a lever to adjust the desired safety of the agent: smaller values of χ impose stronger restrictions on the agent's behavior, whereas larger values allow for riskier but potentially more rewarding tactics. We can set $\chi = 0$ to steer the agent towards never visiting any unsafe state at all.

3. Safe Reinforcement Learning

In reinforcement learning, an agent learns by interacting with a potentially unknown environment with the aim of maximizing its performance for a specific task. The only measure of success is a stepwise reward signal, which the agent uses to discern good from bad behavior. Safe reinforcement learning additionally requires the agent to simultaneously fulfill a safety-critical goal. For example, in autonomous driving, the agent's primary objective is to reach a designated goal position, but it must do so safely by avoiding collisions with other cars or driving off the road at any time.

3.1 Safety Methods

Safety in RL can be achieved by different means. The first thing that comes to mind is a modification of the stepwise reward feedback to discourage the agent from entering unsafe situations by, for example, assigning large negative rewards to unsafe actions or outcomes. Unfortunately, this method is extremely sensitive. If the negative values dominate, the agent might be overly cautious and fail to solve the task. On the other hand, if the values are not sufficiently penalizing, the agent could choose to ignore the safety-relevant part completely. On top of that, mixing both objectives together by providing a single reward makes it difficult to attest to safety guarantees about the agent's behavior.

A better approach is to keep these objectives separated by providing the agent with two signals each step: a reward that reports task-specific performance and a cost to judge the agent's safety. By separating rewards and costs, we end up with a CMDP, which can be solved by simultaneously optimizing both objectives with, for example, *constrained optimization* methods. The CMDP structure carries several advantages: First, there is no need to manually tune the relative importance of the two signals. Instead, we can automatically find a trade-off that allows task completion while ensuring safety. Second, we can deploy different models to tackle each problem individually. For example, one model can be used to optimize the task objective J, while another is used to optimize J^C, and the agent queries both models to determine its behavior. Finally, by keeping these objectives separated, we can also provide concrete safety guarantees about the agent's behavior.

Another approach that takes advantage of the CMDP definition is *shielding*. The shield is a separate module that judges the safety level of the current state or an action the agent is about to execute. Ideally, the shield should always accurately identify any safety-critical behavior and, in the process, prevent the agent from acting or correct the agent's behavior whenever necessary. Shielding can be applied as some sort of action filter preemptively, or as a kind of judge afterward to block or correct selected actions. Depending on our knowledge about the environment, the shield can either be manually defined, automatically derived from the environment, or learned from experience. The former two require knowledge about the environment, while the latter suffers from being less accurate.

Lastly, safe behavior can also be learned by imitating safe actions from an external source. Imagine having a safety guide (potentially pre-trained in a simulator) during training that can be queried whenever unsure or serves as some kind of backup policy in tricky situations. In extreme cases, this guiding could also be conducted manually by a human, though manual intervention is slow and, as such, does not scale well. Instead of having a guide, it is also possible to imitate safe behavior from a large dataset of expert demonstrations. In this case, the agent is presented with safe and unsafe behavior and learns to mimic the safe behavior while avoiding the unsafe. Similar to supervised learning, this requires a large amount of data that covers all safety-relevant scenarios and can be collected beforehand, which might not be possible in complex domains.

3.2 Safe Exploration

Safe RL algorithms focus on learning a policy that is safe, i.e., our training yields a policy that adheres to the cost constraints and performs well on the task objective J. This is often achieved by first training the agent in a simulation and then transferring the policy to the real world. Provided that the training has converged and that the simulation is sufficiently accurate, the safety guarantees should then also carry over to deployment in the real world. However, in certain situations, it is not sufficient to be able to learn a policy that is safe; we are also required to ensure that the learning process happens in a safe way. This can be the case if the true environment is too complex and we cannot build an accurate simulation, or we need to fine-tune our policy in a real-world environment. In these situations, it is necessary to ensure that the data collection process is also safe, i.e., that the agent safely explores the environment during training without exerting any unsafe behavior that might cause irreparable damage. Compared to learning a safe policy, learning safely is even more challenging given the fundamental idea of RL: learning by trial and error. Hence, some algorithms in SRL also focus on *safe exploration*, which makes them suitable for training directly in the real world.

4. Related Work

In this chapter, we present existing algorithms in the field of SRL with a focus on constrained optimization and shielding. We also provide a classification of these works in Table 4.1, in which we analyze each method with respect to certain attributes.

Constrained Optimization Constrained optimization algorithms are the first major group of algorithms we consider in this work for solving CMDPs. One popular approach is the so-called *Lagrangian* method, which combines both objectives of the CMDP into a single optimization problem and solves it iteratively via alternating gradient ascent and gradient descent. Examples include PPOLag [3], which deploys a Lagrangian on top of the PPO objective to balance safety and performance; SPDSP [4], which uses a Lagrangian to keep the agent in a safe region; and CPO [5], which provides near-constraint satisfaction at each iteration during training by approximating constraints of future policy iterates with the help of a Lagrangian. Furthermore, Stooke et al. [6] combine the Lagrangian approach with control theory to model and solve SRL problems as dynamic systems.
Similar to the idea of PPOLag, to make the policy safety-aware, IPO [7] extends the PPO task objective with a surrogate for the cost term and optimizes the resulting objective via gradient ascent. Another approach is SEO-MDP [8], which estimates costs via Gaussian processes and balances safety and exploration by optimizing an interpolation between a pessimistic and optimistic policy. OptLayer [9], on the other hand, ensures constraint satisfaction by adding a constrained optimization layer on top of the policy network. Though this approach ensures safety throughout training, the safety layer requires precise domain knowledge, which is not always available. Lastly, SMBPO [10] learns the environment dynamics and uses reward penalties such that unsafe actions eventually end up with a lower state-action value than safe actions, and as a result, the agent avoids taking unsafe actions.

Neuro-Symbolic Shields In contrast to these approaches, many works focus on neuro-symbolic shields to assert safe behavior by, for example, guaranteeing that a certain safety specification holds (with high probability) throughout training. One of these works is the original shielding paper [11] by Alshiekh et al., which first formalized the concept of shielding in SRL. These shields allow an agent to conduct safe exploration with no safety violations and provide additional guarantees like *minimal interference* and *correctness*. To accomplish these goals, the approach requires an abstraction of the MDP environment model. Given said abstraction and a safety specification in *Linear Temporal Logic* (LTL), a shield in the form of a reactive system is built. By simulating the execution of an action in the safety-relevant abstraction of the MDP, the shield blocks the execution of unsafe actions in a preemptive manner. Carr et al. [12] extend the idea of shields to partially observable MDPs, and Könighofer et al. [13, 14] present a novel way to apply shielding in probabilistic environments. Due to the probabilistic nature of the environment, the best these shields can do is to limit the probability of constraint violations. Furthermore, PLS [15] presents a probabilistic shield for partially observable MDPs that forces the agent's policy to favor safe actions, and REVEL [16] leverages *formal verification* paired with shields to certify safety throughout the entirety of training. Despite their strong safety guarantees, these approaches require some limited but accurate knowledge about the dynamics of the MDP, which might not always be available.

On the other hand, AMBS [17, 18], ABPS [19], CautiousRL [20], and SPICE [21] learn the environment dynamics and judge safety with the help of the learned model. The former three algorithms estimate the probability of satisfying or violating the given safety specification within the next h steps: AMBS and ABPS sample trajectories in the latent space of the model, whereas CautiousRL relies on recursive calculations over nearby states. The latter, SPICE, uses an approximation of its learned model to derive constraints that the unshielded action has to satisfy. If these constraints turn out to be violated, the action is considered unsafe, and SPICE proposes an alternative safe action. The trade-off these algorithms pay for being oblivious to the true environment dynamics is reflected in their safety guarantees: they cannot certify that the agent will be safe at all times.

Neural Shields Other shielding methods do not directly consider safety specifications but instead bound the cost objective J^C, which we introduced in Chapter 2. These algorithms are commonly found in the model-free setting, in which neither a model is assumed nor inferred. For example, CSC [22] follows the intuitive idea of learning an action safety critic that conservatively estimates the probability that taking an action will result in unsafe behavior. Only actions with a low probability estimate are considered safe for execution in the environment. A similar approach is taken in RecoveryRL [23], StLtR [24], and SAILR [25], the latter using an estimate of the advantage instead of the state-action value. Instead of simply blocking unsafe actions, these three approaches deploy a backup policy that suggests safe, alternative actions. SEditor [26] takes this idea one step further and learns two policies simultaneously—one focused on task performance and the other on safety. When queried, the task policy proposes an action, which is then adjusted by the safety-aware policy. In ADVICE [27], the safety critic is replaced with a K-Nearest-Neighbor classifier that predicts whether a given action is safe to execute in a specific state. Though the idea seems promising, the classifier requires a significant number of unique data samples to be collected first, which makes it impractical for safe exploration.

If we have accurate knowledge of the true environment, MPS [28] and DMPS [29] allow us to conduct safe exploration. The idea is straightforward: simulate the execution of an action and check whether we are able to recover in case the action is indeed unsafe. Only then is the action considered safe and executed; otherwise, the backup policy is used. DMPS improves MPS by additionally deploying a planner for the backup policy that searches for high-performing actions that are safe. Lastly, UNP [30] combines *permissibility* with shielding. The major downside here is that a manual definition of unsafe actions is required, which restricts the applicability of the approach to simpler, known domains.

Further SRL Methods ShieldPPO [31] simply uses a lookup table to store unsafe actions and prevents the agent from re-executing them. RCRL [32] is similar to CautiousRL but additionally takes variance into account to provide a more robust estimate.
Apart from shielding, imitation learning can also be used to mimic safe behavior. GSE [33] first learns a safety-focused policy in simulation and then uses it to guide the agent during training in the real environment. The agent is encouraged to mimic the safe behavior. In contrast to having a policy as a teacher, SIM [34] directly tries to imitate safe, good-performing trajectories while avoiding the bad ones.

Table 4.1 again summarizes the main characteristics of these SRL papers. We grouped the algorithms based on their structure into 5 groups: *Constrained Optimization*, *Neuro-Symbolic Shields*, *Neural Shields*, *Classical Shields* and *Imitation Learning*, where *Classical Shields* refers to algorithms that do not utilize neural networks in their shield. As an example consider ShieldPPO which maintains a simple lookup table of unsafe state-action pairs.
Focusing on the exploration aspect, it is important to note that we chose a hard cut: *safe* indicates that the algorithm can safely train in the environment with zero safety violations, and *unsafe* indicates that unsafe behavior will eventually occur. Though, some of the algorithms that are marked as unsafe can still provide certain safety guarantees, e.g., CPO and CSC approximately enforce constraints in each iteration of training. Nonetheless, the main thing to note is that very few of these algorithms can actually ensure complete safety throughout training, and those that are able to require some limited knowledge of the environment. For example, SRLPS, OSRL and the original shielding paper denoted as *Shield*, all require an abstraction of the true environment, and for safety guarantees to hold, these abstractions need to be accurate. This can be problematic in complex environments which are hard to model accurately. In this case, these algorithms will likely experience unsafe behavior as well.
Hence, we dedicated an additional column to the model knowledge aspect and used *known* to indicate that specific knowledge about the true environment model is required, *free* to indicate that the algorithm is model-agnostic, and *learned* to mark algorithms that learn the environment model from experience. As mentioned, most of the algorithms with safe exploration require some knowledge. Lastly, we also considered the state and action spaces which can be *discrete* or *continuous*, and we used *either* to indicate that the algorithm can work in either of these domains. Here we observe that the majority of algorithms can be used in the more challenging, continuous domains, which is beneficial given that most real-world environments are of continuous nature.

Type	**Algorithm**	**Exploration**	**Model**	**States**	**Actions**
Constrained Optimization	OptLayer [9]	safe	known	continuous	continuous
	SEO-MDP [8]	unsafe	known	discrete	discrete
	CPO [5]	unsafe	free	either	continuous
	PPOLag [3]	unsafe	free	either	either
	CPPO [6]	unsafe	free	either	either
	SPDSP [4]	unsafe	free	either	either
	IPO [7]	unsafe	free	either	either
	SMBPO [10]	unsafe	learned	either	either
Neuro-Symbolic Shield	REVEL [16]	safe	known	continuous	continuous
	Shield [11]	safe	known	discrete	discrete
	SRLKI [12]	safe	known	discrete	discrete
	SRLPS [14]	unsafe	known	discrete	discrete
	OSRL [13]	unsafe	known	discrete	discrete
	PLS [15]	unsafe	known	discrete	discrete
	AMBS [17]	unsafe	learned	discrete	discrete
	CautiousRL [20]	unsafe	learned	discrete	discrete
	AMBS [18]	unsafe	learned	continuous	continuous
	SPICE [21]	unsafe	learned	either	continuous
	ABPS [19]	unsafe	learned	either	either
Neural Shield	MPS [28]	safe	known	either	either
	DMPS [29]	safe	known	either	either
	UNP [30]	safe	known	either	either
	RecoveryRL [23]	unsafe	free	either	either
	SAILR [25]	unsafe	free	either	either
	StLtR [24]	unsafe	free	either	either
	CSC [22]	unsafe	free	either	either
	ADVICE [27]	unsafe	free	continuous	continuous
	SEditor [26]	unsafe	free	continuous	continuous
Classical Shield	RCRL [32]	unsafe	learned	discrete	discrete
	ShieldPPO [31]	unsafe	free	discrete	discrete
Imitation Learning	GSE [33]	unsafe	known	either	either
	SIM [34]	unsafe	free	either	either

Table 4.1.: Classification of SRL algorithms.

5. Algorithms and Benchmarks

This chapter lists the algorithms and benchmarks we chose to include in this thesis. On the algorithm side, we have CPO [5], PPO [35], PPOLag [3], SEditor [26], SPICE [21], and SPICE-CSC [21]. As benchmarks, we use SafetyGymnasium [36] and a set of smaller, static benchmarks, which also appeared in the evaluation of SPICE.

5.1 Algorithms

5.1.1 CPO

CPO is a trust-region policy search method [37] that guarantees reward improvement and near-constraint satisfaction at every iteration during policy optimization. It linearly approximates both the return J and the cost objective J^C of future policy iterates using Taylor series. CPO takes small steps by limiting the Kullback-Leibler (KL) divergence of successive policies, which is itself approximated by a second-order Taylor series. All these approximations are combined into a single primal-dual optimization problem and iteratively solved via alternating gradient ascent and gradient descent during on-policy training.

5.1.2 PPO and PPOLag

PPO is another trust-region policy search method aimed solely at maximizing the discounted return J. In contrast to CPO, PPO limits the divergence of successive policies by incorporating a clipped scaling term into the objective. Though PPO does not take into account the cost objective J^C and thus cannot guarantee safety, we include it in this thesis as a baseline for the performance on the return J. To accommodate the lack of safety in PPO, we additionally consider PPOLag, a version of PPO that incorporates J^C with the help of Lagrange multipliers.

5.1.3 SEditor

SEditor is an off-policy algorithm that uses a two-policy approach to find a safe and effective solution to the CMDP problem. The idea is to train a second policy alongside the first one that turns each preliminary and potentially unsafe action into a safe one.

