POLITECNICO DI TORINO

Master degree in Data Science and Engineering

Master Thesis

Learning Constraints in Robot Trajectories

Supervisor
Prof.ssa Barbara Caputo

Co-supervisors:
Prof. Luc De Raedt
Dott. Paolo Morettini

Candidate
Giulia Tortoioli

April 2022

Acknowledgement

I would like to thank my family, my greatest luck, that supported me and believed in me throughout my life.

I would also like to be grateful to the professors and to Paolo whose guidance was really helpful.

Special thanks go to my friends, Italian and international, who made me cherish my university experience.

Abstract

Constraint learning is a research area covering different methodologies for an inductive construction of constraint theories from data. In many contexts, a system can benefit from constraint learning to optimize the process in terms of time and performance. In this thesis, trajectories data of a trained reinforcement learning agent are used as examples to define constraints characterizing the environment. The methodology involves three main steps. In the first phase an agent is trained in a two-dimensional space containing unsafe areas. Then, the agent is tested and its trajectories are stored and used as data for learning a formula that classifies the region of the training examples as positive, thereby identifying safe and unsafe areas. In the last step a new agent is trained by using the formula to assign a penalty to movements towards an unsafe area. The addition of prior knowledge about the environment leads to a significant improvement of the agent's performance in terms of satisfaction of safety specifications. Hence, in fields like autonomous robotics, where safety concerns are paramount, systems can benefit from constraint learning to increase the level of reliability and guarantee a "safer" exploration.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Contextual Overview

The trait that distinguishably characterizes human being is intelligence, the ability to acquire and use knowledge to solve problems. The last century has seen the need for faster and efficient problem-solving systems to deal with tasks with increasing computational complexity, thereby requiring novel technologies. In 1956 the term Artificial Intelligence (AI) was introduced to define the ability of computers to perform human tasks. Machine Learning (ML) is a branch of AI where a system is able to learn and make decisions by utilizing algorithms to obtain inference from data.

The goal of ML models is to make predictions by identifying in the training dataset patterns which generalize the actual distribution of the data. In real world problems, for a solution to be considered valid, ML systems must often satisfy certain conditions along with solving the task. These conditions represent limitations characterizing the task, like cost restrictions, safety requirements, space limits. Identifying and measuring the limitations of the problem is fundamental in order to formulate them in a machine-friendly manner and quantitatively evaluate the results. Constraints are inserted into the learning process of the algorithm as conditions to be fulfilled or, in other situations, represent the expected outcome of the learner. The process of learning a constraint theory from data is called constraint learning. The goal of constraint learning is to define general rules expressed in the form of logical formulas by identifying patterns in a set of training data.

Constraints learned from a dataset can reveal different aspects of the data distribution, such as defining the variables domain, identifying relationships and patterns within the data. Constraint learning algorithms can be applied to automatize rules generation or integrate the knowledge of a system with additional rules by exploiting past experimental data or data stored from similar tasks. This thesis aims to use constraint learning to integrate a ML system, thereby improving the results.

1.2 Motivations

Some ML models need a very high level of reliability in order to be applied in the real world. This is the case of applications in the field of robotics and autonomous systems. These systems demonstrated a strong impact in the economy of industrial and commercial activities as well as in end-user applications and markets. One of the main concerns when building an autonomous system is to guarantee an adequate level of safety. In autonomous robotics, safe exploration is defined as the ability of the robot to achieve safe behaviours when exploring unknown states. Achieving the level of safety required, to allow use in real world, is not always easy, especially if no information is available about the space in which the robot has to move. Information acquired from past experiments or demonstrations is helpful to enrich the system and increase its robustness from failures. Many studies were conducted about guaranteeing safe exploration thanks to integration of past experience in the training process of the robot [4, 5].

1.3 Main Contributions

In this thesis, constraint learning is applied to identify the areas which are not covered by a set of data. The data are extracted from trajectories of a trained agent which moves in a two-dimensional space and must reach a randomly positioned goal while avoiding obstacles (considered as unsafe areas). The objective of the constraint learning procedure is to determine the unsafe regions by generalizing the true support of the trajectories data.

The outcome of the constraint learning procedure is used to integrate with a constrained Reinforcement Learning (RL) system in order to improve its performance. Constrained RL agents are trained to reach a goal while avoiding some harmful regions in the space. The prior knowledge given by the learned constraint theory is inserted into the training process of the agent. In particular the agent is penalized when the learned formula predicts that it is approaching an unsafe area. The addition of the penalty is meant to reduce the probability that the trained agent will fail in satisfying the safety constraints imposed in its task while still preserving its capability to reach the goal. The integration of prior knowledge about the environment characteristics should also make the agent's training converge faster.

Part I

First part: Overview

Chapter 2

Safe Exploration

2.1 Reinforcement Learning

Reinforcement learning (RL) is an approach that implements a goal-directed learning process deriving from interaction with an environment. It differs from many other machine learning techniques as the actions of the learner are not set a priori, but the learner finds out itself the most advantageous actions to take. In fact, the idea of RL is to train an agent to perform a specific task by using a *trial and error* strategy. In a RL system the agent must be capable of perceiving the state of the environment and produce effects by interacting with it [6]. The agent operates in the environment while gradually increasing the likelihood of "behaving well" by maximizing the reward function, which instructs the agent by rewarding for correct actions.

The applications of RL methods are numerous, for example they have a fundamental role in creating AI (Artificial Intelligence) agents for video games, which enables them to achieve super-human performances [7]. RL allows also to perform a set of different tasks, like the ones proposed by OpenAI Gym [8], a toolkit which implements various locomotive and robotic movement tasks. In OpenAI Gym the agent is trained by using an episodic approach and it interacts with the environment during the whole episode. The goal is to maximize the cumulative reward expectation for each episode and to achieve gradual improvement in results along the training process.

2.1.1 Exploration and Exploitation Trade-off

Supervised learning expects a training labeled dataset where each example contains a description of a situation and a label which defines the correct output that the system should give in that situation [6]. Usual tasks of supervised learning are classification and regression, where the objective is to generalize the representations contained in the examples of the training data in order to learn and be able to deal with new situations. While dealing with interactive problems it is very unlikely to have examples which can be considered representative of the reality since the state of the environment is influences by the agent's actions. Therefore, in RL the agent is trained to learn from its previous actions.

At the same time RL cannot be inserted in the context of *unsupervised learning* since its goal is different. In unsupervised learning the objective is the definition of a structure and/or a pattern in a data distribution by using an unlabeled dataset, while RL aims to maximize a given reward function.

RL implements the concept of trade-off between exploration and exploitation. Exploration improves the agent's awareness about the environment possibly leading to long term benefits, while exploitation makes use of the agent's current ability to gain the maximum possible reward. The strategy is to find an optimal balance between exploring new environments' features, even if selecting sub-optimal actions, and exploiting the prior knowledge [9].

2.2 Constrained Reinforcement Learning

In constrained reinforcement learning the agent has an additional objective with respect to maximizing the reward function which is to satisfy constraints. In constrained RL problems environment characteristics and limitations due to the nature of the task are expressed as constraints, while the reward function specifies the goal of task.

In real world applications, constraints specify varied types of restriction. For example, in constrained RL models built in the field of robotics, constraints can represent safety specifications for a robot which aids in navigating obstacles [10, 3]. The objective of the agent is to reach the goal while avoiding the harmful areas by satisfying the constraints. There are also some applications for which the satisfaction of safety constraints is absolutely essential like in the case of autonomous driving, where a lot of research is going on to improve the models and apply them in the real world [11]. Other applications of constrained RL include limitations in resource allocation [12] or budget expressed by constraints that have to be satisfied.

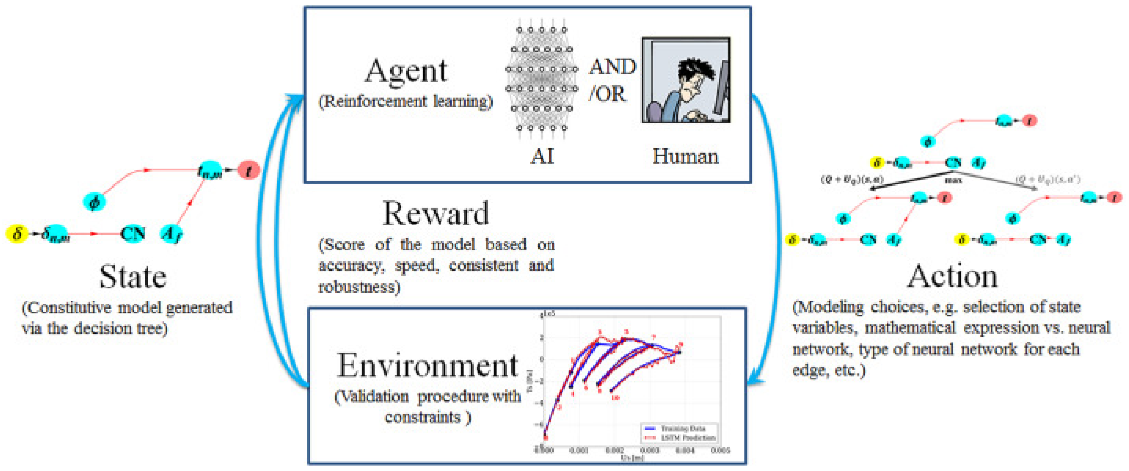

Agent
(Reinforcement learning)
AI
AND /OR
Human
State
(Constitutive model generated via the decision tree)
Reward
(Score of the model based on accuracy, speed, consistent and robustness)
Environment
(Validation procedure with constraints)
Action
(Modeling choices, e.g. selection of state variables, mathematical expression vs. neural network, type of neural network for each edge, etc.)

Figure 2.1: General structure of a constrained reinforcement learning system. Figure source: [1].

2.2.1 Satisfaction of Safety Requirements

In an environment with safety requirements the agent has to find the right **trade-off** between satisfying the task objective and fulfilling the safety specifications. The outcome of the algorithm is a behavioural policy for the agent which is based on the outputs of the previous states.

The *trial and error* process proposed by traditional RL would not be feasible in some dangerous tasks, where the negative effects of an *unsafe* exploration could be detrimental. In fact, there are some real-world problems where simulators may not be reliable enough and, therefore, it would be necessary to train the model directly in reality. A RL system that allows exploration in compliance with safety specifications defines the concept of **safe exploration**.

In order to address a safe exploration problem, the first challenge is to identify the *unsafe factors* to be measured and define a formalism based on how these metrics influence the outcome of the system. An approach to the definition of this problem's formalism is constrained RL and this is the strategy implemented in Safety Gym by OpenAI [3].

2.2.2 Constrained RL in Safety Gym

Safety Gym implements safe exploration by adding safety specifications into the RL models in the form of standardized constrained RL. The systems are implemented following two main concepts:

- keeping safety specifications and performance definitions *separate*;
- using *constraints* to encode safety specifications.