The first policy π_ϕ is aimed at maximizing the discounted return J, and the second policy π_ψ has the goal of minimizing J^C. Both policies interact in the following way using an action editing function h:

1. π_ϕ proposes an action a in state s
2. π_ψ proposes a modification Δa for a in s
3. execute action $\hat{a} = h(a, \Delta a)$

SEditor additionally employs a distance function that penalizes actions far from the original action. Based on its design, the second policy can be seen as a shield that corrects unsafe actions.

5.1.4 SPICE and SPICE-CSC

SPICE and SPICE-CSC tackle the CMDP problem by learning an approximate model M of the environment. Both algorithms then utilize M during training to collect further data for policy optimization. The idea is that the number of safety violations during training can be decreased by sampling from the learned model. On top of that, SPICE uses M as a shield that corrects unsafe actions such that they satisfy a set of constraints ϕ_H for the next H steps as follows:

1. π proposes an action a in state s
2. obtain an approximation f of the learned model M
3. calculate base constraints ϕ_0 iteratively using ϕ_H and f
4. find an action sequence $a_0, \ldots, a_H$ that satisfies ϕ_0 in s
5. execute action a_0

Similar to SEditor, SPICE also employs a distance function that penalizes actions far from the original action.

In contrast to SPICE, SPICE-CSC follows the idea of CSC [22] to utilize a learned safety critic Q^C for shielding instead. Given a fixed threshold ϵ, in a state s, every action a with $Q^C(s, a) \geq \epsilon$ is considered unsafe and rejected. The policy resamples actions for a maximum of 100 iterations until a safe action is found; otherwise, the safest action with the minimum state-action value $Q^C(s, a)$ is executed.

5.2 Benchmarks

5.2.1 SafetyGymnasium

SafetyGymnasium [36] is a collection of different SRL benchmarks. We focus on the original SafetyGym [3] benchmark, coined SafeNavigation in the SafetyGymnasium package. In this work, we utilize the following four environments:

SafetyPointGoal1-v0 (SPG1): A small robot has to navigate to a designated goal position while avoiding unsafe, hazardous regions and avoiding hitting movable obstacles such as vases. At each timestep, the agent can adjust the rotation and translation of the robot.

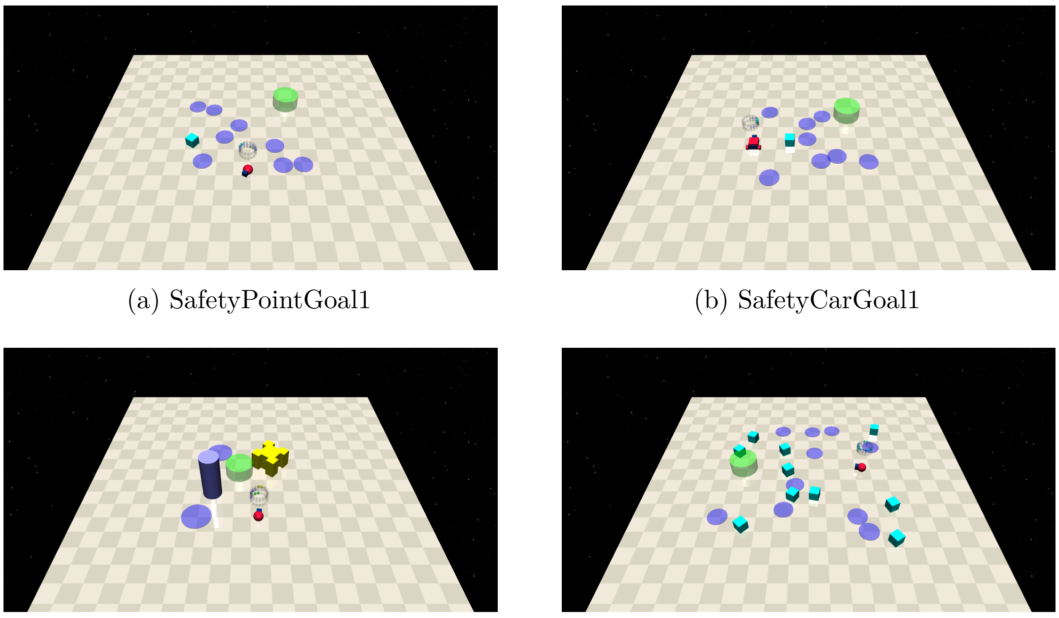
SafetyPointGoal2-v0 (SPG2): This is the same environment as SPG1, but with a higher difficulty level, meaning even more obstacles and hazardous regions.

SafetyCarGoal1-v0 (SCG1): The point robot is replaced with a simplified version of a car. As such, the agent now also has to learn to coordinate the two wheels of the car in order to make progress. The environment layout and task remain unchanged.

SafetyPointPush1-v0 (SPP1): This environment again features the point robot from SPG1, but now the task is different: the robot has to push a box into a specified goal area while avoiding entering unsafe regions and avoiding collisions with obstacles.

All robots are equipped with a lidar that provides a 360-degree scan of the environment and detects nearby objects, obstacles, hazards, and the goal. Given this limited information and the current physical state of the robot, the agent has to learn a policy that safely completes the task. Whenever the robot exhibits unsafe behavior by, for example, hitting an obstacle or entering a hazardous region, the episode is immediately terminated and a cost of 1 is recorded. Furthermore, SafeNavigation environments are non-episodic and have a random layout each episode, which makes it even harder to learn an effective, safe policy. From now on, we omit the suffix *-v0* from the environment names.

Figure 5.1 additionally illustrates the four SafeNavigation tasks. The agent is shown in red, the goal area in green, hazards and obstacles in a bluish color, and the pushable box in yellow. Above the agent, one can also observe the lidar scan of the environment, with the color intensity indicating the distance to objects.


(a) SafetyPointGoal1 (b) SafetyCarGoal1

(c) SafetyPointPush1 (d) SafetyPointGoal2

Figure 5.1.: Illustration of the four SafeNavigation environments.

5.2.2 Static Environments

As our second benchmark, we chose three simpler, static environments that also appeared in the evaluation of SPICE [21]. We use the following three environments, again with a focus on driving and goal reaching:

> **CarRacing (CR)**: The agent has to drive from the start to a specified goal area and back again, all while avoiding the obstacle in the middle.
>
> **MidObstacle (MO)**: Simpler version of *CarRacing*, without the need to drive back to the start.
>
> **Road2D (R2D)**: Similar to *MidObstacle*, but instead of having an obstacle, a speed limit is enforced.

In these three tasks, the agent is only aware of its current position and velocities in the x- and y-directions—no lidar information is available. The velocities can be directly adjusted at each timestep to steer the agent towards the goal area. We again immediately terminate the episode if any unsafe behavior is detected. Furthermore, these benchmarks are episodic and have a consistent, static layout, in contrast to the non-episodic and randomized SafeNavigation environments. The latter is crucial, since the lack of lidar information makes it impossible to learn safe and performant behavior in dynamic, changing environments.

6. Hyperparameter Tuning

Our overall evaluation is split into three parts. First, we tune the hyperparameters of the algorithms to be evaluated. We then utilize the tuned hyperparameters to perform a thorough evaluation on our two benchmarks. Lastly, we derive statistical guarantees for our evaluation runs. This chapter discusses our hyperparameter tuning approach and the results we obtained.

6.1 Tuning Setup

In this thesis, we make use of the Optuna [38] framework for automatic hyperparameter tuning. Optuna strategically samples the user-defined hyperparameter landscape to discover promising hyperparameter configurations that maximize (or minimize) a chosen objective function. We limit our tuning to the algorithms CPO, PPO, PPOLag, and SEditor. The other algorithms, unfortunately, do not leverage any form of vectorization for training. As a consequence, to obtain meaningful results, the time required for hyperparameter tuning would be too high, which is why we only focus on the aforementioned algorithms. Our choice of tunable hyperparameters mainly considers learning rates and discount factors, though there are plenty of other algorithm-dependent options. For every single hyperparameter, we use its default value as a starting point to construct the sampling intervals for the sampler of Optuna. The set of hyperparameters we consider for each algorithm, including their intervals, is shown in Table 6.1.

We tune every algorithm on a per-environment basis with a soft timeout of 96 hours, meaning that after 96 hours, no further trials can be scheduled and Optuna will terminate once the current trial is completed. In each trial, Optuna suggests a set of hyperparameter values, and we train three agents simultaneously on seeds 100, 101, and 102 for a maximum of 2.5 million steps each, which equals 25% of the total number of steps we plan to use for our evaluation runs. During training, we monitor both undiscounted objectives J and J^C, and define a score

$$J^S = b + J \times \begin{cases} \max(1, \frac{J^C}{\chi}), & \text{if } J < 0 \\ \min(1, \frac{\chi}{J^C + \epsilon}), & \text{otherwise} \end{cases} \tag{6.1}$$

which combines these two metrics into a single objective J^S. Here, ϵ is a small constant set to 10^{-8}, and $b = \min(0, \chi - J^C)$ is used as a tiebreaker at $J = 0$. If $J^C \leq \chi$, then $J^S \approx J$, and if $J^C > \chi$, then $J^S < J$, proportional to a ratio between J^C and χ. In the case of PPO, which we include as a baseline for the performance on the

Algorithm	**Hyperparameter**	**Interval**	
		From	To
CPO	gamma	0.9	0.999
	cost_gamma	0.9	0.999
	critic_lr	0.0001	0.01
PPO	gamma	0.9	0.999
	actor_lr	0.00001	0.001
	critic_lr	0.0001	0.01
PPOLag	gamma	0.9	0.999
	cost_gamma	0.9	0.999
	actor_lr	0.00001	0.001
	critic_lr	0.0001	0.01
	lambda_lr	0.0001	0.1
SEditor	discount	0.9	0.999
	actor_lr	0.00001	0.001
	d_actor_lr	0.00001	0.001
	critic_lr	0.0001	0.01
	reward_weight_lr	0.0001	0.1

Table 6.1.: Hyperparameter landscape per algorithm for hyperparameter tuning with Optuna.

return J, we simply set $J^S = J$. The final score during tuning at a given step is calculated as the mean of the three individual scores J^S and serves two purposes. On one hand, it is used to determine the best hyperparameter configuration—the one with the highest score after 2.5 million steps—which we then use in the evaluation runs. Furthermore, Optuna uses this metric to decide whether or not to prune the current hyperparameter configuration early; the earliest we allow it to be pruned is after 10 completed trials and one million environment steps in the current trial. Once a trial has been completed, Optuna strategically suggests a new hyperparameter configuration, and the cycle continues unless the timeout is reached.

We slightly adjust these settings for the three static environments, since, in theory, these should not be as complex to solve as the SafeNavigation tasks. A summary of our settings can be found in Table 6.2. Note that SEditor requires a definition of a per-step cost limit. Since we terminate episodes whenever a non-zero cost is recorded, we can treat the step-wise cost limit as a probability for episode termination and end up with a binomial distribution. Given the maximum allowed episode length T and either cost limit χ or the step-wise cost limit p, we can derive the missing value following the relation

$$\chi = 1 - (1 - p)^T \tag{6.2}$$

on a per-environment basis.

	SafeNavigation	**Static Environments**
Timeout	96h	60h
Parallel Runs	3	3
Seeds	[100,101,102]	[100,101,102]
Training Steps	2,500,000	1,250,000
Objective (maximize)	$\mathbb{E}[J^S]$	$\mathbb{E}[J^S]$
Cost Limit χ	0.394	0.01
Pruning Start Steps	1,000,000	500,000
Pruning Start Trials	10	10
Pruner	MedianPruner	MedianPruner
Sampler	TPESampler	TPESampler

Table 6.2.: Hyperparameter tuning settings per benchmark.

6.2 Tuning Results

We limit our discussion of the tuning results to only two environments: SafetyPointGoal1, which we chose as our SafeNavigation baseline, and MidObstacle, which ranks at medium difficulty among the other environments. The results are shown in Figure 6.1 and Figure 6.2, respectively, with figures for the remaining environments in Appendix C. Note that we set a timeout instead of testing a fixed number of configurations. Due to several differences in network complexity and network update patterns between these algorithms, the number of completed trials per algorithm can differ significantly.

6.2.1 SafetyPointGoal1

First of all, note that in the SafeNavigation tasks, the score is a soft indicator of the number of goals reached by the agent under the assumption that the cost limit is respected. Thus, judging solely by the score, we can already see that SEditor quickly learns a performant policy in this limited timeframe, while PPOLag struggles. CPO, known to be rather slow and more conservative, and PPO also converged to policies with a competitive score.
Furthermore, we find that CPO, PPO, and PPOLag average a higher score if the discount *gamma* is set to a higher value close to 1, whereas SEditor performs best with the discount chosen around 0.95. This indicates a preference for later rewards over immediate ones, which can be attributed to the structure of the task. For example, if there is an obstacle (or a line of obstacles) standing between the agent and the goal, then the agent is forced to take a detour to reach the goal safely, which puts more emphasis on later rewards. The difference in the optimal values for the discount factors between CPO and SEditor could also be directly related to the design of the two frameworks the algorithms originate from. OmniSafe, which contains the implementations for CPO, PPO, and PPOLag, uses two separate discount factors: *gamma* for rewards and

cost_gamma for costs. In contrast, SEditor (implemented in ALF) has only a single hyperparameter called *discount*, which serves as both a reward discount and a cost discount. Looking at the cost discount for CPO and PPOLag, we see small to medium values leading to higher scores, which highlights the need for the agent to attend to immediate danger. In the case of SEditor, the single discount factor of 0.95 tries to strike a balance between securing long-term rewards and avoiding immediate danger. Scores also show a dependence on the learning rates, with lower values leading to favorable results across the board. The only exception is PPO, for which an actor learning rate in the upper half of our search space correlates with a higher score.

6.2.2 MidObstacle

First of all, note that MidObstacle features a goal-directed reward based on the distance to the goal, and compared to the SafeNavigation tasks, the rewards have greater magnitude. This, in turn, leads to greater fluctuations in performance and the final score, resulting in a noisier score distribution. As a consequence, outliers with significantly lower scores negatively impact the coloring of our plots. Nonetheless, we chose to display all trials and, as a remedy to the aforementioned problem, additionally confirmed our conclusions by examining the raw numbers of the best-performing runs. For MidObstacle, a score higher than -1,000 can be indicative of a successful run.

Looking at CPO, we still observe a clear preference for lower critic learning rates and cost discounts, while *gamma* fluctuates heavily and shows no clear trend. PPO still prefers low critic learning rates and a higher value for the discount, but unlike the SafetyPointGoal1 results, this time also an actor learning rate in the bottom half of the interval. A lower actor learning rate leads to slower, but potentially more stable learning of the policy and could be a direct consequence of having rewards with greater magnitude.
When examining PPOLag, the first thing to note is that its scores are not very spread out and are primarily clustered around -40,000. Given our design of the score J^S, this suggests that PPOLag almost exclusively crashes into the obstacle on its way to the goal area. Our manual investigation revealed that this is also the case for PPO, though the score for PPO does not reflect this equally well, as it only considers the return J and not J^C. We will elaborate further on this in the next two chapters. Aside from that, we observe the same hyperparameter trends for PPOLag as in SafetyPointGoal1, with a preference for low learning rates and high values for both discount factors. Furthermore, SEditor also shows a consistent preference for low learning rates and a discount factor between 0.95 and 0.99 to achieve higher scores.
On a final note, when looking again at the scores of CPO and SEditor, the importance of hyperparameter tuning becomes clear. Even small changes in the hyperparameters can have a significant impact on the overall performance of the agent, ranging from consistently reaching the goal to consistently crashing into the obstacle.

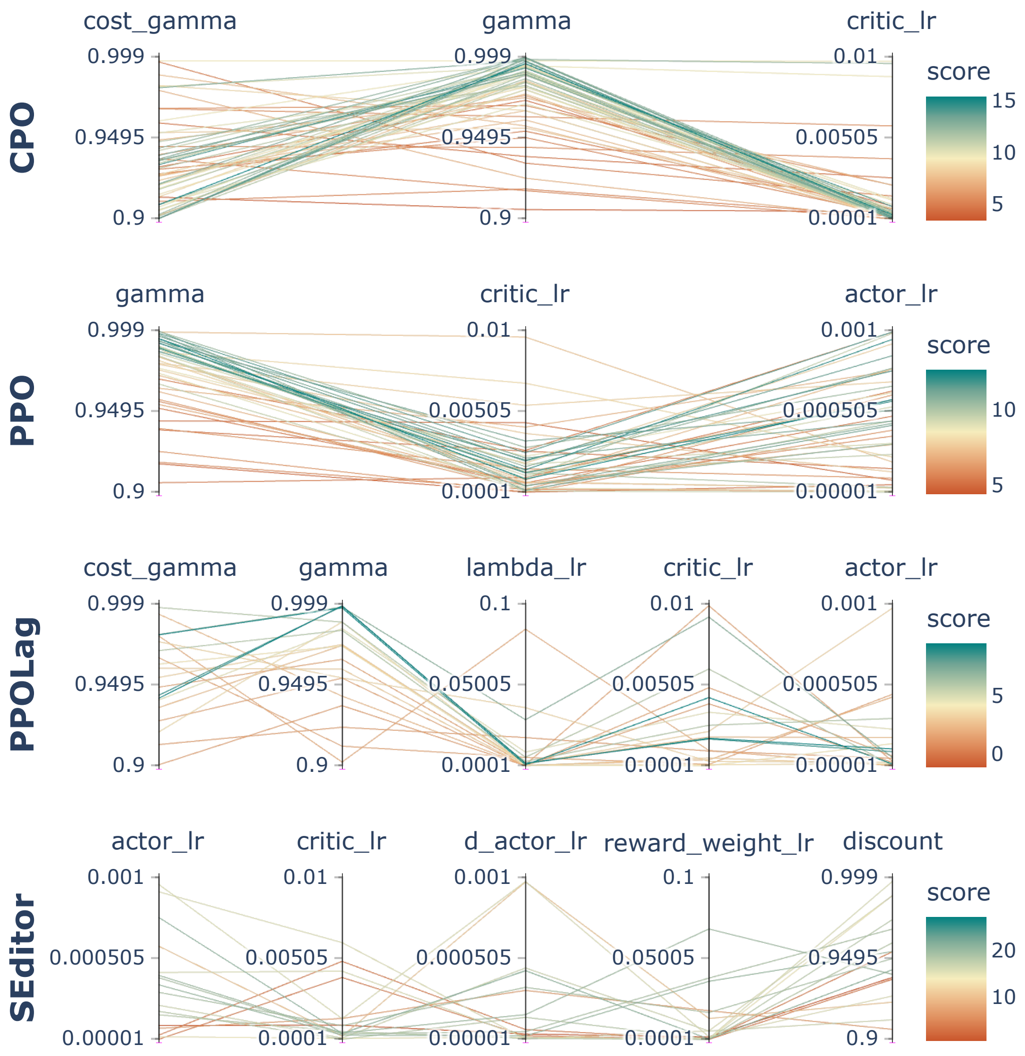


Figure 6.1.: Tuning results per algorithm in SafetyPointGoal1.

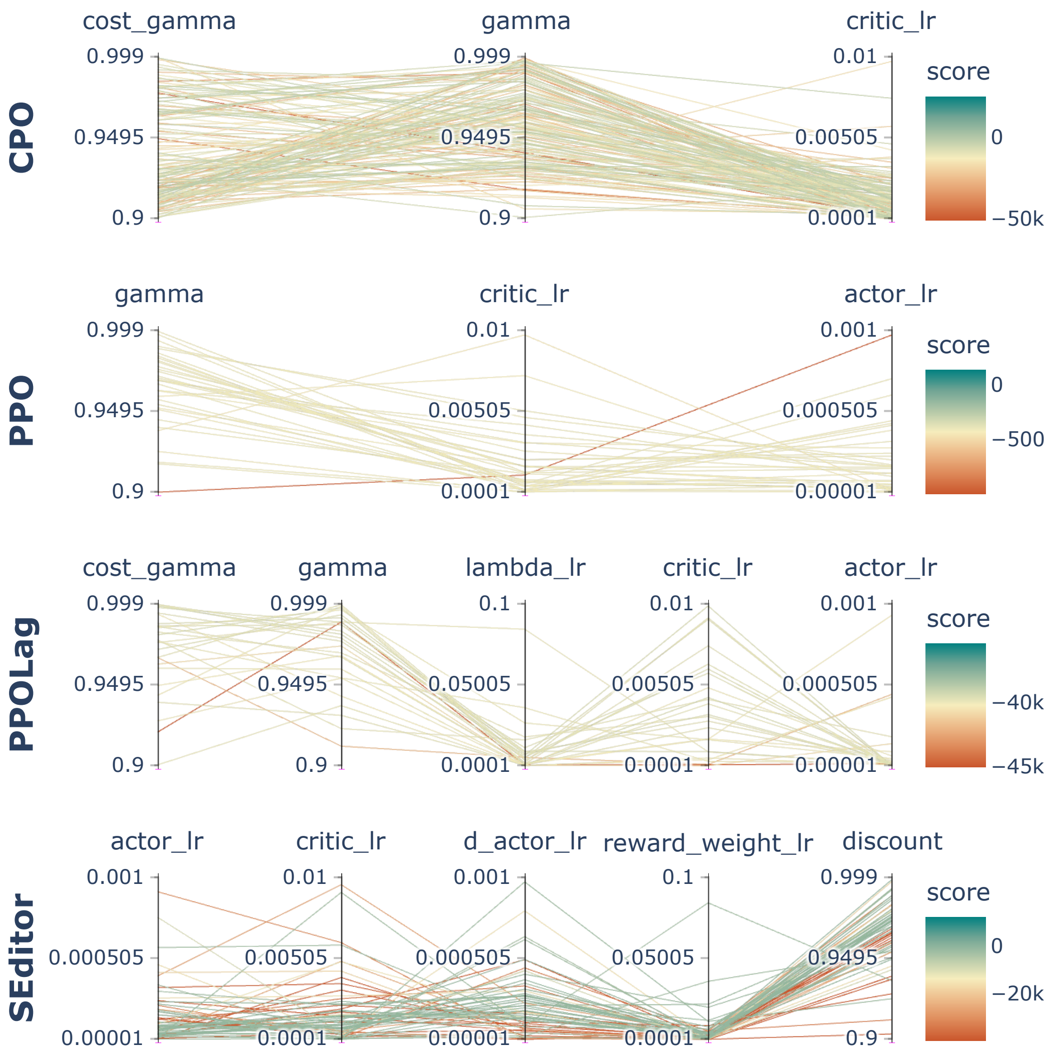


Figure 6.2.: Tuning results per algorithm in MidObstacle.

7. Performance Evaluation

7.1 Evaluation Setup

Using the results from Chapter 6, we determined the best hyperparameters for each algorithm and environment as the configuration that achieved the highest average over the final 10 scores. We fixed these hyperparameters and trained each algorithm for 10 million steps in the SafeNavigation tasks and for 5 million steps in CarRacing, MidObstacle, and Road2D on five different seeds: 200 to 204. Apart from tuning-related hyperparameters, all other settings remained consistent with those used in the tuning runs. We set the same cost limit χ and terminated an episode whenever a nonzero cost was recorded. Note that SEditor requires a definition of a per-step cost limit, which can be computed via Equation 6.2. A more thorough overview of all hyperparameters is provided in Appendix B.
Given our setup, we can interpret χ as a probability of crashing. Though a 40% probability in the SafeNavigation tasks is still high and not safe in the general sense, we chose this specific threshold to be consistent with other works [26]. Recall that the SafeNavigation tasks have an episode length of 1000 steps, which is also higher than for most other safety-oriented environments. Given this challenging setup, we consider our choice of χ a reasonable starting point. To complement our analysis, we also investigated lower values of χ in the SafetyPointGoal1 environment.

7.2 Results

In our plots 7.1, 7.2, 7.3, 7.4, we report the mean performance and one standard deviation ($\pm$) across the five different seeds for both objectives J and J^C. Note that we couldn't evaluate SPICE and SPICE-CSC in any of the SafeNavigation environments due to limitations of Py-Earth [1], which handles the environment model learning. The problem here is that the *Multivariate Adaptive Regression Splines* (MARS) used by Py-Earth struggle to fit splines at the observation space boundary. This issue is especially noticeable in the SafeNavigation domain, in which the lidar of the agent produces normalized observations in the interval $[0, 1)$ for each type of object. Values converge to 1 when an object is close to the agent and to 0 when it is far away, and a value of exactly 0 is used to inform the agent that the natural lidar ray did not intersect with an object in that specific direction. Unless the environment is packed with objects, the majority of the observations will report a value of 0. SPICE preemptively filters out such observations, which leads to the unfortunate situation of all observations being

[1] https://github.com/scikit-learn-contrib/py-earth

removed from the dataset. We disabled the filter and experimented with alternative approaches, but none proved successful. For example, we added a small amount of Gaussian noise to observations with a value of 0 and increased the regularization of the MARS model. Though in some cases the model training remained stable for a longer period of time, it did not consistently converge, and the overall training was eventually aborted. In the end, this limits the applicability of SPICE to certain environments. As a consequence, we were forced to exclude SPICE and SPICE-CSC from our evaluation on the SafetyGymnasium benchmark.

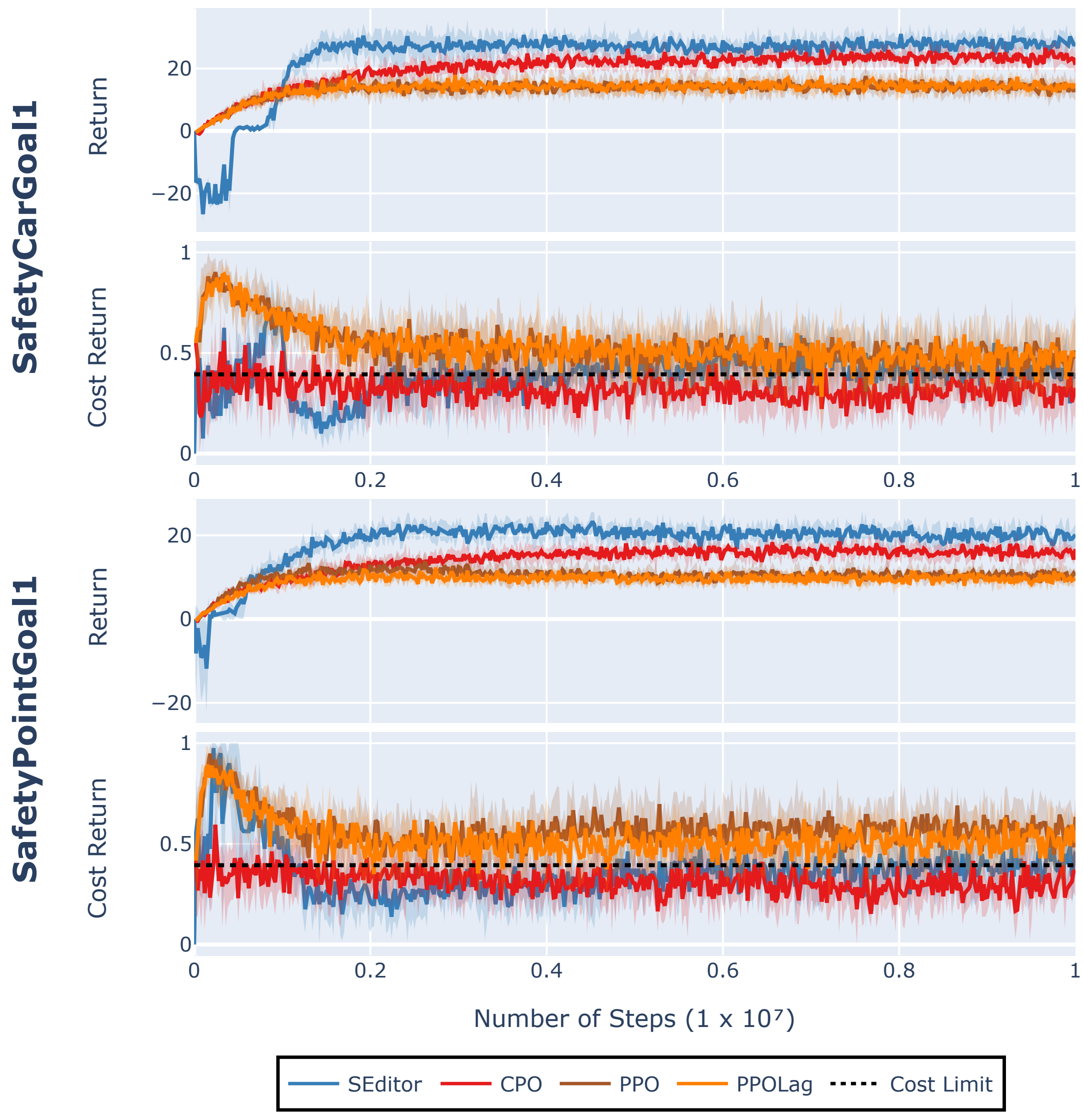


Figure 7.1.: Performance per algorithm in two of the SafeNavigation tasks averaged across 5 seeds. Objective J at the top and objective J^C at the bottom.

7.2.1 SafeNavigation

The results for the SafeNavigation tasks are shown in Figures 7.1, 7.2, with objective J at the top and objective J^C at the bottom of each environment plot.
Looking at the return, we see that SEditor frequently achieves a 20 to 30% higher return than the other algorithms in the long run, while at the same time oscillating closely around the cost limit χ. In SafetyPointGoal2, the return is almost three times as high. On one hand, this can be attributed to differences in hyperparameters and framework settings between these algorithms. For example, SEditor's policy and critic neural networks have a larger number of parameters compared to the other algorithms. Given sufficient training data, this can prove beneficial in large domains with complex structures. However, we deliberately chose not to equalize all hyperparameters, since this could have had a negative impact as well, and instead used the default settings for each algorithm. On the other hand, SEditor's multi-policy design could also simply be more effective in achieving both safety and performance compared to a single-policy approach like CPO or PPO. Though in the early stages of training, we also see SEditor's metrics fluctuate and underperform the other algorithms, especially CPO. The reason for this is likely twofold: the fact that CPO and PPO take small steps to learn the optimal policy, while SEditor has no restrictions in its updates; and the increase in the number of parameters for SEditor, which makes early predictions less accurate.