Constrained RL problem can be quantified by evaluating a set of policies which satisfy the constraints and identifying the optimal one. The set of feasible policies is defined by the framework of **constrained Markov Decision Processes** (CMPDs) [13]. Formally, a CMDP is denoted by a tuple $(\mathcal{S}, \mathcal{A}, C, R, K, x_0)$ [14]:

- $\mathcal{S}$ is the state space;
- $\mathcal{A}$ is the action space;
- C is a set of cost functions, where $c_j : \mathcal{S} \times \mathcal{A} \to \mathbb{R}, j = 1, \ldots, k$;
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function;
- $K : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is the transition probability function;
- $x_0 : \mathcal{S} \to [0,1]$ is the distribution of the initial state.

Π_C is the set of the stationary policies restricted by C, where $\pi \in \Pi = \pi(a|s) : s \in \mathcal{S}, a \in \mathcal{A}$. The reward-based objective function $J_r(\pi)$ and the cost-based constraint function $J_{c_j}(\pi)$ are defined as follows:

$$\begin{aligned} J_r(\pi) &:= \mathbb{E}_{(s_0,a_0,\ldots,)\sim\pi}[\sum_{t=0}^{\infty} \gamma^i R(s_i, a_i)] \\ J_{c_j}(\pi) &:= \mathbb{E}_{(s_0,a_0,\ldots,)\sim\pi}[\sum_{t=0}^{\infty} \gamma^i c_j(s, a)] \end{aligned} \tag{2.1}$$

Hence, $J_r(\pi)$ and $J_{c_j}(\pi)$ are defined as the expected value of the cumulative future reward function and cost function c_j, respectively, given a trajectory distribution $(s_0, a_0, \ldots,)$ depending on π and a discount factor $\gamma \in (0,1)$. Let z_i be a human-defined threshold for the cost function c_i, the formulation of the optimal policy π^* is the following:

$$\pi^* = \arg\max_{\pi \in \Pi_C} J_r(\pi), \tag{2.2}$$

$$\Pi_C = \{\pi : J_{c_i}(\pi) \leq z_i,\ i = 1, \ldots, k\}. \tag{2.3}$$

Constraints formulate the safety specifications which practically refer to *hazards* that the agent must learn to avoid. The goal is to find a good trade-off between the reward for completing the task and the penalty associated with proximity to hazards. There is not a trade-off which is optimal a priori, the right balance is highly dependent on the complexity of the environment, the chosen policy optimization algorithm and, especially, the application's safety requirements.

Safety Gym gives the possibility to create environments with different tasks and complexities. The agent is able to sense the world by using lidar [1] sensors of a robot and interacts with it. There are three pre-made robots (Point, Car, Doggo) which have different characteristics in the way of moving and have increasing complexity. The constraint elements are represented by various objects which can be added to the environment to increase the complexity and the probability for the agent to behave unsafely.

[1]Lidar is a way of determining ranges (varying distance) by focusing a laser on an object and measuring the time it takes for the reflected light to return to the receiver.

Chapter 3

Constraint Learning

Constraints are fundamental in artificial intelligence, they define a restriction on feasible combinations of assignments for a set of variables. In real-world scenarios constraints are meant to represent different kinds of limitations characterizing the application context. One approach to handle constraints is an optimization problem with an objective function and (typically linear) constraints. In every optimization problem common denominator is the goal: test and improve the performance of a model while *"being constrained"* by some factors, such as budget or physical restrictions. When formulating a real world problem, an optimization problem with specified constraints can be clearly limiting because of the potentially high level of uncertainty and variability. In fact, it is also necessary to consider the extent of the limitations and model the constraints appropriately.

A complete and precise specification of the constraints is not always possible. In some cases there are limitations concerning the task that are not known a priori but are revealed from past experiments, moreover, modelling new conditions in the form of logical constraints can be challenging and computationally expensive. Knowledge about unknown constraints can be inferred in many ways depending on the application and, especially, on the type of limitations to be modelled. Furthermore, data acquired from previous simulations or obtained from a system in its past state can be used for constraint learning. Whenever additional information expressed in the form of constraint theories is available, it can be useful to improve the system's performance (i.e. in terms of accuracy and/or time) and also to create different predictive models.

3.1 Learning from Data

The meaning of *constraint learning* differs based on the use case and the procedure used. A deductive strategy is applied by solvers' technology where constraint learning consists in speeding up the process by adding clauses [1] to the constraint theory whenever inconsistencies rise. A solver takes a formula, containing Boolean variables (with true/false value), as input and verify whether it is satisfiable or not. A formula is defined as satisfiable if there are assignments for the variables within it that make it true. The problem of satisfiability determination is called SAT (Boolean Satisfiability Problem).

On the other hand, when a dataset is used to define a constraint theory, the process is **inductive** [15]. Inferring general functions from a set of training examples is a key point in machine learning, where concept learning is the process of defining a general category from provided positive and negative instances. This problem can be thought as a search through of as a set of potential hypotheses for the one that best fits the training data. Constraint learning can be compared to concept learning (where the goal is inducing a binary function) since a constraint theory allows to identify satisfiable and unsatisfiable instances. However, learning constraint theories can also be useful in different contexts, where the purpose is different from just a binary classification of the instances.

3.1.1 Formalization of Constraint Learning

Considering the definition of concept learning, constraint learning can be formalized in line with the PAC-learning (Probably Approximately Correct learning) framework [16]. The goal of PAC-learning is to find an algorithm that, when trained on a random sample of a data distribution, selects a function (defined as *hypothesis*) with a low generalization error.

Hence, a definition of constraint learning can be formulated as follows [15].

Let

- I be a set of possible instances representing partial or complete assignments for the variables V of the unknown constraint theory;

[1]In logic, a clause is defined as disjunction of literals (representing normal atoms and their negations). A more detailed explanation can be found in 4.2.2.

- C be a set of possible constraints;
- $T \subseteq C$ a target constraint theory (unknown);
- X be a set of training instances that are positive when they satisfy T, negative otherwise;

a **constraint theory** $\mathcal{H}$ ($\mathcal{H} \subseteq C$) is found when all the positive instances, and none of the negative ones, contained in X are satisfied (or satisfiable when considering partial assignments) in H.

As before mentioned, the SAT problem consists in assigning a binary value (true or false) to a specific formula basing on its satisfiability. However, the set of problems that can be solved when formulating a constraint theory goes beyond just binary classification. For example, one type of problem that can be solved with constraints' definition is to find a completion to partial assignments of the variables within the formula. Constraint theories can be used to infer values to a subset of variables $V \setminus X_v$, given particular assignments for the other variables $X_v \subseteq V$. If the task consists in finding the best possible set of values for the variables with unknown assignments, constraint learning recalls a problem of learning a probabilistic model.

3.1.2 Approaches to Constraint Learning

Many situations in which constraints or constraint optimization are applied benefit from constraint learning. Learning logical theories is possible in applications like constraint programming, operations research and other machine learning problems. A technique to learn constraint theories is inductive logic programming, a branch of machine learning that looks towards inductively building first-order clausal theories using instances and prior knowledge [17].

Constraints can be learned by data coming from different sources, such as datasets or spreadsheets. A system called ModelSeeker [18] was developed to work with data coming from many kind of domain (such as puzzles and sports scheduling) to generate a set of matching constraints from a global catalog. It resulted that the model could find constraints set even when dealing with very limited amount of examples. Another example of a system constructing constraints from a data source is a tool, named Tacle, which reconstructs formulae applied in a spreadsheet by taking as input a comma separated file [19]. An unsupervised approach was adopted to infer constraints defining relations and/or formulas involving both rows and columns. The result was a spreadsheet integrated with the dependency formulas between cells.

Constraint learning can be helpful also in many Natural Language Processing (NLP) [2] applications, where it is often necessary to reconstruct the structure of the text and to do it by hand is a very tedious process.

A solution to reduce the human involvement was a model which detects latent topics and their main linguistic properties in input documents and then uses the acquired knowledge to infer constraints [20].

Furthermore, many studies were made in learning logical rules in the field of knowledge base reasoning [21] and automatic error detection [22].

3.2 Constraints as Prior Knowledge

When creating a new system for any application, the initial phase usually consists in building a reliable and fairly general model suitable for tasks similar to the one considered. Clearly, in a specific use case the system may need further specifications to achieve better (and in some cases acceptable) performance.

An example is a supply chain optimization task, where the limitations to be considered are multiple, in terms of costs, allocation of resources given the limited space, production times and customer requests. In this case data collected from different sites of the production chain can be used to get more in-depth information on the process in its strengths and bottlenecks. In contexts like this one, the output of a constraint learning process, consisting in a constraint theory which specifies the data distribution, represents a very powerful source of information that, if integrated into the system, helps to improve the current model. Furthermore additional constraint specifications can lead to the creation of different probabilistic models for other tasks, such as monitoring and analysis of the system performances.

3.2.1 Importance of Prior Knowledge

Prior background knowledge, seen as critical acquisition of general information, has a strong impact in learning since it allows to generalize to future situations. Strong benefits were shown, for example, in category learning, where prior knowledge, connecting to category's characteristics, helps the

[2]Natural language processing utilizes computational techniques to analyse and create natural language and speech.

learner to speed up the learning process by identifying an eventual underlying thematic pattern [23]. Prior knowledge has also been viewed as beneficial because it imposes limits on the learning process, such as limiting the number of hypotheses that learners consider [24].

Machine learning algorithms can be guided from different knowledge representations (i.e. logic rules, algebraic representations, prior simulation results) to create an "informed" model [25]. The advantages brought by this type of process are mainly for models that do not have enough data to be able to generalize on the real distribution or in cases where data are not informative enough to achieve the desired performance. The integration of prior knowledge allows a greater robustness, accuracy and in some applications a more computationally efficient learning process. Moreover, there are certain types of applications that require a system with an extremely high level of reliability to ensure an adequate level of safety or, in other cases, a successful prevention from failures that would lead to considerable economic damage. In these situations, ensuring the desired level of model's performance often requires the integration of additional knowledge and experience.

3.2.2 Applications

Prior knowledge can have an important role in Deep Learning (DL) [3]. In fact, Deep Neural Networks (DNNs) are usually trained by using large datasets which are not always available, this is why transfer learning is frequently used [26]. Transfer learning consists in building a machine learning model for a problem by basing on knowledge acquired in solving similar tasks. In order to make DL models less empirical, and more fast and scalable, the exploitation of domain knowledge can be advantageous [27]. Different techniques are applied in this field, such as the integration of logic rules describing some task specifications and/or limitations to regularize and boost the performance of a deep learning model for image classification [28]. Another common strategy to use prior knowledge in DNNs is to add penalty components to the loss function minimized by the model. This technique was applied by Rieger et al. [29] that demonstrated how alignment with domain knowledge by penalizing neural networks removes bias and improves the predictive accuracy

[3]In the context of machine learning, deep learning is focused in creating algorithms which emulate the structure and the functionalities of the human brain. These algorithms are called artificial neural networks or deep neural networks, depending on their complexity.

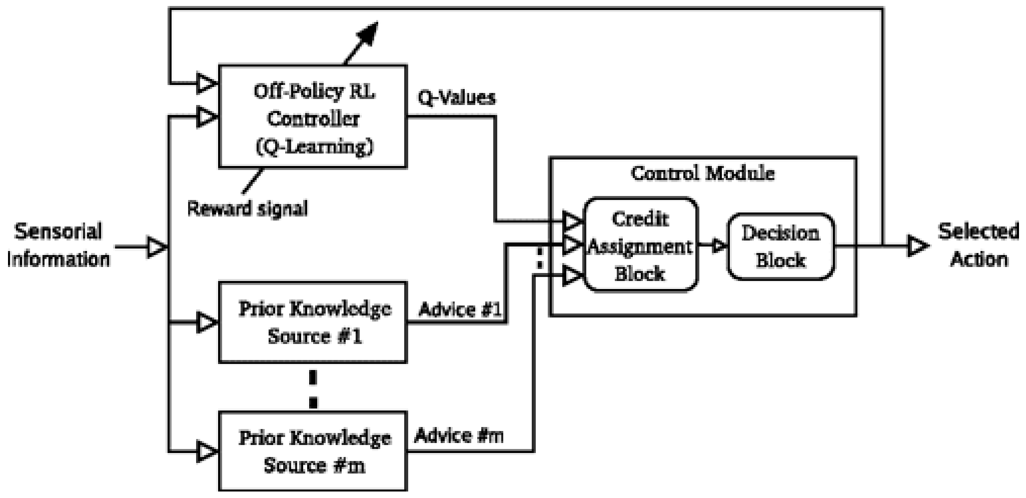


Figure 3.1: Supervised reinforcement learning block diagram. Figure source: [2].

on different datasets. Prior domain knowledge can be expressed in the form of constraints describing known rules (i.e. physics rules) and be used to supervise the training of a neural network [30].

The incorporation of prior knowledge is also applied to causal networks (structures containing causal relationships). Past experiments can be exploited to infer new constraints that complete and improve the original causal model [31]. Furthermore, relationships between nodes can be used for the creation of a knowledge graph to infer a causal Bayesian network with a learning accuracy substantially higher with respect to the original one [32].

In time series the injection of prior knowledge to catch temporal relationships is more challenging since it is required a computationally efficient method which is able to reveal latent causes even when the prior information is not completely correct. This problem was tackled by creating an efficient approach to reconstruct the timing of latent variables by using prior knowledge from similar datasets or other time periods [33].

3.2.3 Prior Knowledge in Reinforcement Learning

Traditional RL systems are built to rely only on the current state of the environment to predict the best future actions for the agent. This is one of the limitations of RL that results in lack of robustness and stability in certain applications, as well as in a slow convergence of the algorithm. The general idea is to reproduce the human capability of recalling past experiences to take action when facing with danger.

Whenever some prior knowledge on the environment's limitations is available (and it often happens in real cases), it can be used to enhance the stability of the agent during its learning process and speed up the convergence. This kind of process is applied in many applications in the robotic field, where complex environments require very long training time for an agent, without any prior information, to learn how to move in the environment and/or interact with it.

The integration of RL systems with additional information has been addressed by applying different strategies. For example, in [2] a *control module* is developed to balance the information transfer between the RL module and the prior knowledge sources throughout the training, generating a sort of Supervised Reinforcement Learning (see Figure 3.1). Another technique to avoid the so-called *tabula-rasa learning* is to initialize the network weights by exploiting the information on the domain [34].

A further problem could be faced in the acquisition of information, because often the only strategy is to use examples, especially in the field of robotics. If not known a priori, background knowledge can be extrapolated from demonstrations (from source agents or humans) and used to learn a policy [35] or as an integration for the learning process [36].

Frameworks proposed by OpenAI for RL algorithms development (Gym [8] and Safety Gym [3]) were used to experiment the integration of external knowledge in the training process. Recently, Hsu et al. [37] implemented an unsupervised technique to select a action for the agent in a particular state within a buffered store of safe actions. This method, when integrated with the conventional policy optimization algorithm, significantly reduced the number of agent's failures. In another study conducted by Pinto et al. two agents were jointly trained, one with RL to learn an optimal policy for a specific task (chosen between the ones proposed by OpenAI Gym) and the other to induce disruption in the learning process. This approach increases the robustness of the RL model to errors and uncertainties in different test environments [38].

Part II

Second part: Constraint Learning from Trajectories

Chapter 4

Method

The methodology involves three main steps:

1. Constrained RL agent traning;

2. Constraint learning process;

3. Application in new training.

The objective is to demonstrate that it is possible to use 2D data to learn constraints which can help an agent to respect safety specifications in a reinforcement learning task. In this chapter the methodology is illustrated in detail following the different steps.

4.1 Constrained RL Agent Training

As a first step, the environment is set and created to train the agent. The framework that was used to build the environment is **Safety Gym** [3] (overview in Section 2.2.2).

The library, as previously mentioned, provides different kinds of robot (specified through MuJoCo XML files [39]), and for this work the simplest one (the Point) is tested and used for the experiments. The set of algorithms for constrained and unconstrained RL are implemented in a repository, provided by OpenAI, which is called *Safety Starter Agents* [3].

All the environments are built in 2D space and the training of the agent is episodic. All the actions of the agents are continuous and linearly scaled in the interval $[-1, +1]$.

4.1.1 Task Definition

Safety Gym proposes three different tasks, which are mutually exclusive. In this thesis, **Goal** was implemented: a goal circle is generated in a random position and the robot has to move towards it. Once the robot has reached the goal, a new one is created in another random position in the 2D space and the process repeats until the end of the episode.

Two types of rewards are associated to the goal achievement:

1. *sparse reward*: corresponds to the prize obtained when reaching the goal;

2. *dense reward*: an additional reward given for approaching the goal.

At each the step i, the total reward value is updated by summing the contribution of the sparse $r_s(i)$ (always ≥ 0) and dense $r_d(i)$ rewards:

$$r(i) = r_s(i) + r_d(i)$$

$$r_s(i) = \begin{cases} R_s & \text{if } \delta(i) \leq G_r \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$r_d(i) = (\delta(i-1) - \delta(i))R_d$$

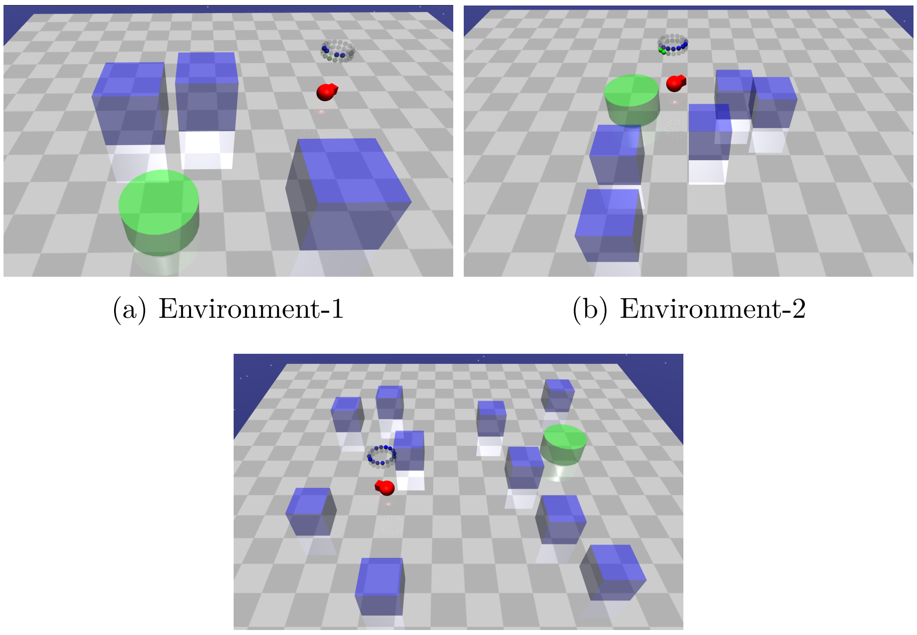
Where $\delta(i)$ is the distance between the robot and the center of the goal at the step i, G_r is the size of the goal radius, R_s and R_d are the values associated to the sparse reward and dense reward respectively.

4.1.2 Environment Setting

Within the environment-builder Safety Gym it is possible to both use environments with predefined configuration and create customized ones. All the specifications of a new environment (differing from the default ones) are listed in a dictionary that is used as an input for the class *Engine*.

Considering that the purpose of the training is the collection of trajectories of a trained agent, it is preferable not to insert constraint elements of different types within the space since they would not be interpretable and would only increase the complexity of the training. For this reason the elements chosen as “obstacles” are the **Hazards** and they do not represent physical obstacles, but rather areas to be avoided because potentially harmful.

Every hazard is represented in the 2D space with a cubic shape with a specified dimension. Various environments were created with different levels of complexity in order to test the behaviour of the agent while increasing the number of constraints and changing their dimensions (see Figure 4.1).


(a) Environment-1 (b) Environment-2

(c) Environment-3

Figure 4.1: Three environments created with Safety Gym. *Environment-1* contains three hazards with dimensions of 0.6 units, while *Environment-2* and *Environment-3* present five and ten hazards respectively with dimensions of 0.4 units. The green cylinder indicates the goal. The agent is represented in red with goal and hazards pseudo-lidars above.

The robot Point is equipped with one actuator to turn and one to move both forward and backwards. A small square is placed in front of the robot and facilitates the interpretation of the direction of its movement.

The agent can cross the hazards during training, but every time it enters one of them there is a penalty. The cost $c(i)$ for each timestep i dependent on the presence of the hazards is formulated as follows:

$$
\begin{gathered}
c(i) = \sum_{h \in H} c_h(i) \\
c_h(i) = \begin{cases} H_c(H_s - \gamma_h(i)) & \text{if } \gamma_h(i) \leq H_s \\ 0 & \text{otherwise} \end{cases}
\end{gathered}
\tag{4.2}
$$

Where H is the set of hazards in the environment, $\gamma_h(i)$ is the distance between the agent and the hazard h at step i, H_c is the cost associated to the violation of the constraint and H_s is the size of the hazard (that is half the side length).

The layout of the environment is randomly initialized at the start of the training and it stays constant throughout the episodes. Instead, the initial position of the agent is randomly selected at the start of each episode.

4.1.3 Policy Optimization: Lagrangian Methods

To address continuous control tasks, policy optimization is an efficient reinforcement learning strategy, where the policy represents the function that maps the agent's state to its next action. Reinforcement learning is considered to be an optimization problem in which the predicted reward is optimized in relation to the policy parameters.