CPO achieves the second-best performance in the SafeNavigation tasks with respect to the return J, and it performs best on the cost objective J^C, especially early on. It quickly reaches the desired level of safety and then continuously improves the objective J. This behavior is likely a consequence of the improvement guarantees that CPO provides and the small steps it takes on its way of learning an optimal policy.
Comparing PPO and PPOLag, we find that PPO actually performs better or at least as well as PPOLag. This is surprising given that PPO does not even take safety into account, while PPOLag does. Their difference becomes more pronounced in SafetyPointGoal2, where PPOLag completely fails to learn safe behavior, while PPO's overall performance is consistent with CPO in the long run. In general, the performance of PPO seems rather counterintuitive. We included PPO as a baseline solely for the return J, but compared to the other algorithms, it neither improves upon the objective J nor ends up with a significantly higher value of J^C. One possible cause could be our choice to terminate episodes whenever any unsafe behavior is executed. Given this restriction, PPO does not gain an advantage when it exhibits unsafe but potentially rewarding behavior.
Another reason could be our choice of $\chi = 0.394$, which might simply be too loose, such that the goal-directed reward signal is sufficient for PPO to learn a well-performing policy that also adheres to the cost limit. We investigated this phenomenon by retraining CPO, PPOLag, and SEditor in SafetyPointGoal1 with different values for χ, namely 0.25, 0.1, and 0.01. We did not tune any hyperparameters but instead used the same ones from the original experiments where χ was set to 0.394. Refer to Figure 7.3 for the results of this experiment. The trace of PPO is also included for comparison.

The first thing to note is that neither PPO nor PPOLag is able to learn safe behavior that complies with lower cost limits χ. Both show the same performance in all four experiments in SafetyPointGoal1 regardless of the value of χ and average around $J^C = 0.5$ in the later stages of training. This illustrates that using a simple Lagrangian to combine both objectives of the CMDP has its limitations when it comes to ensuring safety.

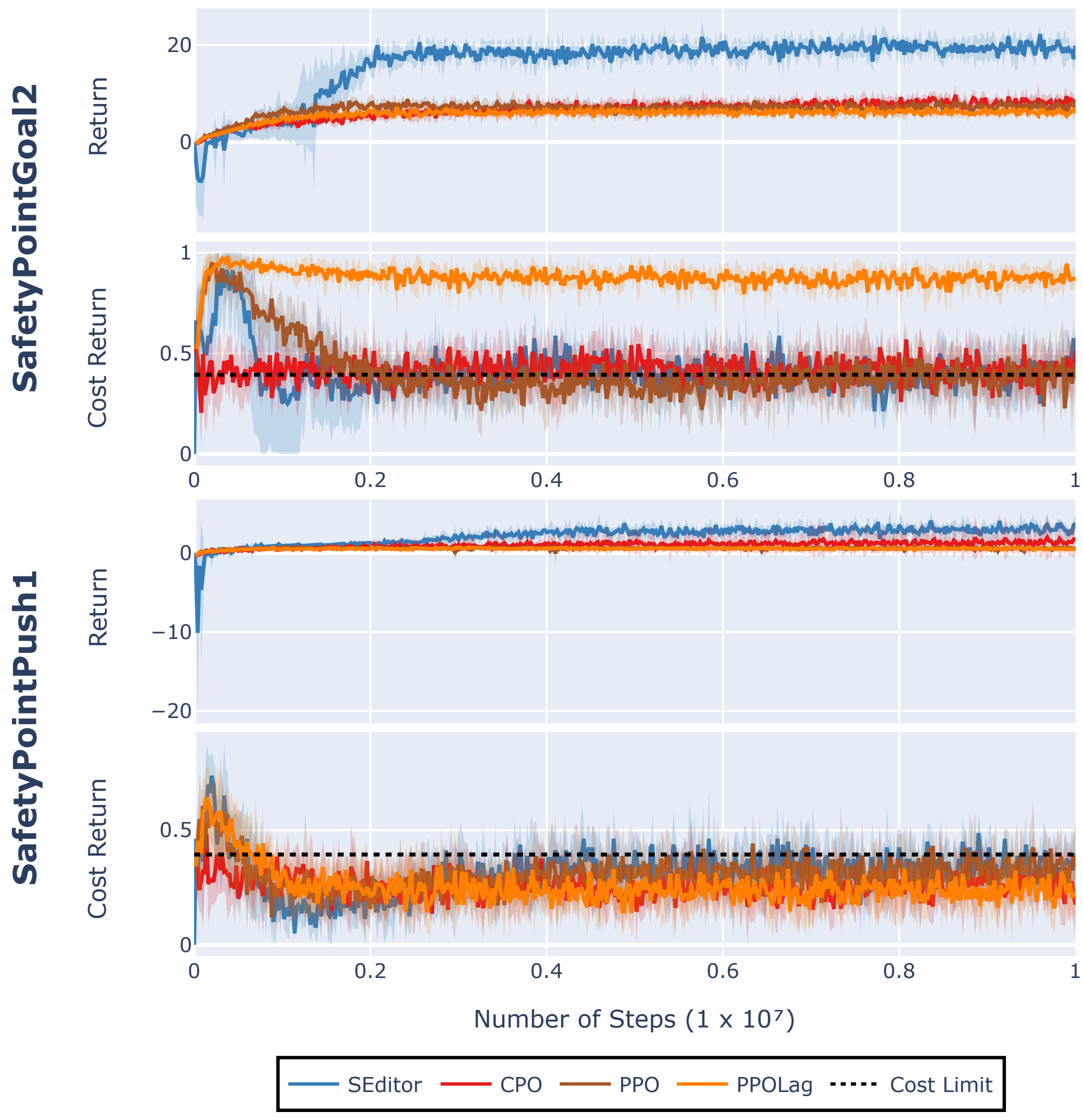


Figure 7.2.: Performance per algorithm in two of the SafeNavigation tasks averaged across 5 seeds. Objective J at the top and objective J^C at the bottom.

If we take a look at CPO and SEditor, we see that both algorithms are able to adhere to the cost limit of 0.25, with CPO even satisfying the cost limit of 0.1. However, neither algorithm is able to solve the task with χ set to 0.01. This highlights the overall difficulty of learning a policy that is both safe with high probability and performant over a longer period of time. When it comes to the objective J, SEditor's performance remains consistent across all three experiments, while CPO's performance drops as the cost limit becomes more strict. This can be attributed to the fact that CPO first tries to satisfy the safety constraint and treats the reward signal as a secondary objective, whereas SEditor tackles both objectives simultaneously without providing any guarantees.

7.2.2 Static Environments

Our second evaluation was performed on three smaller, static environments that also appeared in the evaluation of SPICE. We focus on CarRacing, MidObstacle, and Road2D, and chose a lower cost limit of $\chi = 0.01$ for all three tasks, which we chose considering the lack of randomness. Our results, which now also include the algorithms SPICE and SPICE-CSC, are shown in Figure 7.4.

Focusing on Road2D, we see that all algorithms are able to learn a safe policy that never violates the cost constraint. Compared to all other tasks, in Road2D we employ a speed limit that the agent must not exceed, instead of having a static obstacle in its path. Given that all algorithms are able to adhere to the speed limit, we assume that this is a simpler constraint to satisfy than the static obstacle. Looking at the return J, SPICE and SPICE-CSC perform the worst, while CPO, PPO, and PPOLag perform the best. SPICE and SPICE-CSC also show the largest variance in their performance, which can be explained by the fact that these algorithms learn a potentially inaccurate environment model and incorporate it into their training process. Furthermore, SEditor finds an optimal policy quickly but is unable to sustain consistent performance in the later stages of training.
In MidObstacle, both PPO and PPOLag quickly reach the maximum cost return of 1. Though their return is high, this is an artifact of the environment's design: instead of reaching the goal area, both algorithms learn to quickly crash into the obstacle. Since the agent receives a negative reward based on the distance to the goal area, short and unsafe episodes can result in a higher return than longer, potentially safe episodes. Besides that, we see a similar trend as before, with SPICE and SPICE-CSC showing almost the same performance, but overall worse than CPO and SEditor. Both SEditor and CPO learn a policy that is safe and performant, but CPO does so with more consistency in the value of J^C throughout training.
Lastly, in CarRacing, the most difficult of these three tasks, we see that every algorithm struggles to learn a safe policy except SPICE. Though SPICE learns safe behavior, it does not manage to solve the task and instead only learns to keep distance from the obstacle, hence the low value of J. Manual inspection revealed that none of the algorithms were able to reach the goal area. SEditor and CPO only step into the area of the first checkpoint but never complete the full task. While PPO shows the exact same behavior as in MidObstacle, PPOLag manages to slightly improve its safety behavior. Although the mean value of J^C decreased for PPOLag, its variance greatly increased, which shows its inability to consistently learn safe behavior.

Env	CPO	PPO	PPOLag	SEditor	SPICE	SPICE-CSC
SCG1	**3078 ± 82**	6152 ± 1063	6134 ± 924	4905 ± 50	-	-
SPG1	**2992 ± 26**	6638 ± 933	5873 ± 451	4699 ± 43	-	-
SPG2	**4357 ± 96**	4720 ± 553	16326 ± 939	5482 ± 344	-	-
SPP1	**2604 ± 450**	3234 ± 680	2637 ± 477	4328 ± 350	-	-
CR	1439 ± 557	60529 ± 826	31280 ± 20232	778 ± 161	**47 ± 29**	353 ± 103
MO	1140 ± 620	61421 ± 485	61343 ± 495	2056 ± 377	**29 ± 16**	383 ± 217
R2D	**0 ± 0**	**0 ± 0**	**0 ± 0**	**0 ± 0**	**0 ± 0**	**0 ± 0**

Table 7.1.: Mean and one standard deviation (±) of the cumulative safety violations during training across 5 seeds. Lowest number of violations per environment highlighted in bold. Environment names abbreviated following Section 5.2.

7.2.3 Safe Exploration

In summary, we have seen that CPO and SEditor learn a policy that is both safe and performant. CPO is more effective at ensuring safety, while SEditor can learn better-performing policies that also have a high probability of being safe. PPO and PPOLag are less effective at ensuring safety, but can still converge to policies with a reasonable return. Furthermore, SPICE and SPICE-CSC can provide a high level of safety, but at the cost of a lower value of J.
However, in certain environments, we not only want to learn a safe policy, but we also need to ensure that the learning process happens in a safe way. Hence, the overall number of safety violations should ideally be zero throughout training. To this end, we also monitored the cumulative number of safety violations over the course of training, the mean and one standard deviation (±) of which are shown in Table 7.1.

Focusing on the four SafeNavigation tasks, the same trend continues: CPO has the overall lowest number and smallest variance in the number of safety violations, while either PPO or PPOLag manage the highest. The latter can be attributed to their general inability to learn safe behavior. The only exception here is the SafetyPointPush1 domain, where SEditor shows the worst result. Looking back at Figure 7.2, this is likely the result of PPO, PPOLag, and CPO converging to policies that are overly cautious with respect to our chosen cost limit.
In the other three tasks, SPICE and SPICE-CSC clearly dominate the field. The reason is that both algorithms are able to utilize their learned environment model for gathering additional data and, as such, require fewer interactions with the true environment. But, as we have seen earlier, this also comes at the cost of a lower value for the return J. The remaining results follow the trend seen in Figure 7.4: CPO and SEditor both remain reasonably safe throughout training, while PPO and PPOLag frequently crash into the obstacle. Interestingly, all algorithms have zero violations in the Road2D environment. We assume that the default speed limit is set too high and the algorithms reach the goal area before exceeding the speed limit on the direct path to the goal.

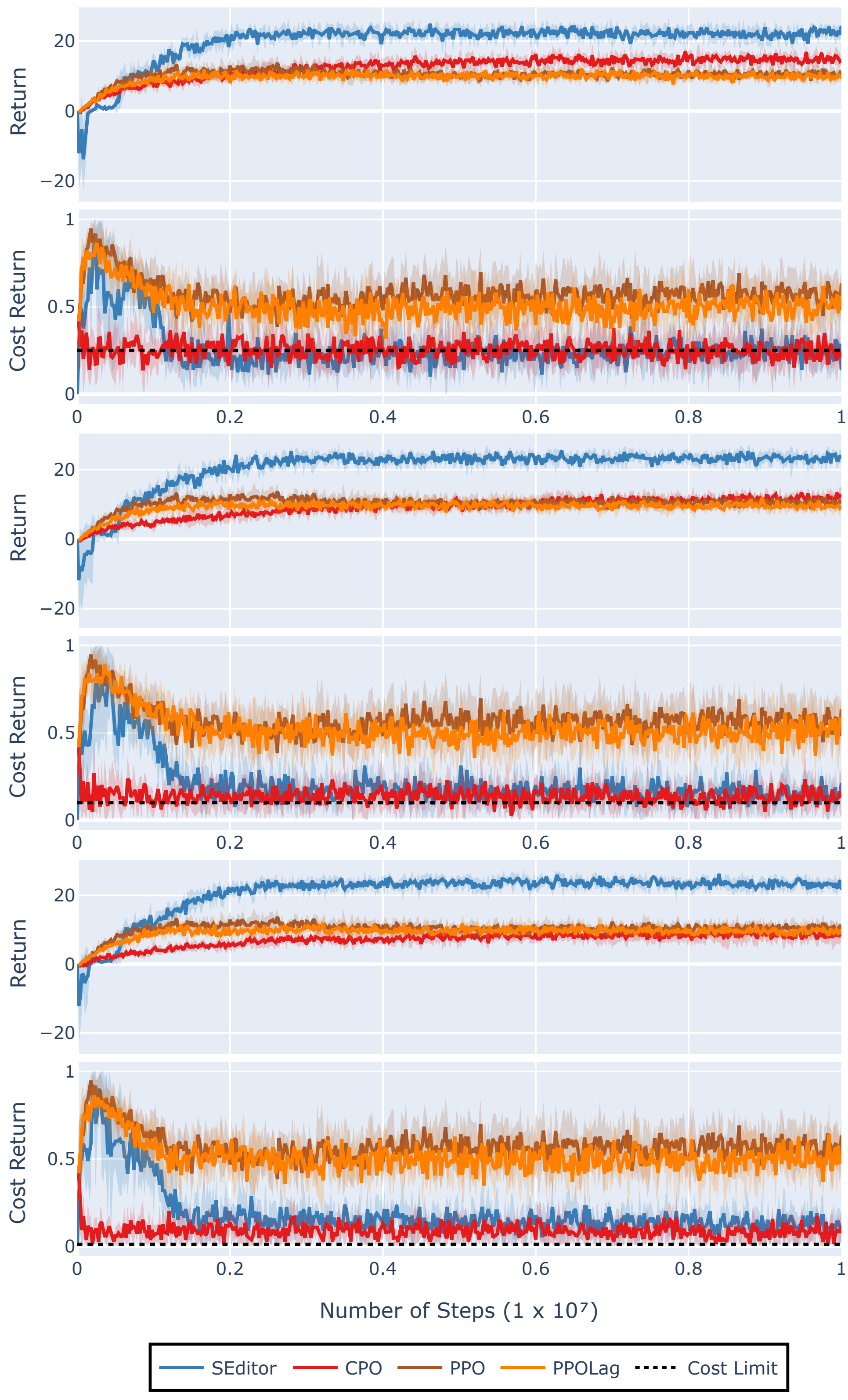


Figure 7.3.: Performance per algorithm for different cost limits in SafetyPointGoal1, averaged across 5 seeds. Top: $\chi = 0.25$, middle: $\chi = 0.1$, bottom: $\chi = 0.01$.