When dealing with constrained RL there is the need for an algorithm where the policy improvement step ensures both an increase in reward and the fulfillment of the constraints. The choice of the algorithm mainly depends on the performance that it gives considering the metrics of interest for the specific task.

In Safety Gym two methods are proposed for the constrained environments: Constrained Policy Optimization (CPO) [40] and Lagrangian methods. CPO guarantees constraints satisfaction by solving trust region optimization problems each time the policy is updated. Lagrangian methods, instead, are applied on Proximal Policy Optimization (PPO) [41] and Trust Region Policy Optimization (TRPO) [42] to enforce the constraints by applying adaptive penalty coefficients.

For this application the key point is to have an agent that "fails as little as possible", which means that the most suitable algorithm is the one that gives the lowest cost rate. In Safety Gym all the algorithms were tested on different environments. By looking at the results on the experiment *Point-Goal1* (Figure 4.2), the algorithms which give the best performances in terms

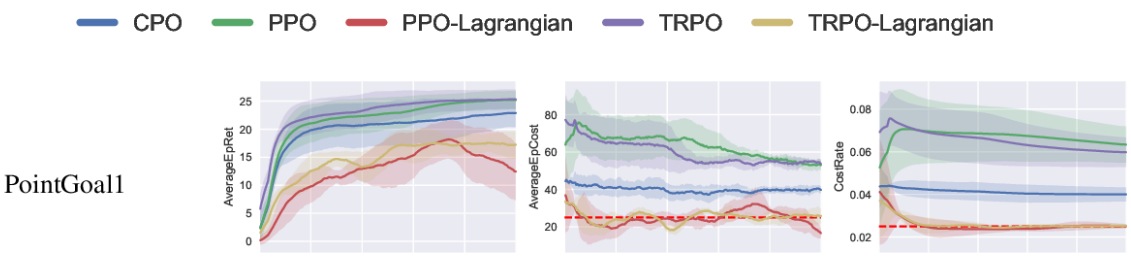


Figure 4.2: Evaluation of five algorithms throughout the training of *Point-Goal1* (task: Goal, robot: Point, constraints: medium density of unsafe elements). Three metrics are considered: average episodic reward, average episodic cost and cost rate. Figure source: [3].

of safety are **PPO-Lagrangian** and **TRPO-Lagrangian**.

TRPO includes a Kullback–Leibler (KL) divergence constraint which allows a monotonic improvement with the right hyperparameters. KL divergence measures the difference between two distributions. TRPO ensures that the policy does not change much from one step to another by limiting the value of KL divergence. The formulation of the optimization problem implemented by TRPO is the following:

$$\begin{aligned} &\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_i[\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta old}(a_i|s_i)}\hat{A}_i] \\ &\text{subject to} \quad \mathbb{E}_i[KL[\pi_{\theta old}(\cdot|s_i), \pi_\theta(\cdot|s_i)] \leq \tau \end{aligned} \tag{4.3}$$

Where τ is the boundary set on the value of the KL divergence and corresponds to the size of the trust region. $\hat{A}_i$ is a function which measures the advantage of choosing a specific action in a certain state.

In PPO there is a variation with respect to TRPO, the constraint is not added separately but it is incorporated into the objective function as penalty. KL divergence is multiplied by a constant and subtracted from the objective function. PPO's performance is similar to TRPO's in terms of reliability but, since it uses only first-order optimization, the implementation is much simpler. In both algorithms the objective function does not contain information about the costs,it is rather a function of the expected reward. Hence, the Lagrangian approach is combined with PPO and TRPO algorithms resulting to solve the following **max-min optimization problem**:

$$\max_{\theta} \min_{\lambda \geq 0} \mathcal{L}(\theta, \lambda) = f(\theta) - \lambda g(\theta) \tag{4.4}$$

Where $f(\theta)$ is the objective function, $g(\theta) \leq 0$ is the constraint and the problem is solved by gradient ascent on the set of policy parameters θ and gradient descent [1] on the regularization rate λ.

4.2 Constraint Learning Process

After training the agent, a sample of its trajectories, stored during the test phase, is used as dataset for the constraint theory learning process.

The goal is to find a formula that generalizes to the true support of the data distribution thereby distinguishing safe from unsafe areas. The approach used for the constraints learning process is based on INCAL+ [43], an algorithm for support learning from positive-only data. Some additions were applied to the method in order to guarantee optimal results in a two-dimensional setting.

Part of the trajectories data is used as validation set to choose the formula, if more than one is produced by the algorithm. The result is a formula which is only satisfied by datapoints included in the area predicted as safe.

4.2.1 Trajectories as Training Data

Once trained, the agent is tested for a pre-defined amount of episodes during which the process is monitored through a set of metrics. At the end of each episode the following metrics concerning the current episode are provided as an output:

- *EpRet*: cumulative reward obtained in the episode;
- *EpCost*: cumulative cost for violations in the episode;
- *EpLen*: number of steps for the episode;

The value of *EpCost* determines how much "precise" the agent has been in the considered episode, thus a value of 0 indicates that the agent was able to

[1]Gradient descent and ascent are first-order optimization algorithms which, given a function, find a local minimum and maximum, respectively.

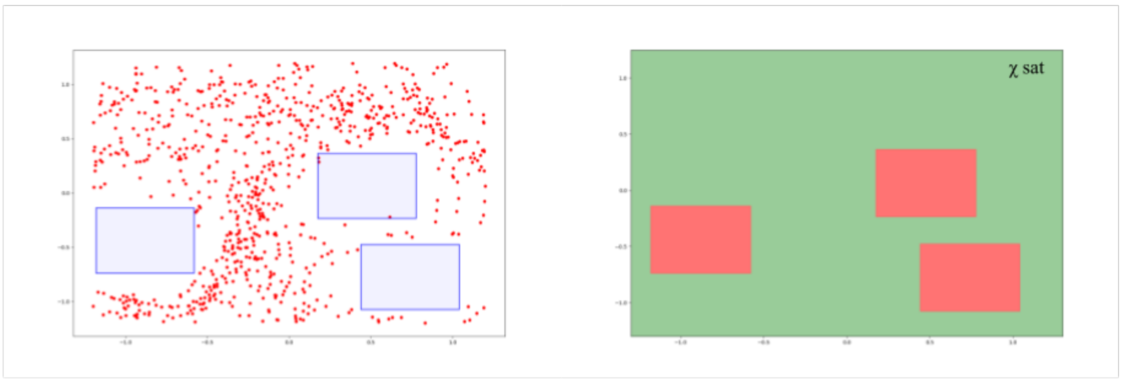


Figure 4.3: The first figure represents a sample of trajectories in *Environment-1* with three hazards. The formula χ retrieved by the learning algorithm should be as similar as possible to the one shown in the figure on the right. The area satisfied by χ is represented by the green region, while the red parts represent the area where χ is unsatisfied.

navigate without violating any of the safety specifications. This information is helpful because it allows the identification of safe trajectories that can be stored for the constraint learning process.

The result is a set of positions occupied by the agent which, considering the strategy of choice, can be defined as an **"expert agent"**.

4.2.2 Constraint Learning

The approach of INCAL+ is to use an incremental scheme to find a formula which correctly classifies the given examples, by gradually increasing the complexity.

In this application, each two-dimensional position stored from the expert agent represents an example of positive variable assignment and, since negative examples are not provided in the training data, the unfeasible region has to be inferred. The goal of the learning process is to construct a formula that designates the area occupied by the hazards in the environment as unsafe, thereby distinguishing them from the safe region (see Figure 4.3).

Critical aspects Cleaning data by removing unsafe trajectories allows to have datapoints that do not contain errors, but their distribution in the space is far from being uniform. This could drive the algorithm to wrong solutions in some cases.

For example, the agent probably follows circular trajectories and this would imply a scarcity of data in the corners if considering a rectangular or square window. These areas could be misclassified as unsafe if not containing enough datapoints. For this reason the size of the training set has to be such as to have points belonging also to areas with lower probability density. On the other hand, a larger amount of data would result in a greater probability of having points close to the unsafe areas. This would increase the computational complexity of the algorithm, therefore it would be more difficult to identify the unsafe areas.

As a solution, a subset of data representative for the particular window is selected and a small percentage of error in the result of the formula is tolerated. The strategy to incorporate tolerance to exceptions into the learning process is illustrated in 2.

Definitions In logic, propositional resolution represents a rule of inference and it works on expressions in clausal form. A propositional logical formula is composed by atomic propositions linked by logical connectives (such as And, Not, Or). A clause is defined as disjunction of literals (defining normal atoms and their negations), rendering it true even if only one of the literals is true. Every logical formula can be expressed in a distinct normal form, such as disjunctive and conjunctive normal forms. In boolean logic, Conjunctive Normal Form (CNF) is an approach to represent formulas as conjunctions of clauses with disjunctive clauses. Given the variables U, V, Z, an example of a CNF formula could be:

$$(U \vee V) \wedge (\neg V \vee Z)$$

Instead, in Disjunctive Normal Form (DNF) the formula is composed by disjunction of conjunctive clauses. The approach implemented in INCAL+ represents a variation of INCAL [44], a SMT($\mathcal{LRA}$) learning algorithm, to a case where negative examples are unavailable. SMT (Satisfiability Modulo Theories) determines satisfiability of formulas considering a specific background theory (through generalizing SAT). The background theory $\mathcal{LRA}$ refers to Linear Real Arithmetic, a logic theory that permits equations and inequalities over real numbers. Given a set of positive and negative examples, a maximum number of clauses k and linear inequalities h, INCAL finds a formula that correctly classifies all the examples. INCAL+ implements a method to estimate the negative instances to be inserted in the data used to learn the formula.

Algorithm The learning process consists of finding a SMT($\mathcal{LRA}$) formula which covers all the positive instances, except from the percentage of exceptions allowed, and is not satisfied by examples "far" from the positive ones. A distance measure d is calculated using the training data:

$$\begin{gathered} d = \max \min_{x1 \in X} L_2(x1, x2), \\ \forall x2 \in X \land x2 \neq x1. \end{gathered} \tag{4.5}$$

Hence, d corresponds to the maximum value of the N minimal euclidean distances between every datapoint and all the others in X, where N is the size of the training data.

In order to define the negative region, bounding boxes are constructed from the training data with sides of dimension θ (given by d multiplied with a user-defined threshold μ). A bounding box for a datapoint $p = (p_x, p_y)$ is formulated as follows:

$$b(p) = (x >= p_x - \theta_x \land x <= p_x + \theta_x) \land (y >= p_y - \theta_y \land y <= p_y + \theta_y) \tag{4.6}$$

The assumption is that the union of the bounding boxes B_θ of the training data is a fairly accurate estimate of the true support χ^*. So the learned formula $\hat{\chi}$ should cover as best as possible the region occupied by B_θ while not covering its complement.