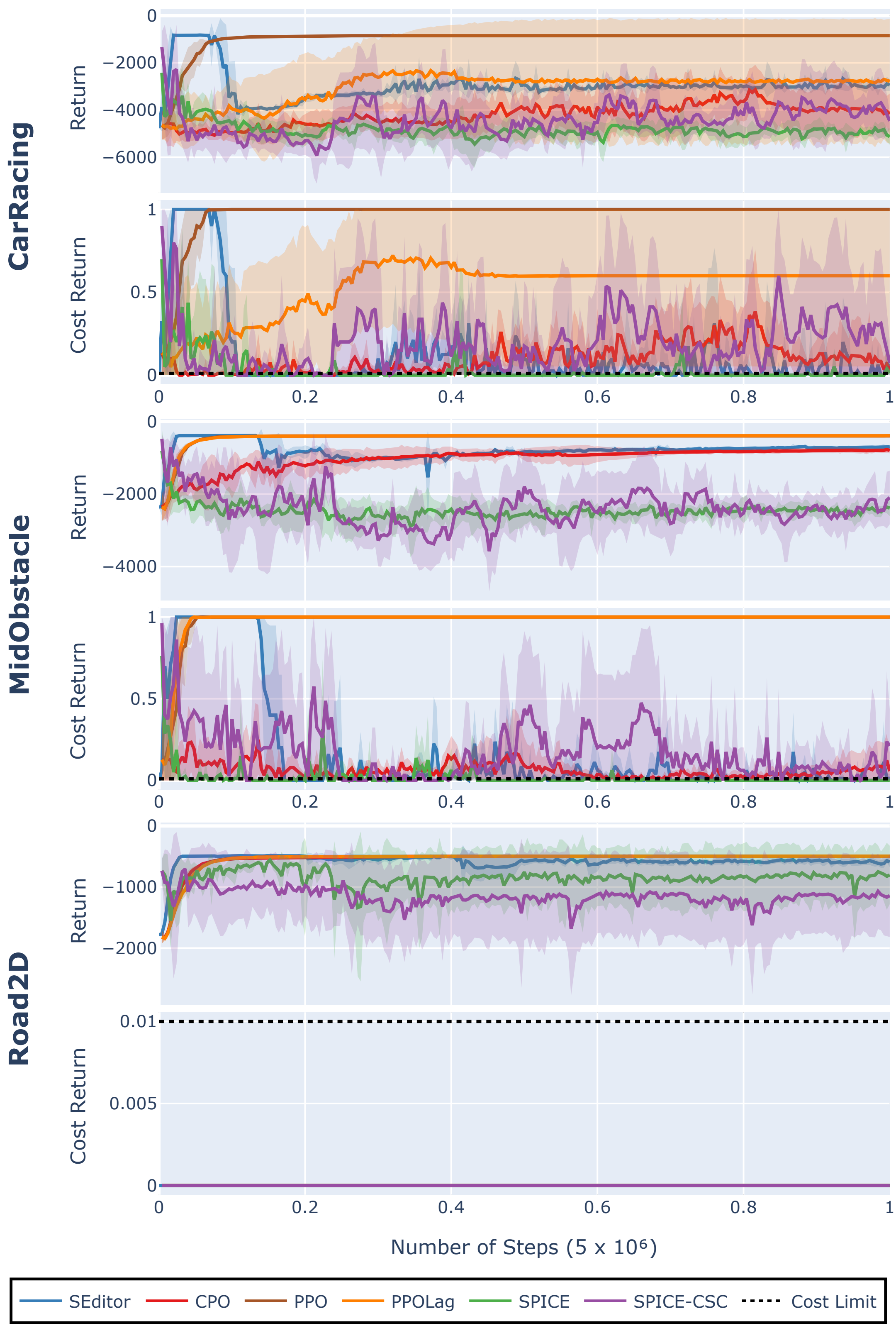


Figure 7.4.: Performance per algorithm in CarRacing, MidObstacle and Road2D averaged across 5 seeds. Objective J at the top and objective J^C at the bottom.

8. Statistical Guarantees

As the final part of this thesis, we aim to derive statistical guarantees for the metrics discussed so far. To that end, we utilize a tool called PyDSMC [1], which is based on *Deep Statistical Model Checking* (DSMC) [39]. PyDSMC can be used to approximate confidence intervals for a variety of user-defined metrics, including but not limited to the return J and the cost return J^C.

8.1 PyDSMC Setup and Metrics

PyDSMC requires three components: an environment, a policy from which to sample actions, and a set of metrics to be analyzed. In the previous chapter, we discussed the performance of each algorithm by looking at their averages over five distinct seeds. For the analysis with PyDSMC, we picked only the best-performing seed of each algorithm per environment. We ranked seeds by averaging over the final 10 scores J^S (as defined in Equation 6.1) and selected the one with the highest average score. This seed was fixed for the PyDSMC run and also served as the checkpoint containing the agent's policy and critics. As before, the cost limit χ was set to 0.394 or 0.01, depending on the environment. Finally, we selected five metrics to analyze, which are

- Undiscounted Return J,
- Undiscounted Cost Return J^C,
- Score J^S,
- Episode Length, and
- Number of Completed Tasks.

For hyperparameter tuning of PPO, the score J^S was set equal to J, given that PPO should only take into account the return. For the analysis of PPO with PyDSMC, we define J^S following Equation 6.1, i.e., the score presented here also takes into account the cost return J^C. This way, we obtain an estimate of the overall performance of PPO. Furthermore, the estimated interval for the return J is a valid approximation of the score J^S used in the tuning runs.

[1]https://github.com/neuro-mechanistic-modeling/PyDSMC

PyDSMC calculates a confidence interval $[a, b]$ for the true value μ of each metric such that

$$P(\mu \in [a, b]) \geq 1 - \kappa. \tag{8.1}$$

Given that we allow relative errors for each estimate, we have a symmetric confidence interval such that $(1 - \epsilon) \times \hat{\mu} \leq a \leq b \leq (1 + \epsilon) \times \hat{\mu}$, where $\hat{\mu}$ is the estimate of the true value μ. ϵ defines the half-width of the confidence interval, while κ defines the confidence level. We set them to 0.025 and 0.05, respectively. To derive the confidence interval, we did not manually fix a specific statistical method but instead let PyDSMC choose the best method that conforms to the properties of each metric. Depending on the choice of algorithm, environment, and the values of ϵ and κ, approximating each confidence interval can require a significant number of trajectories. Consequently, we enforce a time limit of 6 hours for PyDSMC, after which it will terminate the analysis and return the latest estimate of the confidence interval.

8.2 PyDSMC Results

Our main results are presented in Figure 8.1. As there is a major difference in the reward structure between the two sets of environments, we chose to separate the results into two columns for better readability: one for the four SafeNavigation tasks and one for CarRacing, MidObstacle, and Road2D.

In SafetyCarGoal1, our results match those seen in Figure 7.1. SEditor achieves the highest return, while CPO has the lowest cost return J^C, even below the cost limit χ. PPO and PPOLag show similar performance, but both perform significantly worse than SEditor and CPO.
In SafetyPointGoal1, things are a bit different. Despite observing the same tendencies for the return J, the results for the cost return J^C are quite different. In the respective evaluation runs, CPO showed the best safety behavior, while PPO and PPOLag were both unsafe to the same extent. In contrast, the analysis with PyDSMC shows that PPO is significantly worse than the three other algorithms, and SEditor performs slightly better than CPO with respect to costs. Furthermore, in SafetyPointGoal2, CPO also runs at a higher cost than in its evaluation runs in Figure 7.2.
There are multiple reasons for this. First, we analyzed only a single checkpoint with PyDSMC, whereas we averaged over five different seeds in our evaluation runs. Thus, the results shown here are checkpoint-dependent and, as such, can diverge from the mean results in the evaluation runs. Second, for this analysis, we set the policy of the agent into evaluation mode, i.e., it produces deterministic actions. In the evaluation runs, the on-policy algorithms (CPO, PPO, and PPOLag) used stochastic actions, which is why the intervals can slightly differ.
Besides that, we observe that the number of completed tasks correlates with the return J, while there is a negative correlation between the cost return J^C and the episode length. The reason is that the agent receives a reward of 1 whenever it reaches a goal, and we terminate episodes whenever unsafe behavior occurs. Furthermore, the score J^S

shows the same tendencies as the return J, which is not surprising given that it is a scaled version of J.

As was the case for the evaluation runs 7.4, in CarRacing none of the algorithms were able to reach the goal area. However, in the analysis with PyDSMC, we find that PPOLag is indeed able to guarantee safety in this task. We suspect that this specific policy was able to learn safe behavior, while the majority of other seeds failed to do so. This assumption is based on the fact that in the evaluation runs for CarRacing, PPOLag showed significant variance across the five seeds in the cost return J^C. Nonetheless, this goes to show that PPOLag can be used to learn a safe policy for this task. Furthermore, the results of the other algorithms are consistent with those seen in the evaluation runs: PPO learns to crash into the obstacle as fast as possible, while CPO, SEditor, and both versions of SPICE manage to converge to a safe policy that unfortunately isn't able to solve the task.
In MidObstacle, the results of the statistical evaluation also agree with the performance evaluation. According to the approximations obtained by PyDSMC, PPO and PPOLag are both unsafe, while the other algorithms are safe. From the set of safe algorithms, CPO and SEditor perform equally well with respect to the return J. This is surprising given that SEditor does not manage to reach the goal, i.e., SEditor doesn't finish the task, but achieves the same return as CPO while averaging double the steps per episode in an environment with a negative per-step reward. Further investigation revealed that SEditor learns to efficiently circumvent the obstacle but targets an area near the goal instead of the goal itself. We provide an illustration of the different behaviors in Appendix D. Our suspicion is that during training, SEditor got stuck in some local minimum and was unable to escape it, the reason being an interplay between the reward structure of the environment and the reward normalization of SEditor. MidObstacle has no additional reward for reaching the goal area but only a smooth reward that decreases linearly based on the distance to a fixed goal position. Hence, there is no strong incentive to go all the way to the goal. Combined with the fact that the normalization of SEditor squashes the overall magnitude of the rewards, we get the aforementioned behavior. On the other hand, CPO doesn't suffer the same problem and consistently reaches the goal on a suboptimal, more conservative path.
In Road2D, we observe the exact same problem. Although the metrics are consistent with those seen in the evaluation runs, the number of times the agent is able to reach the goal area for SEditor is significantly lower than for the other algorithms. We again manually investigated this phenomenon and come to the same conclusion as for MidObstacle: SEditor likely got stuck in some local minimum.
As a final point, recall that in the tuning runs of PPO C.2, the score J^S appeared high compared to other algorithms, but we argued that this was misleading as it did not account for the safety objective J^C, and PPO did not solve the task. In this analysis, we now observe that the safety-aware score J^S for PPO is typically the lowest among all algorithms. This is not surprising, given that PPO does not prioritize safety, and we use the value of J^C as a scaling factor in the calculation of J^S.

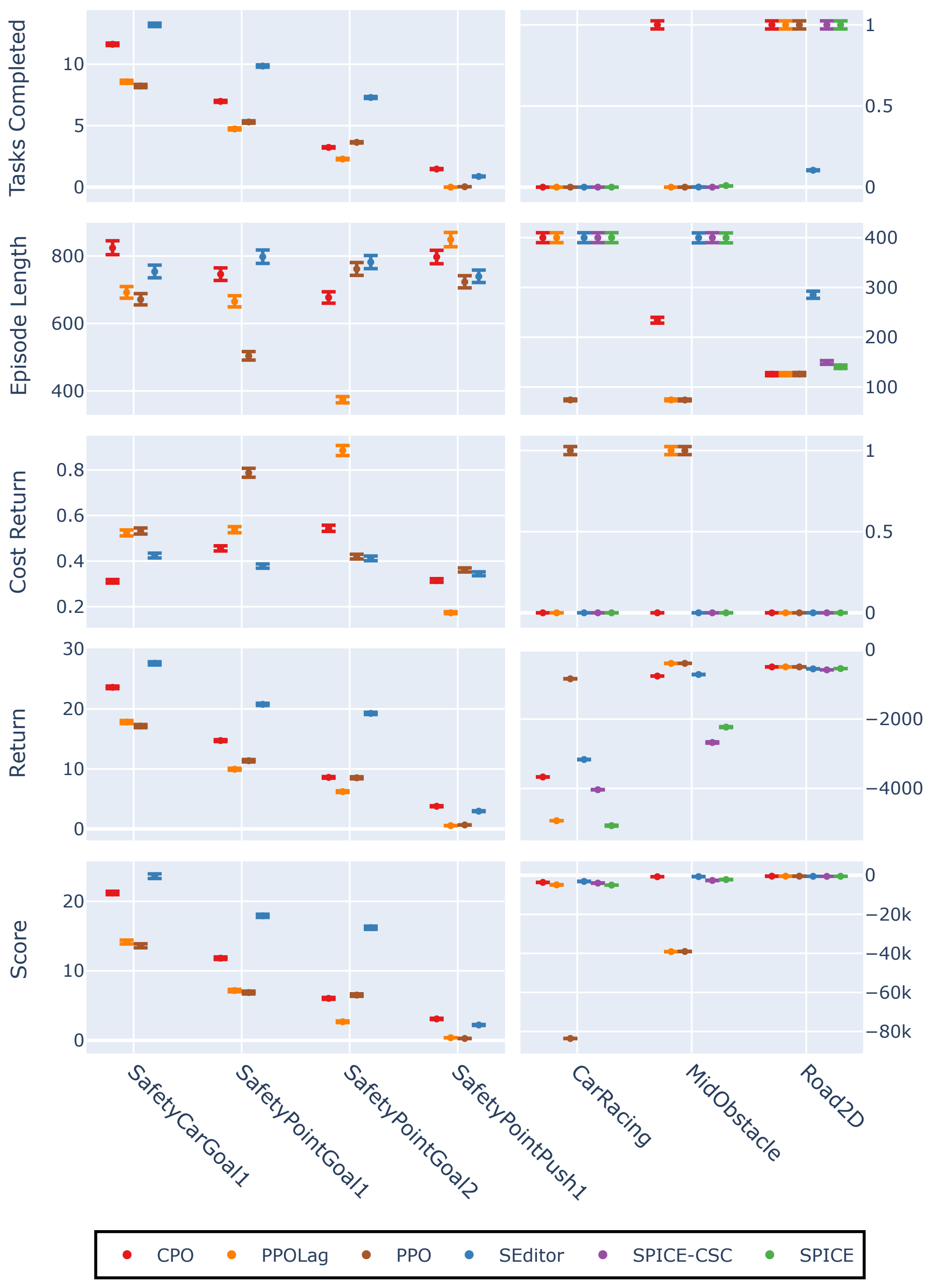


Figure 8.1.: PyDSMC confidence intervals for every metric, algorithm and environment.