The value of d stays constant if the training data do not change, so the value of the multiplier μ is changed throughout the various steps of the algorithm. The methodology is shown in 1.

The algorithm starts the learning process with the predefined threshold multiplier μ_0 and time limit. If a valid solution is found, the formula is added to the list of results and after every iteration the value of μ is updated. A predefined constant *mult* is used to decrease the value of μ in case a solution is found or increase it otherwise. In particular, given N iterations (hops), the updating rule for $\mu_k,\ k = 0, \ldots, N-1$ is the following:

Algorithm 1 Constraint learning process: adaptive search of the threshold multiplier used by the LearnSupport algorithm.

```
 1: procedure LearnSupportsAdaptive($X$, $\mu_0$, $timeout$, $mult$, $hops$,
    $neg\_bootstrap$, $tolerance$)
 2:     $i \leftarrow 0$
 3:     $results \leftarrow []$
 4:     while $i < hops$ do
 5:         $res \leftarrow$ LearnSupport($X, \mu_i, timeout, neg\_bootstrap, tolerance$)
 6:         if $res$ is found then
 7:             Add $res$ to $results$
 8:             $\mu_{i+1} \leftarrow$ decrease $\mu_i$
 9:         else
10:             $\mu_{i+1} \leftarrow$ increase $\mu_i$
11:         $i \leftarrow i + 1$
        return $results$
```

$$\mu_k = \begin{cases} decrease(\mu_{k-1}) & \text{if valid formula in iteration } k-1 \\ increase(\mu_{k-1}) & \text{otherwise} \end{cases}$$

$$decrease(\mu_j) = \begin{cases} \mu_j/mult & \text{if } \mu_j \text{ minimum multiplier} \\ (\mu_j + M_s(j)/2 & \text{otherwise} \end{cases} \tag{4.7}$$

$$increase(\mu_j) = \begin{cases} \mu_j * mult & \text{if } \mu_j \text{ is maximum multiplier} \\ (\mu_j + m_l(j)/2 & \text{otherwise} \end{cases}$$

Where $M_s(j)$ is the maximum of the previous multiplier values smaller than μ_j, while $m_l(j)$ is the minimum of the previous multipliers larger than μ_j

The threshold $\theta = \mu \cdot d$ determines the distance between the safe area identified by B_θ and the unsafe area. Lower values of θ allow the formula to learn a more detailed support but with increase in the runtime, so the adaptive search aims to find the most accurate result by respecting the imposed time limit.

At every iteration of the adaptive search a formula is incrementally learned from the training data (procedure details in 2).

Algorithm 2 Constraint learning algorithm: incremental scheme to learn a formula $\chi \in CNF(k,h)$.

```
1: procedure LearnSupport($X$, $\mu$, $timeout$, $neg\_bootstrap$, $tolerance$)
2:     $k, h \leftarrow$ initialization
3:     $\theta \leftarrow \mu \cdot d$ normalized for the variable domains
4:     $exceptions \leftarrow tolerance \cdot |X|$
5:     while $timeout$ do
6:         $Neg \leftarrow$ negative examples from complement of $B_\theta$
7:         $S \leftarrow X \cup Neg$
8:         $D \leftarrow$ sample from $S$
9:         while $|D| > 0$ do
10:            $\chi \leftarrow$ LearnFormula$(D, k, h)$
11:            if solver sat for $|S| - exceptions$ instances in $S$ then
12:                return $\chi$
13:            else
14:                $D \leftarrow$ sample from wrongly covered instances in $S$
15:        $k, h \leftarrow$ increase by using updating rule
```

Every time the LearnSupport function is called, the initial complexity of the formula χ is set: k corresponds to the number of clauses and h to the halfspaces over bounded real variables. The maximum number of examples wrongly-classified by the final formula is computed by multiplying the parameter *tolerance* (between 0 and 1) and the cardinality of the training data. The parameter *neg_bootstrap* indicates the amount of examples to sample for selecting negative instances positioned in the area covered by the complement of B_θ. The examples with negative labels are selected using a boostrap technique, consisting in a strategy of random sampling with replacement from a dataset. The value *neg_bootstrap* indicates the amount of data (sampled within the variables domain) from which the negative examples are bootstrapped. Once selected, the negative samples are added to the original dataset X containing only-positive instances.

The resulting dataset S is used as training set for learning a formula $\chi \in CNF(k,h)$.

The goal of the procedure 2 is to find a valid formula with the minimum complexity. Formally this leads to the following problem:

$$\min_{\chi} \; w_k k + w_h h \tag{4.8}$$

where w_k and w_h are the weights associated to k and h respectively. A higher weight indicates a slower increase for the parameter.

Every time LEARNFORMULA returns *unsat*, which means that the solver was not able to find a valid formula with the given k and h, the values of the parameters are updated with respect to 4.8. This process is repeated (the values of k and h keep increasing) until either a solution is found or the time limit is reached (in this case no valid formula is returned by LEARN-SUPPORT).

The learning process of the formula follows the strategy adopted in INCAL [44] for incremental learning of a SMT($\mathcal{LRA}$) formula. This technique makes the learning process more efficient by gradually adding a subset of the training data to pass to the solver (any state-of-the-art solver can be used, in this project MathSAT [45] was chosen).

The **incremental learning procedure** to retrieve the CNF formula is the following:

1. an initial subset D is randomly selected from the training data S, with dimension n (i.e. $1 \div 10$ of the original dataset X);

2. the solver starts the process to find a formula χ, with k clauses and h linear inequalities, satisfying all the data in D;

3. if χ is found, the satisfiability is tested on all the data S;

4. if the formula satisfies all the data in S except from the maximum errors allowed (*exceptions*), then χ is accepted as a valid solution. Otherwise a set of maximum $n/2$ instances of S, wrongly classified by χ, are selected to be passed to the solver (back to 2.).

The incremental scheme allows a faster and more efficient resolution than learning using all the data in a batch, which, representing an NP-complete problem [2], could be computationally very expensive in the case of a large amount of data. Moreover, this strategy allows to accept a “non-perfect” formula. In fact, the SMT solver finds a valid assignment to all the decision variables and, in this case, the resulting formula could be valid for a portion of data large enough to be accepted without the need to look for a perfect solution that is potentially difficult to find (in terms of efficiency and complexity).

[2]NP-complete are problems yet to be efficiently solved by existing algorithms.

4.2.3 Validation

The procedure LEARNSUPPORTADAPTIVE (detailed in 1) returns a set of CNF formulas retrieved by the learning algorithm with different values of θ. It is therefore necessary to define a validation process to choose the best formula $\hat{\chi}$. The objective is to find the formula which has the minimum misclassification error when compared to the true support χ^* of the data distribution.

The validation procedure is described in 3.

Algorithm 3 Validation method applied to decide the best formula between the set return by LEARNSUPPORTSADAPTIVE.

```
1: procedure VALIDATION($formulas, X$)
2:     $V \leftarrow$ samples from original dataset $X$
3:     $scores \leftarrow []$
4:     for $\chi$ and $\theta_\chi$ in $formulas$ do
5:         $W_{\theta_\chi} \leftarrow$ union of bounding boxes with size $\theta_\chi$ of samples in $V$
6:         $vol_{W'_{\theta_\chi}} \leftarrow$ volume of $W'_{\theta_\chi}$
7:         $vol_{W'_{\theta_\chi} \cap \chi} \leftarrow$ volume of intersection between $W'_{\theta_\chi}$ and $\chi$
8:         $falseNegative \leftarrow$ portion of examples in $V$ unsatisfied by $\chi$
9:         $scores \leftarrow$ add the score: $\frac{vol_{W'_{\theta_\chi} \cap \chi}}{vol_{W'_{\theta_\chi}}} + falseNegative$
10:    $\hat{\chi} \leftarrow$ formula with minimum score in $scores$
11:    return $\hat{\chi}$
```

A sample V is randomly selected from the trajectories set X and used as validation set. V is then used to "construct" an estimate of the true support by computing W_{θ_χ}, which consists in the union of the bounding boxes (4.6) of each example in V with size θ_χ (where χ stands for the current formula being validated).

A score is associated to every formula returned by the constraint learning procedure and $\hat{\chi}$ corresponds to the one with lowest score. The score for each formula χ is the sum of two contributions:

1. The first component is a division between two volume measures (the volume of a formula ϕ, in case of 2D data, corresponds to the approximation of the area satisfied by ϕ). The score is given by a division with numerator the volume of the intersection between W'_{θ_χ} (complement of

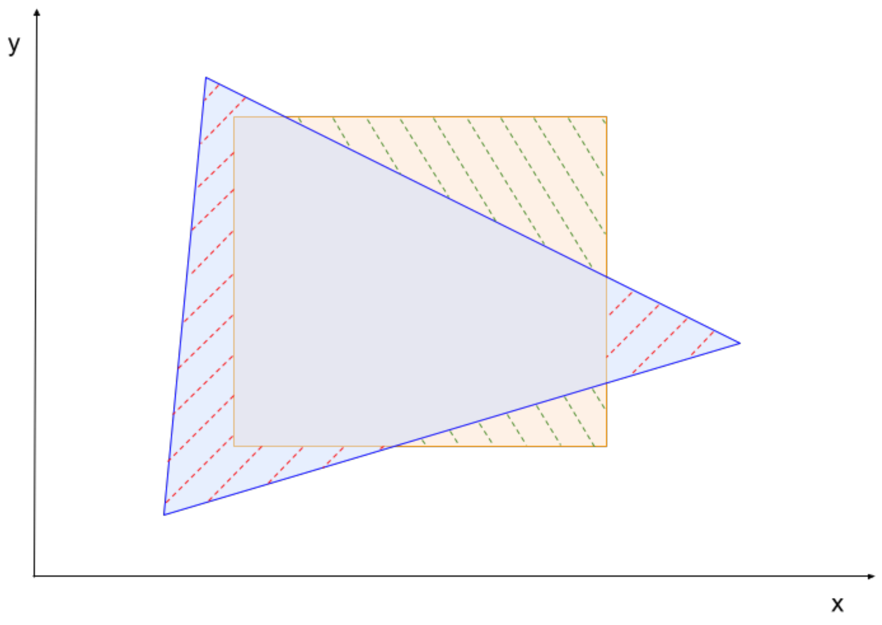


Figure 4.4: The orange and blue areas correspond to the regions satisfied by W_{θ_χ} and χ respectively. The red dashed area represents the false negative error, while the blue dashed area identifies the false positive error. The validation score is an estimate of the total error area.

W_{θ_χ}) and χ and denominator the volume of W'_{θ_χ}. Practically, this measure represent the portion of region unsatisfied by W_{θ_χ} that is instead covered by χ (false positive error).

2. The second measure of the score is computed as the quantity of samples in V unsatisfied by χ normalized by the dimension of V. Since V contains only positive data, this value consists in the false negative error.

Therefore, a lower score represent a less significant differences between the areas described by the the two formulas χ and W_{θ_χ} (Figure 4.4).

4.3 Application in New Training

As a result of the constraint learning process, the formula $\hat{\chi}$ covers the area estimated as safe and is unsatisfied for the region(s) estimated as unsafe. Hence, $\hat{\chi}$ defines an additional information on the environment characteristics that can be exploited by integrating it in the RL algorithm in order to improve the agent's performance (overview in Section 3.2.3).

A new agent is trained in an environment with the same safety specifications and same configuration of the previous one from which the test trajectories were used to learn the constraint theory. The strategy chosen to add the prior information to the training of the new agent is the addition of a **penalty** to the reward function whenever the robot "gets too close" to an area predicted as unsafe. In order to define if and how much the agent is approaching to an hazard, the formula is used to evaluate the robot's neighbourhood and retrieve the weight for the penalty.

4.3.1 Penalty Function

In the original model, a cost is assigned to the agent whenever it enters an area occupied by a hazard, so the behavioural policy for the next step is only determined by the actual position of the robot. To have a broader view of the space in which the agent is moving, it is convenient to consider a reasonable neighborhood $\mathcal{N}$ of the position occupied by an agent in a given step. Assigning a penalty based on the neighborhood leads to two main advantages:

- the best action for the agent is predicted not only basing on the current state but also considering the possible steps in the near future;
- the penalty score can be regularized by a weight.

A sample of neighbors can be chosen using different techniques (i.e. sample from a Gaussian distribution with center the current position of the robot, or else select a sample from a uniform distribution around the point of interest). In this case, the trajectories from the first training are used to estimate an appropriate neighborhood from which to sample. In each step of the agent's training, a set of neighbors $\mathcal{N}$ is considered by sampling from concentric circles around the current position of the robot (see Figure 4.5). Two circles at distance d_s and $d_s * 2$ are considered, where d_s corresponds to the average euclidean distance traveled by the agent between one step and another. An estimate of d_s is computed using a sample of consecutive positions occupied from the original agent during the testing phase.

The penalty function $p(\cdot)$ for a step i is computed as follows:

$$p(i) = \frac{P}{|\mathcal{N}(i)|} \sum_{(x_n,y_n)\in\mathcal{N}(i)} (x_n, y_n) \nvDash \hat{\chi} \tag{4.9}$$

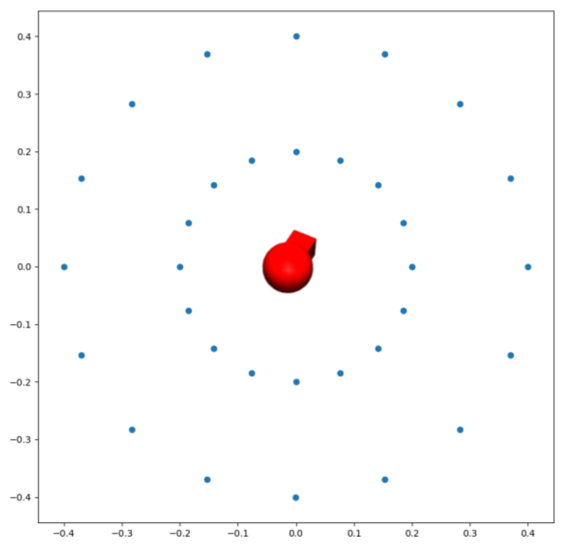


Figure 4.5: Neighborhood of robot Point, 32 samples selected from two circumferences with centre the position of the robot.

Where P is a penalty coefficient that is multiplied by the portion of neighbors in $\mathcal{N}(i)$ which do not satisfy the formula $\hat{\chi}$.

The new agent takes the original configuration with the addition of a new set of key-value assignments as input for the environment creation. The parameter P is assigned by the user as hyperparameter of the algorithm.

4.3.2 Reward Function Update

The concept of penalty function is defined as the integration of a term to the objective function, that imposes a significant penalty when constraints are violated. This approach was introduced by Fiacco and McCormick [46]. Reward-penalized functions are a way to approximate a constrained optimization problem using an unconstrained one.

Hence, in this procedure the regions identified as unsafe from the formula $\hat{\chi}$ are used to integrate a soft constraint in the form of penalty to the objective function maximized by the algorithm. For every timestep i the original reward function $r(i)$ in 4.1 is substituted by a new function $r_p(i)$ including the penalty computation $p(i)$:

$$r_p(i) = r(i) - p(i) \tag{4.10}$$

The purpose of this re-formulation is to prevent the agent to assume unsafe behaviours to increase the value of the reward function. From the start of the training, the agent is penalized not only for violations of the safety specifications but also for approaching the hazards. This means that it can rapidly learn how to move safely even without violating any safety rule.

Moreover, since the penalty is weighted by the amount of unsafe neighbors, $r_p(i)$ cannot have sudden changes between one step and another and the stability of the learning process is not affected.

The influence of $p(i)$ on the total reward function highly depends on the hyperparameter P (in Eq. 4.9). It is important to appropriately choose the value of P with respect of R_s and R_d in Eq. 4.1. The general rule is that $P < R_s \land P < R_d$, otherwise the agent would fail to achieve the goal.

Chapter 5

Experiments

A complete pipeline of the methodology was developed by using Python as programming language. A minimum of three experiments were run for each section of the thesis and evaluated quantitatively.

In this chapter the results of the experiments for every step are illustrated and discussed. In the Section 5.2 it is also presented a variation to the original constraint learning algorithm caused by an empirical evaluation.

5.1 Training of The First Agent

The environments were customized according to the purpose of the problem and all the changes are applied by modifying the files in the directory "envs" inside Safety Gym. The reinforcement learning system takes as input a configuration for the environment and task specifications in the form of key-value pairs. Some of the parameters were keep as default, while the following ones were used to create the different settings:

- *placement_extents*: spatial limits $[Xmin, Ymin, Xmax, Ymax]$ of the objects' location (i.e. goal and hazards) in the environment;
- *hazards_num*: number of hazards in the environment;
- *hazards_size*: half of the side size for each hazard;
- *hazards_locations*: locations of the hazards, randomly initialized at each training.
- *goal_size*: radius for the circular goal.

Environment Configuration and Parameters Setting Three environments were created: *Environment-1*, *Environment-2*, *Environment-3* (shown in Figure 4.1). The configuration for the environments is detailed in Table 5.1 (*Number, Size* and *Locations* refers to the hazards).

Name	Placements	Number	Size	Locations	Goal size
Environment-1	[-1.5,-1.5,1.5,1.5]	3	0.3	random	0.3
Environment-2	[-1.5,-1.5,1.5,1.5]	5	0.2	random	0.3
Environment-3	[-2.5,-2.5,2.5,2.5]	3	0.3	random	0.3

Table 5.1: Layout configuration of the environments.

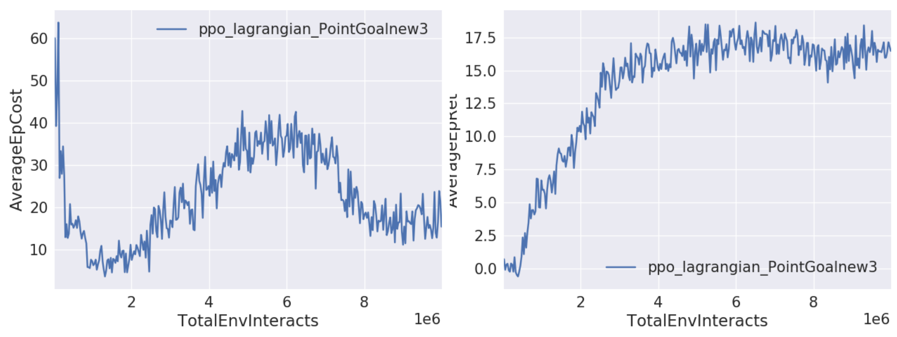
The parameters of the reward function (sparse and dense reward) were kept as default: $R_s = 1$, $R_d = 1$. Since the goal was to train an agent which respects the safety specifications as much as possible, the hazards cost parameter H_c was set to 100.

All the agents were trained for 10^7 steps, with 30000 steps for every epoch (resulting in 333 total epochs). For every epoch average metrics were stored in order to evaluate the system performances throughout the learning process. In order to reduce noise, some scores were produced by taking the average over the last five epochs of training. The main metrics are listed below:

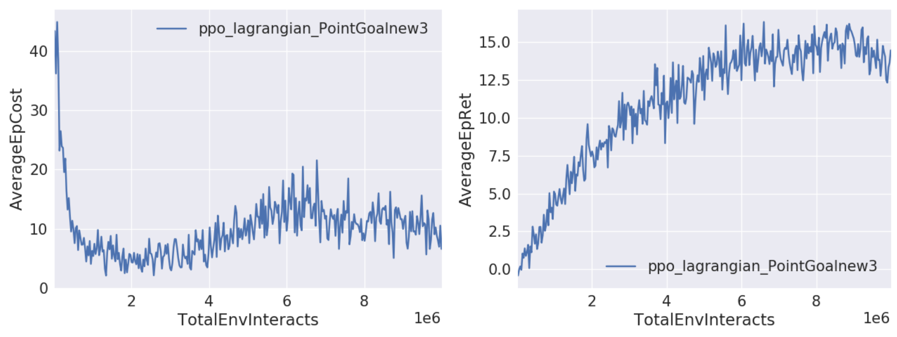
- *AverageEpCost*: normalized average cost;
- *AverageEpRet*: normalized average reward;
- *CostRate*: normalized cost rate, that is the sum of all the costs divided by the amount of interactions with the environment.

After the training, the policy is tested and a set of metrics are computed for every single episode (the metrics are described in Section 4.2.1). At the end of the testing phase of the agent the metrics were averaged for the different episodes and a metric measuring the percentage of episodes without constraint violations was also calculated (*PercentageSafe*).

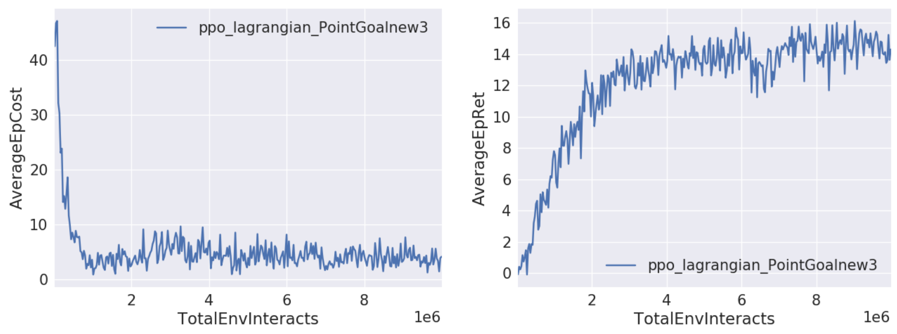
In order to define an upper boundary for the amount of safety breaches allowed, different values of limit for the constraints violation cost were tested: $z = [5,10,25]$. The different experiments were performed on *Environment-3* with the algorithm PPO-lagrangian in order to decide which z to use. The results throughout the training are shown in Figure 5.1. The results show that $z = 5$ gives the best performance in terms of safety specifications satisfaction while still guaranteeing a high reward rate.