9. Conclusion

9.1 Contributions

In this thesis, we have provided an overview of existing SRL algorithms and compared popular constrained optimization and shielded approaches on state-of-the-art benchmarks. We first presented the theoretical foundations of SRL and built a classification of the most prominent approaches. There, we emphasized the challenges of ensuring safety in RL given the fundamental idea of learning via trial and error. As such, we've seen that algorithms that allow an agent to remain completely safe require certain knowledge about the environment and the task at hand. Since we made the assumption of a black-box environment, we restricted our analysis to a smaller subset of algorithms that only provide probabilistic safety guarantees. In the group of constrained optimization algorithms, we used PPOLag and CPO, and for shielded methods we considered SEditor, SPICE, and SPICE-CSC.
Our overall evaluation was split into three parts: hyperparameter tuning, performance evaluation, and statistical analysis. We first demonstrated the importance of hyperparameter tuning for RL and drew connections between optimal hyperparameters, algorithm structure, and environment layout. Our performance evaluation was carried out in a diverse set of environments, including the challenging SafeNavigation tasks from SafetyGymnasium. Here, we discovered the difficulties in ensuring a high level of safety at adequate task performance and found that a straightforward approach like PPOLag is not sufficient. Instead, more elaborate algorithms like CPO and SEditor dominated the field. Overall, the final policy learned by CPO achieved the highest level of safety with little to no variance, and SEditor showed the most efficient behavior with reasonably few safety violations. We also hinted at the importance of safe exploration and found that learned environment models can greatly benefit the agent's safety throughout training. However, in the context of SPICE, the environment model turned out to be a limiting factor in terms of overall applicability. As a final part, we used PyDSMC for statistical analysis and confirmed our prior findings.

Considering our initial motivation of comparing constrained optimization to shielding methods, we come to the conclusion that there is no obvious winner for solving CMDPs. Shielding can certainly provide great benefits, e.g., in the case of SEditor, the interplay of both policies led to very high task performance at a reasonable safety level. Or, in the case of SPICE, the learned model greatly benefited safety during training. On the other hand, constrained optimization approaches like CPO showed steady improvement on both objectives and also resulted in safe behavior. Hence, the optimal choice of method depends on the task itself, our knowledge of and ability to model the environment, and the guarantees we aim to achieve.

9.2 Limitations and Future Work

Our classification only provides a coarse overview of SRL algorithms. Every algorithm makes different assumptions, and our current classification doesn't reflect all of them. For example, CautiousRL [20] assumes limited observability of nearby states, while SMBPO [10] assumes that a safety violation will always occur in a finite number of steps. To gain a deeper understanding of these algorithms, a more fine-grained study is needed.
Furthermore, our hyperparameter tuning only considers a small subset of all possible hyperparameters. Tuning other, potentially more promising hyperparameters could lead to different results. Though we believe that the overall trends presented here should remain valid. Additionally, our tuning results—and consequently all other results—heavily depend on the design of the score function. Although we carefully designed the score function to only punish the agent when the safety level exceeds the cost limit χ, we have no guarantees that this is optimal. It might be worthwhile to design the score function on a more fine-grained level, e.g., separately for each environment.
Another important limitation is that our conclusions are based on an analysis of just a small subset of all available SRL algorithms. Other shielding methods like AMBS [17], CautiousRL [20], or SAILR [25] might outperform the ones we considered and are worth investigating in the future.
On a final note, it is also important to mention that we chose a relatively high cost limit for the SafeNavigation tasks. Though we did provide a small analysis for lower cost limits, it could be worthwhile to perform a more thorough analysis with limits close to or at zero. The reason being that, ultimately, the goal of SRL is to develop algorithms that are both effective at solving the task and do not exert any unsafe behavior. Hence, an extensive analysis with stricter cost limits could prove useful in uncovering further advantages and disadvantages of current SRL algorithms.

Bibliography

[1] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* USA: John Wiley & Sons, Inc., 1st ed., 1994.

[2] E. Altman, *Constrained Markov Decision Processes.* Chapman and Hall, 1999.

[3] A. Ray, J. Achiam, and D. Amodei, "Benchmarking Safe Exploration in Deep Reinforcement Learning," 2019.

[4] S. Paternain, M. Calvo-Fullana, L. F. O. Chamon, and A. Ribeiro, "Safe policies for reinforcement learning via primal-dual methods," Jan. 2022. arXiv:1911.09101 [cs, eess, math].

[5] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," May 2017. arXiv:1705.10528 [cs].

[6] A. Stooke, J. Achiam, and P. Abbeel, "Responsive safety in reinforcement learning by pid lagrangian methods," July 2020. arXiv:2007.03964 [math].

[7] Y. Liu, J. Ding, and X. Liu, "Ipo: Interior-point policy optimization under constraints," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, p. 4940–4947, Apr. 2020.

[8] A. Wachi, Y. Sui, Y. Yue, and M. Ono, "Safe exploration and optimization of constrained mdps using gaussian processes," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.

[9] T.-H. Pham, G. De Magistris, and R. Tachibana, "Optlayer - practical constrained optimization for deep reinforcement learning in the real world," Feb. 2018. arXiv:1709.07643 [cs].

[10] G. Thomas, Y. Luo, and T. Ma, "Safe reinforcement learning by imagining the near future," in *Advances in Neural Information Processing Systems*, vol. 34, p. 13859–13869, Curran Associates, Inc., 2021.

[11] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.

[12] S. Carr, N. Jansen, S. Junges, and U. Topcu, "Safe reinforcement learning via shielding under partial observability," Aug. 2022. arXiv:2204.00755 [cs].

[13] B. Könighofer, J. Rudolf, A. Palmisano, M. Tappler, and R. Bloem, "Online shielding for reinforcement learning," *Innovations in Systems and Software Engineering*, vol. 19, p. 379–394, Dec. 2023.

[14] N. Jansen, B. Könighofer, S. Junges, A. Serban, and R. Bloem, "Safe reinforcement learning using probabilistic shields (invited paper)," in *DROPS-IDN/v2/document/10.4230/LIPIcs.CONCUR.2020.3*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[15] W.-C. Yang, G. Marra, G. Rens, and L. De Raedt, "Safe reinforcement learning via probabilistic logic shields," Mar. 2023. arXiv:2303.03226 [cs].

[16] G. Anderson, A. Verma, I. Dillig, and S. Chaudhuri, "Neurosymbolic reinforcement learning with formally verified exploration," Oct. 2020. arXiv:2009.12612 [cs, stat].

[17] A. W. Goodall and F. Belardinelli, *Approximate Model-Based Shielding for Safe Reinforcement Learning.* Sept. 2023. arXiv:2308.00707 [cs, eess].

[18] A. W. Goodall and F. Belardinelli, "Leveraging approximate model-based shielding for probabilistic safety guarantees in continuous environments," Feb. 2024. arXiv:2402.00816 [cs].

[19] C. He, B. G. Leon, and F. Belardinelli, "Do androids dream of electric fences? safety-aware reinforcement learning with latent shielding," Dec. 2021. arXiv:2112.11490 [cs, eess].

[20] M. Hasanbeig, A. Abate, and D. Kroening, "Cautious reinforcement learning with logical constraints," Mar. 2020. arXiv:2002.12156 [cs, eess, stat].

[21] G. Anderson, S. Chaudhuri, and I. Dillig, "Guiding safe exploration with weakest preconditions," Feb. 2023. arXiv:2209.14148 [cs].

[22] H. Bharadhwaj, A. Kumar, N. Rhinehart, S. Levine, F. Shkurti, and A. Garg, "Conservative safety critics for exploration," Apr. 2021. arXiv:2010.14497 [cs, stat].

[23] B. Thananjeyan, A. Balakrishna, S. Nair, M. Luo, K. Srinivasan, M. Hwang, J. E. Gonzalez, J. Ibarz, C. Finn, and K. Goldberg, "Recovery rl: Safe reinforcement learning with learned recovery zones," *IEEE Robotics and Automation Letters*, vol. 6, p. 4915–4922, July 2021.

[24] K.-C. Hsu, A. Z. Ren, D. P. Nguyen, A. Majumdar, and J. F. Fisac, "Sim-to-lab-to-real: Safe reinforcement learning with shielding and generalization guarantees," *Artificial Intelligence*, vol. 314, p. 103811, Jan. 2023.

[25] N. C. Wagener, B. Boots, and C.-A. Cheng, "Safe reinforcement learning using

advantage-based intervention," in *Proceedings of the 38th International Conference on Machine Learning*, p. 10630–10640, PMLR, July 2021.

[26] H. Yu, W. Xu, and H. Zhang, "Towards safe reinforcement learning with a safety editor policy," *Advances in Neural Information Processing Systems*, vol. 35, p. 2608–2621, Dec. 2022.

[27] D. Bethell, S. Gerasimou, R. Calinescu, and C. Imrie, "Safe reinforcement learning in black-box environments via adaptive shielding," May 2024. arXiv:2405.18180 [cs].

[28] O. Bastani, "Safe reinforcement learning with nonlinear dynamics via model predictive shielding," in *2021 American Control Conference (ACC)*, p. 3488–3494, May 2021.

[29] A. Banerjee, K. Rahmani, J. Biswas, and I. Dillig, "Dynamic model predictive shielding for provably safe reinforcement learning," May 2024. arXiv:2405.13863 [cs].

[30] A. Politowicz, S. Mazumder, and B. Liu, "Safety through permissibility: Shield construction for fast and safe reinforcement learning," May 2024. arXiv:2405.19414 [cs].

[31] S. S. Shperberg, B. Liu, and P. Stone, "Learning a shield from catastrophic action effects: Never repeat the same mistake," Feb. 2022. arXiv:2202.09516 [cs].

[32] R. Mitta, H. Hasanbeig, J. Wang, D. Kroening, Y. Kantaros, and A. Abate, "Safeguarded progress in reinforcement learning: Safe bayesian exploration for control policy synthesis," Dec. 2023. arXiv:2312.11314 [cs, eess].

[33] Q. Yang, T. D. Simão, N. Jansen, S. H. Tindemans, and M. T. J. Spaan, "Reinforcement learning by guided safe exploration," July 2023.

[34] H. Hoang, T. Mai, and P. Varakantham, "Imitate the good and avoid the bad: An incremental approach to safe reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, p. 12439–12447, Mar. 2024.

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[36] J. Ji, B. Zhang, J. Zhou, X. Pan, W. Huang, R. Sun, Y. Geng, Y. Zhong, J. Dai, and Y. Yang, "Safety gymnasium: A unified safe reinforcement learning benchmark," in *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[37] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization,” Apr. 2017. arXiv:1502.05477 [cs].

[38] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631, 2019.

[39] T. P. Gros, H. Hermanns, J. Hoffmann, M. Klauck, and M. Steinmetz, “Deep statistical model checking,” in *Formal Techniques for Distributed Objects, Components, and Systems* (A. Gotsman and A. Sokolova, eds.), (Cham), pp. 96–114, Springer International Publishing, 2020.

[40] O. J. Achiam, S. Adler, S. Agarwal, *et al.*, “Gpt-4 technical report,” 2023.

Appendices

A. SafeNavigation Details

In this thesis, we slightly modified some of the original SafeNavigation tasks of Safety-Gymnasium. Similar changes were made in the evaluation of SEditor [26].

We equipped the agent with a fine-grained, natural lidar and slightly lowered the number of obstacles. The reasoning here is that the original 16-bin pseudo lidar does not reveal sufficient information about the shape of an object and therefore is not suited for scenarios that require a high level of safety. This change should also make these challenging environments more realistic, as the natural lidar is closer to the lidar systems found in modern self-driving cars. Furthermore, in the original SafetyGymnasium tasks, the agent is allowed a cost budget of 25 for each task, and the episode does not terminate whenever any unsafe behavior is detected. We terminate the episode whenever a cost of 1 is recorded and set a much lower cost limit. Hence, to make these tasks feasible for all algorithms, we chose to slightly lower the number of obstacles.

Table A.1 provides a summary of all the changes. Note that changes made to a lower-level task are also reflected in higher levels, unless overridden. For example, setting the velocity cost of vases to 0 in Goal1 also nullifies it in Goal2.

Task	**Attribute**	**Change**
All Tasks	Lidar Type	pseudo → natural
	Lidar Bins	16 → 64
Goal1	Hazards	8 → 6
	Vases Velocity Cost	1.0 → 0.0
	Constrain Vases	False → True
Goal2	Hazards	10 → 7
	Vases	10 → 7
Push1	Constrain Pillars	False → True

Table A.1.: Changes made to the original SafeNavigation tasks.

B. Hyperparameters

In this section, we provide a detailed overview of algorithm-specific settings used in all our experiments. Since we essentially have three different code bases—CPO, PPO, and PPOLag originate from OmniSafe; SEditor is implemented in ALF; and SPICE is a standalone implementation—the general hyperparameters as well as the capabilities of each algorithm differ substantially. For example, SPICE does not implement state normalization while OmniSafe does, and SEditor uses a beta distribution for the policy, while the other frameworks support Gaussian distributions. We decided against equalizing all hyperparameters across all algorithms, as this could have had a negative impact on overall performance, and instead used the default values most of the time. Refer to Table B.1 for a summary of the hyperparameters used in all our experiments. Tuned hyperparameters are marked with a "$*$", and their values can be found in Table C.1.

CPO, PPO, and PPOLag are on-policy algorithms, while SEditor and SPICE are off-policy. On-policy algorithms do not require a replay buffer or evaluation during training. Instead, these algorithms use the trajectories collected under the current policy to compute updates and report performance based on those trajectories. Off-policy algorithms perform updates more frequently and require a separate evaluation phase, as the policy continuously evolves throughout an episode. For a fair performance comparison, we set the evaluation interval of SEditor and SPICE to match the update interval of CPO, PPO, and PPOLag.
Though we equalize these two intervals, we cannot ensure that the number of evaluation episodes matches. In this work, we terminate an episode whenever a non-zero cost is recorded. As a consequence, the number of episodes that on-policy algorithms can perform between updates depends on their safety behavior, i.e., if the agent crashes frequently, the algorithm will run many short episodes. In comparison, the number of evaluation episodes for off-policy algorithms is fixed. However, this should only have a minor impact on the results, as it primarily affects the accuracy of our estimates for both objectives J and J^C.
Further note that SEditor uses two policies with the same network structure. Though we did not clarify this in the table, we tuned the learning rate of each policy individually, as shown in Figure C.4.

On a final note, we want to emphasize that Table B.1 is not a complete list of all hyperparameters, but rather a summary of the important ones. Hyperparameters not contained in this table can be found in the source code of the respective algorithms and were left unmodified.