(a) $z = 25$



(b) $z = 10$



(c) $z = 5$

Figure 5.1: Visualization of average performance in terms of average cost and reward during the training of PPO-lagrangian in Environment-3, using different values of z.

Policy Optimization Algorithm After setting all the parameters for training the RL agent, the policy optimization algorithms PPO-Lagrangian and TRPO-Lagrangian were tested using the defined configuration of the environment. In order to choose the approach that best performs in constraint satisfaction, the two algorithms were tested on three different runs of the training in the most complex environment in terms of safety specifications, *Environment-3*.

The performances achieved during the training using PPO-Lagrangian and TRPO-Lagrangian are represented in Figure 5.2.

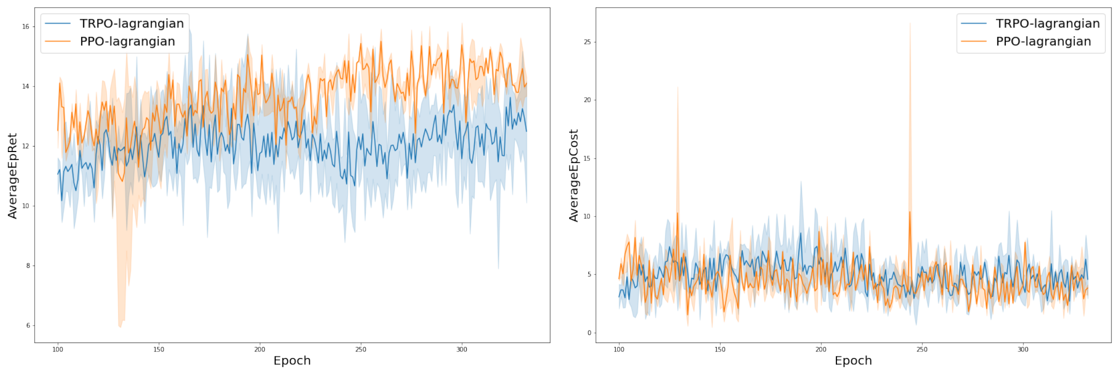


Figure 5.2: Comparison of training performances between TRPO-Lagrangian and PPO-Lagrangian. The plot represent the aggregate results of three different runs on *Environment-3* for each algorithm. For the sake of visibility the results are shown from the epoch 100.

The results in terms of cost are quite similar between the two algorithms, even if the values for PPO-Lagrangian are slightly lower on average. Instead, a consistent difference is revealed in the average reward, where PPO-Lagrangian performs substantially better, especially in the last part of the training. Thanks to this comparison, PPO-Lagrangian turned out to be more robust in respecting constraints also in concurrence with an improvement in task performance.

The test aggregated results achieved by the two policy algorithms are shown in Table 5.2, the values are averaged between three agents trained in *Environment-3* (with different hazard locations). The average scores in the test confirm that PPO-Lagrangian generally performs better.

Hence, despite the similarity of the two algorithms, the modification of the TRPO surrogate objective function applied in PPO leads to better results as well as drastically reducing the computation complexity.

Consequently, PPO-Lagrangian was used also for the experiments in *Environment-1* and *Environment-2*. The results in the different configurations were used for the constraint learning process in the next step and for the comparison with performance of the agent with prior knowledge.

Algorithm	AverageEpRet	AverageEpCost	PercentageSafe
PPO-Lagrangian	**14.072**	**4.520**	**0.685**
TRPO-Lagrangian	12.305	4.986	0.649

Table 5.2: Test performances using PPO-Lagrangian and TRPO-Lagrangian in *Environment-3*

5.2 Constraint Learning Procedure

Clean trajectories (as explained in Section 4.2.1) were stored from the testing process of the agents in the environments. In order to have a significant dataset of trajectories to sample from, each agent was tested for 500 episodes with 1000 steps each. To generate a complete dataset within a limited size, the positions of the agent were stored every five steps. The constraint learning procedure takes several parameters as input:

- μ_0: initial threshold multiplier;
- N: number of iterations of the algorithm;
- *timeout*: time limit for every iteration of the algorithm;
- m: constant value for μ updating;
- S: training sample dimension, selected from the trajectories data within the region delimited by *placement_extents*
- V: validation sample dimension, selected from the trajectories data within the region delimited by *placement_extents*
- k, h, w_k, w_h: initial values of k and h and the corresponding weights for minimization problem in 4.8;
- α: tolerance of the algorithm to errors.
- *nb*: coefficient to compute the number of negative example generated, the value is expressed as a float.

Parameters Setting All the experiments were run locally, so the parameters *hops* and *timeout* were set considering the limitations in terms of time and computational resources.

The target of the algorithm is to retrieve a formula which is as precise as possible in identifying harmful areas. Hence, the initial value of the threshold multiplier μ_0 was set to 1.5, value chosen because considered quite reasonable. In fact, a value of $\theta_0 = \mu_0 \cdot d$ close to the distance measure d (Eq. 4.5), calculated from the training data, allows to define a nearly accurate region by the union of the training data bounding boxes B_θ and to consequently identify the unsafe regions (even in the most complex environment). If a formula with θ_0 was found by the algorithm, θ_0 becomes an upper bound for the threshold value, else becoming the lower bound (Eq. 4.7).

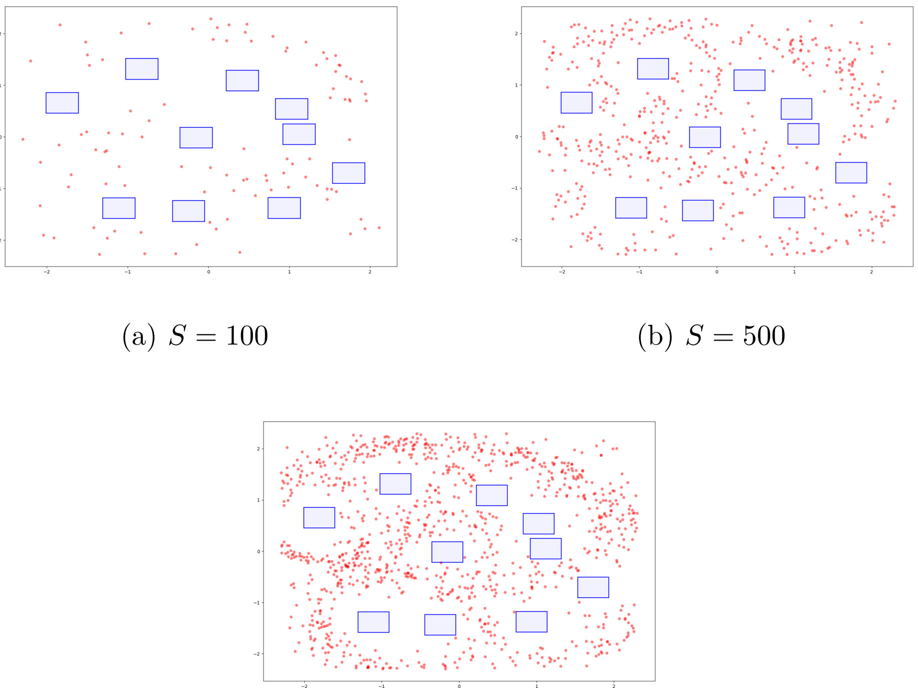
The value of m has been set to 2 so as not to have too drastic variations of the multiplier μ_k, since the range of vaules should be around d in this application.

The initialization of k and h was set considering the use case and the type of constraints that the algorithm should learn. The environments contain hazards, which are closed areas with square shape, therefore, to represent the region occupied by one hazard, a clause with four inequalities is needed. The initial values were set considering the minimum complexity needed to represent at least one hazard: $k = 1$ and $h = 4$. For a similar reasoning, it was considered that the algorithm must scale with the increase in the number of harmful zones, but this increase is not linear between the number of clauses and inequalities (as mentioned above). This imbalance was reproduced by giving different weights to k and h, keeping the weight of k greater so as to have a slower increase. Since the goal is to approximate the hazard zones from a non-uniform data set, w_k was assigned the value 3 (instead of the target value 4) and $w_h = 1$. In this way, an increase $3 : 1$ of k and h was considered acceptable because it still allows the identification of k closed areas in space.

The coefficient nb defines the amount NB of negative instances inserted in the original dataset, computed as $NB = nb * S$ (being X the original set of trajectories). Since the negative examples are bootstrapped from a set of data, it is not possible to predict a priori the number of negative samples generated. The value of nb was set to 1 in order to produce an amount of estimated negative instances which is significant but does not exceed the quantity of positive data.

The sample size S had to be large enough to distinguish safe and unsafe areas, the choice depended on the dimension of the environment's window and of the hazards. An estimate of the minimum amount of datapoints needed was made empirically (example in Figure 5.3). The following values were chosen as lower bound S_m for dimension of S in the different environments:

- *Environment-1*: $S_m = 500$
- *Environment-2*: $S_m = 800$
- *Environment-3*: $S_m = 1000$



(a) $S = 100$ (b) $S = 500$

(c) $S = 1000$

Figure 5.3: Visualization of samples from trajectories in *Environment-3* with increasing S.

A lower bound for the dataset size was set since a large amount of samples would significantly increase the number of operations to be performed by the solver, consequently, leading to a greater risk of not finding a valid solution within the timeout.

The maximum time interval for the constraint learning process was set to 10000 seconds ($timeout = 1000$ and $N = 10$). The tolerance to errors was fixed to $\alpha = 0.2$ in order to add some flexibility to the algorithm while ensuring that the solution is not compromised and can be considered valid.

In every environment, experiments were run by starting from the defined minimal sample size S_m and then increasing the amount of samples. The constraint learning procedure returned a set of formulas (with size up to the number of iterations N) and then $\hat{\chi}$ was chosen by the validation process. The validation set was sampled from the original trajectories dataset and the size was set as: $V = 2S$. Since the validation data were used to construct an estimate of the true support to be used as a target for the learning process, a greater amount of data was considered to ensure a better representation.

Evaluation of results In order to evaluate the performance in learning constraints and to decide if the procedure was appropriate or modifications were needed, the results were compared to the ground truth. To define a measure, a set of 50000 examples within the environments space limits were sampled and classified by the learned formula $\hat{\chi}$ and by the ground truth χ^* (constructed by using the *hazards_locations* of the environment). The metric used to evaluate the classification performance of $\hat{\chi}$ was the *F1-score*, calculated as:

$$F1(c) = 2 * \frac{precision(c) * recall(c)}{precision(c) + recall(c)}$$
$$\textit{F1-score} = \frac{F1(Pos) + F1(Neg)}{2} \tag{5.1}$$

The result is an arithmetic mean between the F1 scores for the positive class *Pos* and the negative class *Neg*. This metric was chosen since it gives a "fair" score when the dataset imbalanced (like in this case were the positive class is significantly more represented).