Attribute	**CPO**	**PPO**	**PPOLag**	**SEditor**	**SPICE SPICE-CSC**
Number of Envs	16	←	←	←	1
Steps until Update	20,000	←	←	80	1
Collect Steps	0	←	←	10,000	←
Buffer Capacity	-	-	-	50,000	1,000,000
Consecutive Updates	10	20	←	1	←
Batch Size	128	←	←	1,024	256
Target KL	0.01	0.02	←	-	-
Reward Normalization	No	←	←	Yes	No
Cost Normalization	No	-	No	Yes	No
State Normalization	Yes	←	←	No	←
Gamma	$*$	$*$	$*$	$*$	0.99
Cost Gamma	$*$	-	$*$	$*$	0.99
Entropy Coefficient	0.0	←	←	1.0	0.2
Entropy LR	-	-	-	0.0003	←
Eval Interval	-	-	-	20,000	←
Eval Episodes	-	-	-	16	←
Policy Type	Gaussian	←	←	Beta	Gaussian
Actor Hidden Sizes	2x64	←	←	3x256	2x256
Actor Activation	Tanh	←	←	←	ReLU
Actor LR	-	$*$	$*$	$*$	0.0003
Critic Hidden Sizes	2x64	←	←	3x256	2x256
Critic Activation	Tanh	←	←	←	ReLU
Critic LR	$*$	$*$	$*$	$*$	0.0003
Target Critic LR	-	-	-	0.005	←
Number of Critics	1	←	←	2	←
Lagrange Value	-	-	0.001	1.0	-
Lagrange LR	-	-	$*$	$*$	-
Optimizer	Adam	←	←	←	←
Safety Horizon	-	-	-	-	5
Neural Threshold	-	-	-	-	0.1
Train Real Episodes	-	-	-	-	10
Train Sim Episodes	-	-	-	-	70
Model Update Interval	-	-	-	-	100

Table B.1.: Overview of hyperparameters and other settings used in all our experiments. "←" indicates that the algorithm uses the value to the left of that cell,"-" marks unused or unsupported hyperparameters and "$*$" is a placeholder for values which we tuned. These are listed in Table C.1.

C. Extended Tuning Results

Figures C.1, C.2, C.3, and C.4 on the following pages summarize all our tuning results. Refer to Chapter 6 for details regarding the setup of the tuning runs and a more elaborate discussion of SafetyPointGoal1 and MidObstacle. Table C.1 presents the best values for the tuned hyperparameters of each algorithm and environment. These hyperparameters were also used for the evaluation runs in Chapter 7 and the statistical analysis in Chapter 8.

Overall, we observe that all four algorithms share some common hyperparameter preferences. For instance, a low critic learning rate or a high gamma value typically leads to better performance in all seven environments. This behavior is clearly visible in the SafeNavigation tasks, while greater fluctuations occur in CarRacing, MidObstacle, and Road2D. Looking at CPO in Figure C.1, we see that the optimal cost discount factor varies by environment, which highlights the importance of environment-specific hyperparameter optimization. PPO C.2 and PPOLag C.3 show similar behavior, with actor learning rate and cost discount being the hyperparameters that fluctuate most across environments, while others remain fairly stable. Unfortunately, for SEditor C.4, we managed only around 15 tuning runs in each SafeNavigation task, making it difficult to draw robust conclusions. Nonetheless, based on the collected data, we observe a general tendency toward lower learning rates and higher discount values to achieve better scores.
As mentioned earlier in the tuning chapter, the scores in each environment reflect algorithm performance. In SafeNavigation tasks, the score correlates with the number of goals reached by the agent. In CarRacing, MidObstacle, and Road2D, higher scores indicate that the agent learns to navigate towards the goal — the higher the score, the faster the agent is traveling in the right direction. By comparing scores across environments, we see that PPOLag fails to learn a good policy in CarRacing and MidObstacle. This may be because PPOLag requires more than 25% of the total steps used for tuning to learn a performant, safe policy or because the algorithm is too restricted in its exploration. Though the score does not reflect this, PPO encounters the same issue. This is not too surprising, given that PPO and PPOLag are closely related to each other. Conversely, CPO and SEditor achieve scores that suggest task completion in every single environment.

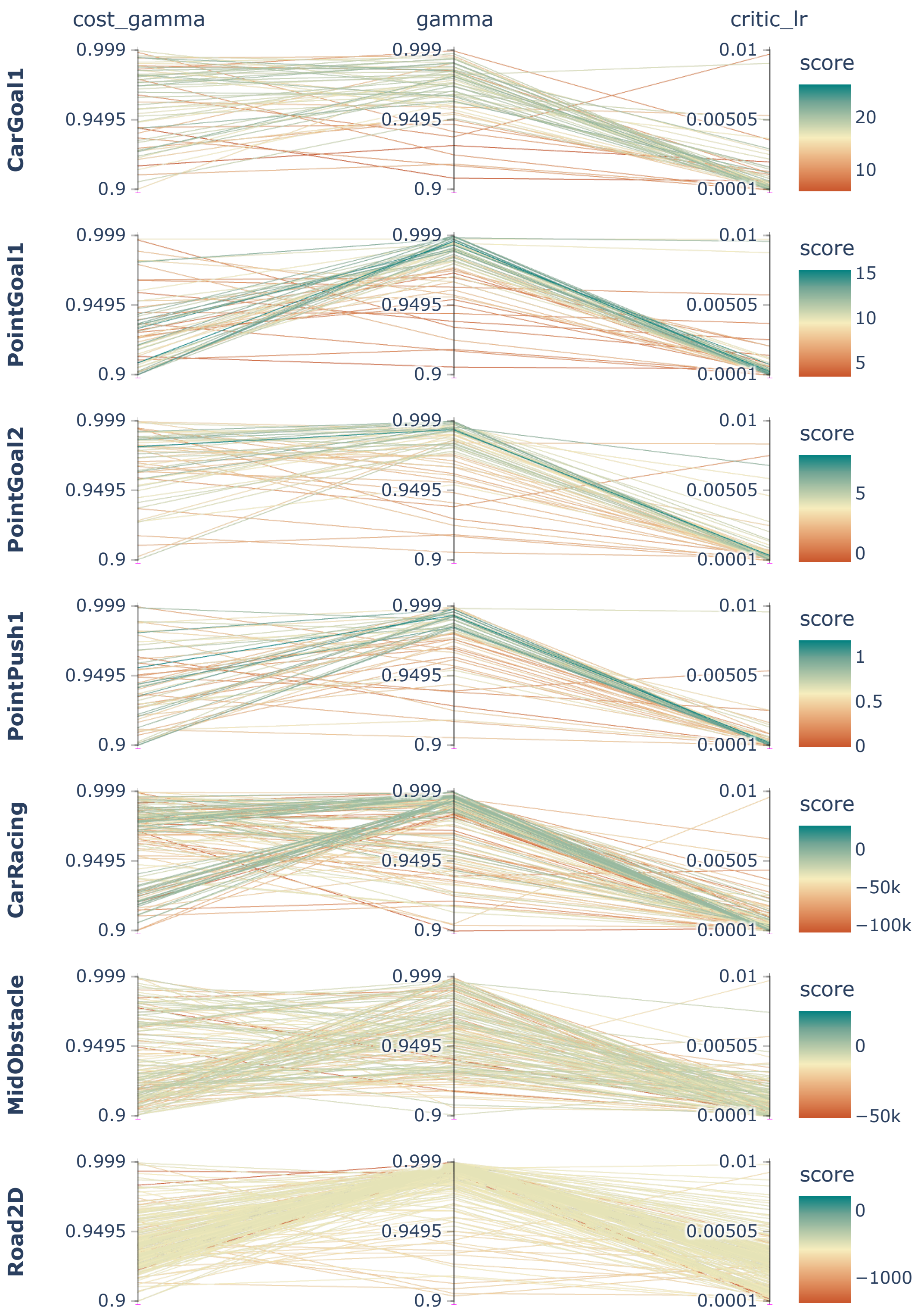


Figure C.1.: Tuning results per environment for CPO. SafeNavigation tasks on top, static environments below. The "Safety" prefix is omitted for brevity.

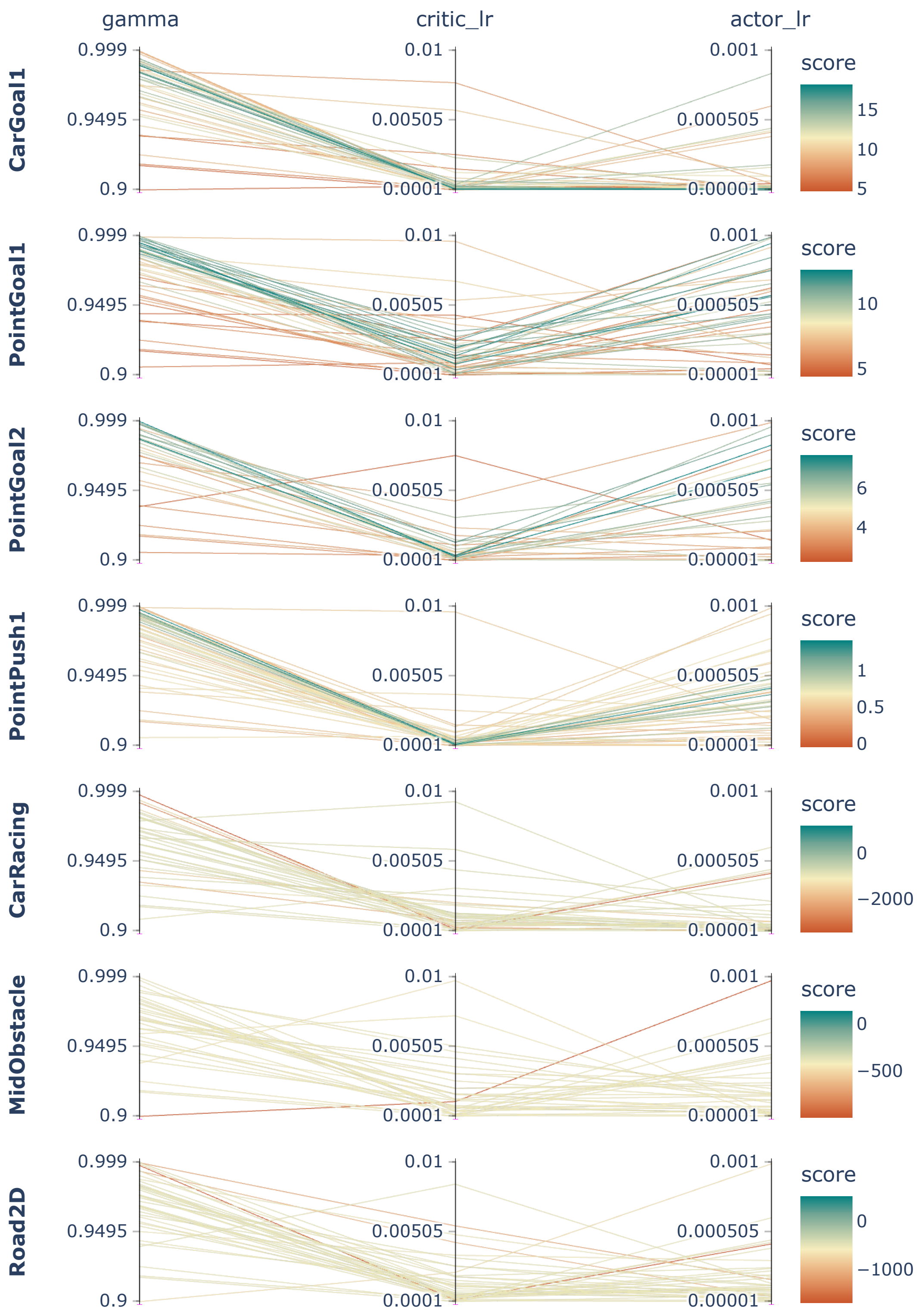


Figure C.2.: Tuning results per environment for PPO. SafeNavigation tasks on top, static environments below. The "Safety" prefix is omitted for brevity.

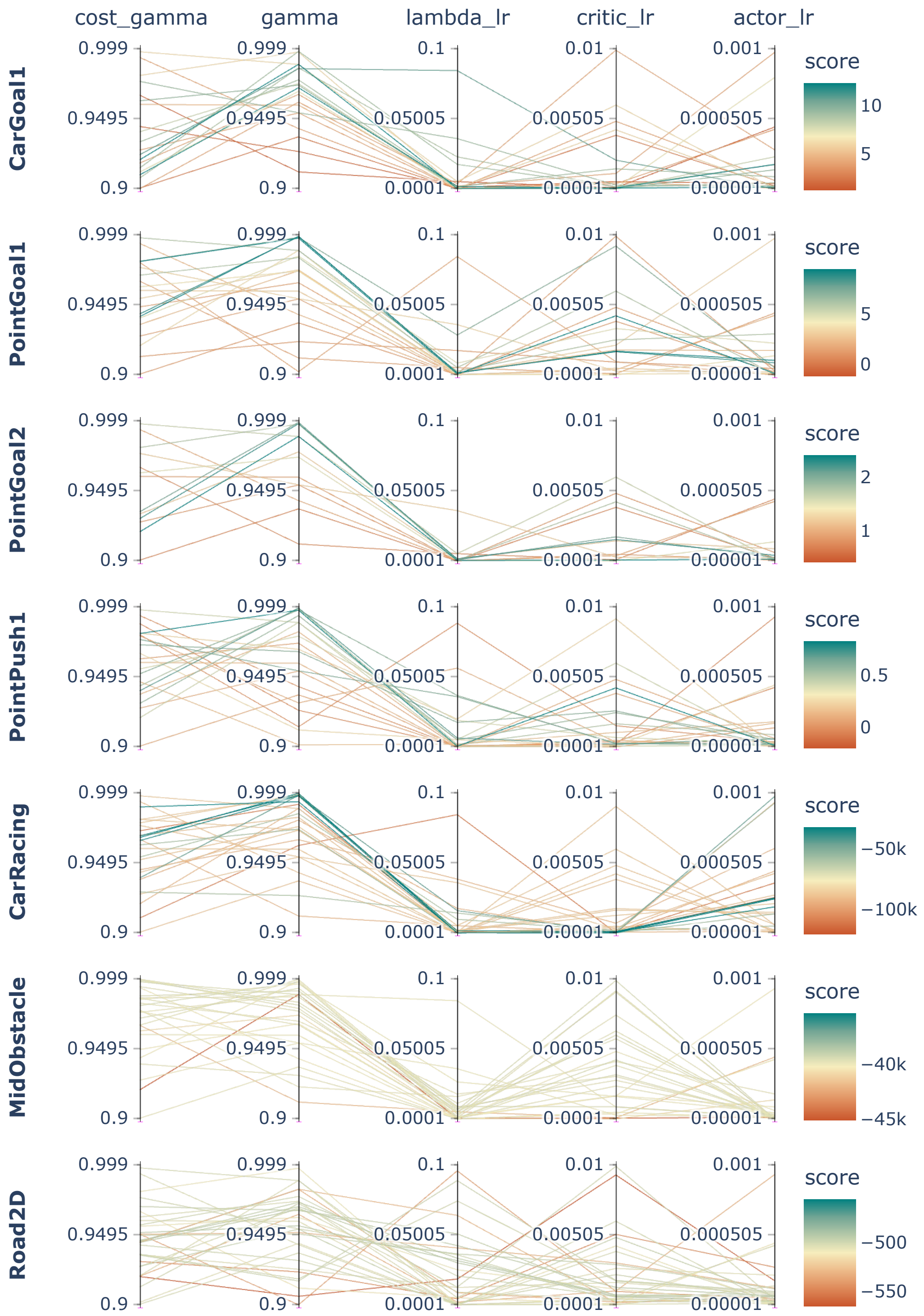


Figure C.3.: Tuning results per environment for PPOLag. SafeNavigation tasks on top, static environments below. The "Safety" prefix is omitted for brevity.

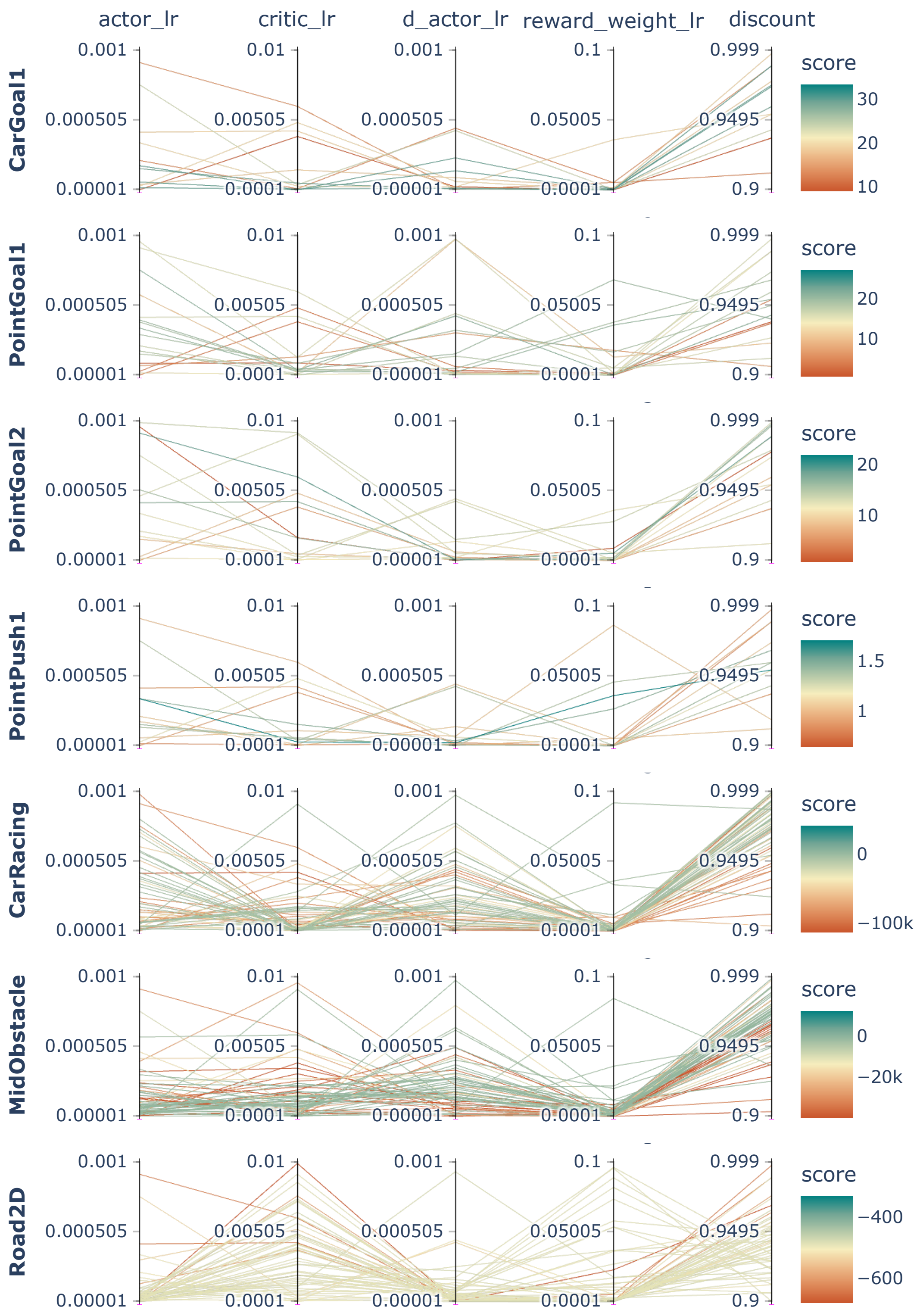


Figure C.4.: Tuning results per environment for SEditor. SafeNavigation tasks on top, static environments below. The "Safety" prefix is omitted for brevity.

	CR	**MO**	**R2D**	**SCG1**	**SPG1**	**SPG2**	**SPP1**
CPO							
cost_gamma	0.92121	0.91616	0.92925	0.99401	0.90867	0.98095	0.95544
gamma	0.99273	0.96729	0.99336	0.98434	0.99511	0.99306	0.99269
critic_lr	0.00016	0.00220	0.00314	0.00012	0.00036	0.00047	0.00032
PPO							
gamma	0.98127	0.96853	0.96795	0.98862	0.99424	0.99896	0.99688
critic_lr	0.00110	0.00363	0.00038	0.00015	0.00090	0.00037	0.00022
actor_lr	0.00001	0.00017	0.00013	0.00002	0.00058	0.00067	0.00042
PPOLag							
cost_gamma	0.98922	0.98036	0.95126	0.92071	0.98036	0.92071	0.98036
gamma	0.99299	0.99688	0.95612	0.98824	0.99688	0.98824	0.99688
lambda_lr	0.00024	0.00066	0.03183	0.00046	0.00066	0.00046	0.00066
critic_lr	0.00017	0.00429	0.00082	0.00016	0.00429	0.00016	0.00429
actor_lr	0.00020	0.00002	0.00008	0.00002	0.00002	0.00002	0.00002
SEditor							
actor_lr	0.00057	0.00018	0.00008	0.00003	0.00076	0.00092	0.00035
critic_lr	0.00028	0.00011	0.00645	0.00016	0.00047	0.00604	0.00037
d_actor_lr	0.00032	0.00015	0.00001	0.00002	0.00043	0.00001	0.00003
reward_weight_lr	0.00215	0.00043	0.00013	0.00046	0.00034	0.00538	0.03607
discount	0.98649	0.97351	0.96521	0.98824	0.94274	0.98815	0.95392

Table C.1.: Tuned hyperparameters for all algorithms and environments used in this study. Rounded to five decimal places. Environment names are abbreviated in accordance with Section 5.2.

D. Heatmaps

As an addition to the results in Chapter 8, we provide heatmaps that visualize the behavior of the analyzed checkpoints in the environments CarRacing, MidObstacle, and Road2D. The heatmaps are provided in Figure D.1 as a 6x3 grid, where each row corresponds to a specific algorithm and each column to a specific environment. For the trajectory generation, we used the exact same settings as in the PyDSMC runs, i.e., the same checkpoints and seeds. In summary, each heatmap was generated using a collection of 1024 sampled trajectories. When looking at these heatmaps, it's important to note that each of the three environments limits the available space of the agent, and we decided to further crop each heatmap to contain the most relevant region. Some trajectories might be cut off, but more importantly, due to the boundary imposed by the environment, in some of the plots—for example, CPO in CarRacing—the trajectories suddenly show a perfectly straight line. This is a direct artifact of the environment clipping the agent to stay inside the bounding box.

First of all, when looking at these heatmaps, we can directly confirm our previous findings in the evaluation with PyDSMC. In Road2D, almost all algorithms travel directly to the goal area without any problems. The only exception is SEditor, which focuses on some location southwest of the goal area. The exact same behavior is visible in CarRacing and MidObstacle, but in these two environments, SEditor circumnavigates the obstacle first. As we argued in the discussion of the statistical evaluation, we suspect that this unwanted behavior can be attributed to the smoothness of the reward function and the reward normalization of SEditor.
Furthermore, the heatmaps confirm our suspicion that PPO and PPOLag instantly crash into the obstacle in MidObstacle. In CarRacing, PPOLag instead travels in the opposite direction—away from the obstacle and the goal area. We did not expect this behavior, given that the reward function is designed to reward the agent for going towards the goal area. Also, we do not exactly know the cause, but one reason could be a combination of the inner workings of PPOLag and the reward function design of CarRacing. In CarRacing, the agent has to first travel to the green area and then back to the start. The reward function always points to the next target, i.e., as soon as the agent reaches the top-right corner, it flips and directs the agent back to the start. This sudden change could be the reason why we see all algorithms struggle and, at best, make it to the first checkpoint. Refer to CPO and SPICE-CSC, both of which are able to reach the green area but show no sign of movement back towards the start.
On top of that, the heatmaps show that both SPICE and SPICE-CSC exhibit very cautious behavior. Both algorithms keep a great distance from the obstacle but struggle to find a safe path to the goal area. On the other hand, CPO also acts rather conservatively but is able to consistently solve the task in MidObstacle. Compare this to SEditor, which efficiently circumnavigates the obstacle with little to no distance but has the aforementioned problem of being stuck in a local minimum.

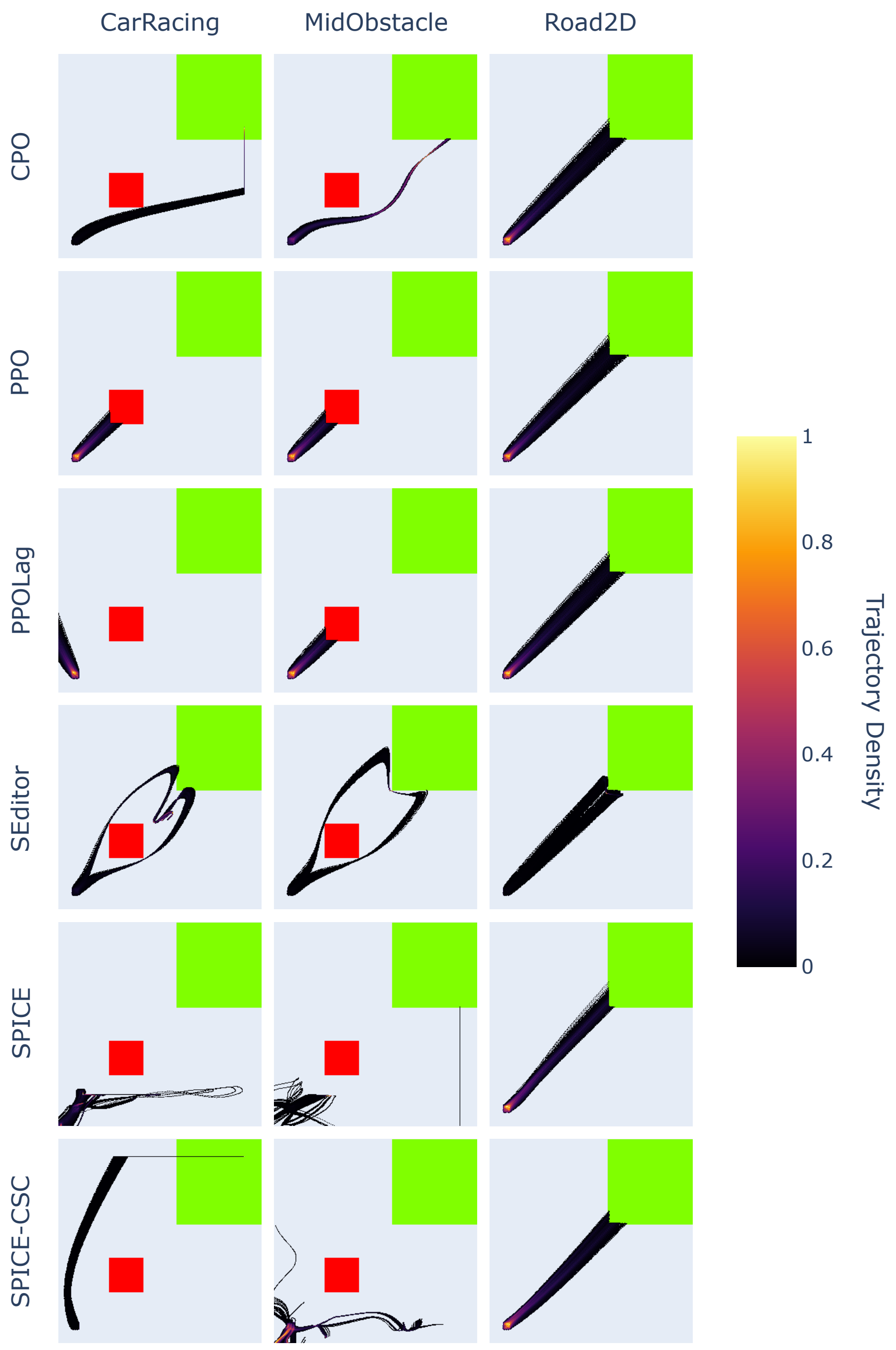


Figure D.1.: Heatmap visualizations of checkpoints used in Chapter 8 for the environments CarRacing, MidObstacle and Road2D.

E. Source Code

Here, we provide the source code for this thesis. We had to make adjustments to most of the existing codebases to either fix bugs, add features, or adapt them to our needs.

- **CPO, PPO, PPOLag**: `https://gitlab.cs.uni-saarland.de/s8phsaue/cpo`, a small wrapper to run algorithms from OmniSafe on our modified benchmarks.
- **OmniSafe**: `https://gitlab.cs.uni-saarland.de/s8phsaue/omnisafe`, OmniSafe with minor bug fixes.
- **ALF, SEditor**: `https://gitlab.cs.uni-saarland.de/s8phsaue/alf`, ALF with support for our benchmarks and SEditor adjustments for tuning.
- **SPICE, SPICE-CSC**: `https://gitlab.cs.uni-saarland.de/s8phsaue/spice`, SPICE with bug fixes, additional customizations, and extended logging.
- **Py-Earth**: `https://gitlab.cs.uni-saarland.de/s8phsaue/py-earth/-/tree/python3.10`, Py-Earth ported to Python 3.10.
- **Benchmarks**: `https://gitlab.cs.uni-saarland.de/s8phsaue/benchmarks`, collection of all benchmarks used in this thesis.
- **SafetyGymnasium**: `https://gitlab.cs.uni-saarland.de/s8phsaue/safetygymnasium`, SafetyGymnasium with bug fix for natural lidar.
- **PyDSMC, Heatmaps**: `https://gitlab.cs.uni-saarland.de/s8phsaue/dsmc`, small wrapper for PyDSMC and code for generating heatmaps.
- **Optuna**: `https://gitlab.cs.uni-saarland.de/s8phsaue/optuna`, wrapper for running Optuna on our modified codebases and benchmarks.

F. Note on the Usage of Generative AI

With the release of ChatGPT [40] back in 2022, the deployment and usage of generative AI (GenAI) applications in everyday life have continuously increased. Large Language Models (LLMs) such as ChatGPT can assist the user in a variety of different tasks, ranging from researching and simple text generation to complex problem solving. As a consequence, these tools are also frequently utilized in academia, e.g., by students who require help with their exercises or who want to learn more about a certain topic. Though these tools bring great benefits, there is also the potential for misuse, which is especially worrisome when it comes to, for example, writing reports, examinations, or exams. As GenAI was also used in the creation of this thesis, we want to be transparent about its contributions.

We used the Windsurf [1] extension in Visual Studio Code

- for code completion when coding, and
- for text completion during writing.

On top of that, we made use of Microsoft Copilot [2]

- as an assistant during code debugging,
- as a spelling and grammar checker, and
- as a search engine.

This is the prompt we used for spelling and grammar checking:

> I am a researcher and I am writing a paper for a highly recognized conference. Your job is to correct any spelling or grammar mistakes in the document. Try to change as little as possible of the original content, but make sure to correct all mistakes. Only return the corrected text. Here is the text you should correct:

The main motivation for using these tools was to increase productivity, to assist in learning and to enhance writing quality. In the context of this thesis, we believe that this is a reasonable and appropriate use of generative AI.

[1]https://windsurf.com/
[2]https://copilot.microsoft.com/

Furthermore, we did not use GenAI

- to automatically generate entire blocks of code,
- to automatically generate entire paragraphs of text, or
- for any other purpose other than the ones mentioned above.