Table 5.3 reports the results achieved by the learned formula in the three environments. The performances are not very satisfying in terms of F1 score, especially for *Environment-3*, were the learned formulas could not identify the unsafe regions properly. Constraint learning failed whenever there was an increase of hazards in the environment configuration (see Figure 5.4). In many cases, the formulae learned for the agents trained in *Environment-3* fail to identify even one of the harmful areas. This is due to the fact that in the learning process the algorithm cannot scale fast enough to reach the complexity (in terms of k and h) necessary to represent 10 closed zones, the hazards. In *Environment-3* the target values of k and h are $k = 10$ and $h = 4k$ (since the unsafe zones are square), so the formula should reach a similar complexity in order to represent the hazards fairly accurately.

Even when adding samples to the training dataset, the algorithm could not improve its performance, indeed in some cases the results even worsened

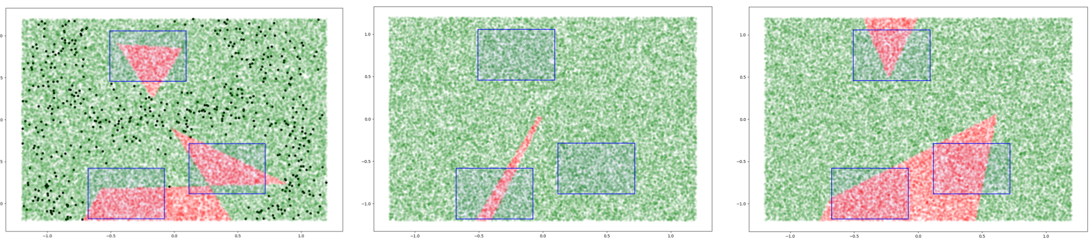
Environment-1			
Experiment	F1, $S = 500$	F1, $S = 800$	F1,$S = 1000$
Point1-env1	0.760	0.501	0.748
Point2-env1	0.653	0.820	0.776
Point3-env1	0.653	0.601	0.785
Statistics	$\mu = 0.689$ $\sigma = 0.050$	$\mu = 0.641$ $\sigma = 0.133$	$\mu = 0.770$ $\sigma = 0.016$
Environment-2			
Experiment	F1, $S = 800$	F1, $S = 1000$	F1,$S = 1500$
Point1-env2	0.608	0.518	0.672
Point2-env2	0.537	0.609	0.607
Point3-env2	0.573	0.511	0.469
Statistics	$\mu = 0.573$ $\sigma = 0.029$	$\mu = 0.546$ $\sigma = 0.045$	$\mu = 0.592$ $\sigma = 0.073$
Environment-3			
Experiment	F1, $S = 1000$	F1, $S = 1500$	F1,$S = 2000$
Point1-env3	0.481	0.480	0.597
Point2-env3	0.475	0.480	0.493
Point3-env3	0.486	0.480	0.480
Statistics	$\mu = 0.481$ $\sigma = 0.005$	$\mu = 480$ $\sigma = 0.000$	$\mu = 0.523$ $\sigma = 0.052$

Table 5.3: Results in terms of F1-score of the constraint learning procedure. The experiments were run with the same set of parameters except from the sample dimension S. Aggregated statistics (mean and standard deviation) were calculated for the results in experiments with the same configuration.

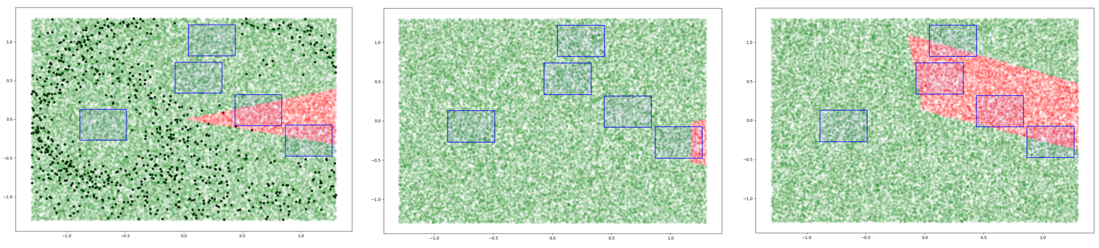
due to the increase of computational complexity. Moreover the system was not considered very reliable due to the variability of performance between the trajectories in the same type of environment and with the same set of parameters (only variability is the random position of the hazards).

5.2.1 Variation

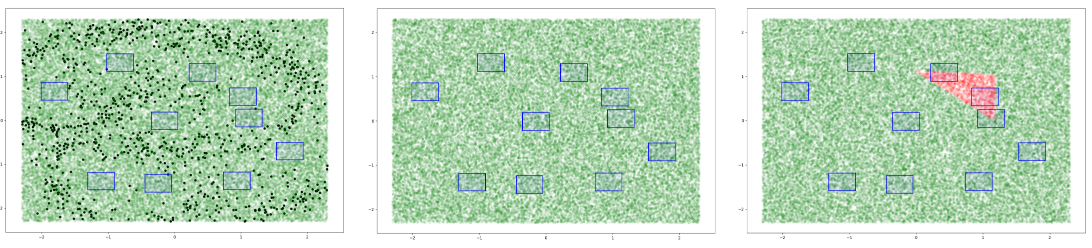
The results achieved by applying the constraint learning procedure were not accurate enough to be used for representing the unsafe areas of the environment. To understand how to variate the procedure in order to get a formula that identifies the unsafe regions with acceptable accuracy, the limitations of



(a) *Environment-1*: $S = 500,\ S = 800,\ S = 1000$



(b) *Environment-2*: $S = 800,\ S = 1000,\ S = 1500$



(c) *Environment-3*: $S = 1000,\ S = 1500,\ S = 2000$

Figure 5.4: Visualization of the formula $\hat{\chi}$ retrieved by the constraint learning procedure in different environments with an increasing amount of training data. The red areas represent the regions classified as unsafe from $\hat{\chi}$. For the sake of visibility the training data are shown (in black) only for the results with the minimum amount of samples S.

the constraint learning process were analysed in a simplified example case.

The sample S was randomly selected from datapoints uniformly distributed in the safe zones of a test environment with a configurable number of hazards. The problems identified in the learning process were the following:

- the algorithm failed with an increasing number of hazards with a proportionally small area with respect to the spatial window considered;
- when adding more samples to the training data, the execution time of the algorithm increased consistently and in many cases the algorithm

was unable to correctly classify the desired portion $(1 - \alpha)S$ of training data.

Therefore, the test environment was created by using a low number of hazards in a small space and reducing the number of training samples. All the parameters of the algorithm were kept as before. The visualization of the formulas learned in the simplified environment (in Figure 5.5) shows an improvement in the performance of the constraint learning procedure.

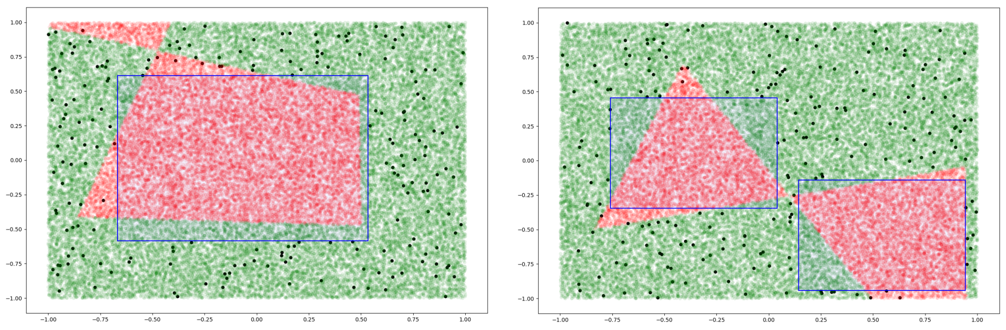


Figure 5.5: Visualization of the constraint learning results in two simplified environments with 200 data uniformly distributed within the safe region. The learned formulas can quite accurately distinguish safe and unsafe areas (F1-score ≈ 0.87).

These results led to the conclusion that, by decreasing the number of samples and increasing the area occupied by unsafe areas in proportion to the total space, it was possible to learn acceptable constraint theories. This reasoning was transferred to learning from the trajectories of the agent by creating a series of subsets starting from the original data. The idea was to define sub-areas smaller than the original space (delimited by *placement_extents)* and individually learn the constraints using training data sampled from the trajectories within the created sub-area. The final formula is given by the union of the formulas learned in the individual created regions. The procedure for the generation of the sub-areas from the original space is detailed in 4.

Given a desired minimal dimension for the new areas side ($dimension$), every variables' domain interval is divided in M sub-intervals, where M corresponds to the integer division between the original interval length and $dimension$. The final set of intervals $subsets$, consisting in (sub_x, sub_y) pairs, is used to define M sub-areas A from the original space.

Algorithm 4 Generation of multiple sets of training data to learn constraints in sub-areas of the original data space.

```
1: procedure CreateSubsets($X, dimension, placement\_extents$)
2:     $ls \leftarrow$ variables $(x, y)$ intervals from $placement\_extents$
3:     for $l_v$ in $ls$ do
4:         $M \leftarrow l_v \div dimension$
5:         if $M == 0$ then
6:             $sub_v \leftarrow l_v$
7:         else
8:             $sub_v \leftarrow M$ equal intervals from $l_v$
9:     $subsets \leftarrow$ set of $length(sub_x) * length(sub_y)$ intervals
10:    return $subsets$
```

The constraint learning procedure LearnSupportsAdaptive is iteratively executed for every $a_j \in A$ (where $j = 1, \ldots, M$) using as dataset the set of agent's trajectories in the area of a_j. Every iteration of LearnSupportsAdaptive returns a formula $\hat{\chi_j}$ for the sub-area a_j, the global formula $\hat{\chi}$ is given by the union of the M formulas generated:

$$\hat{\chi} = \bigcup_{j=1}^{M} \hat{\chi_j} \tag{5.2}$$

5.2.2 Results

Considering the dimension of the original environments' space in this task, the minimum side dimension was set to 1 ($dimension = 1$). From the experiments in the test environments it was found that the algorithm was able to scale correctly with a number of samples equal to 200. This information was used to set $S_j = 200$ for each a_j. Hence, the amount of trajectories subsets and of total training samples for each environment was:

- *Environment-1*: $M = 4, S = 800$;
- *Environment-2*: $M = 4, S = 800$;
- *Environment-3*: $M = 16, S = 3200$;

The amount of training samples for *Environment-1* and *Environment-2* were consistent with the evaluation done before about the minimum number

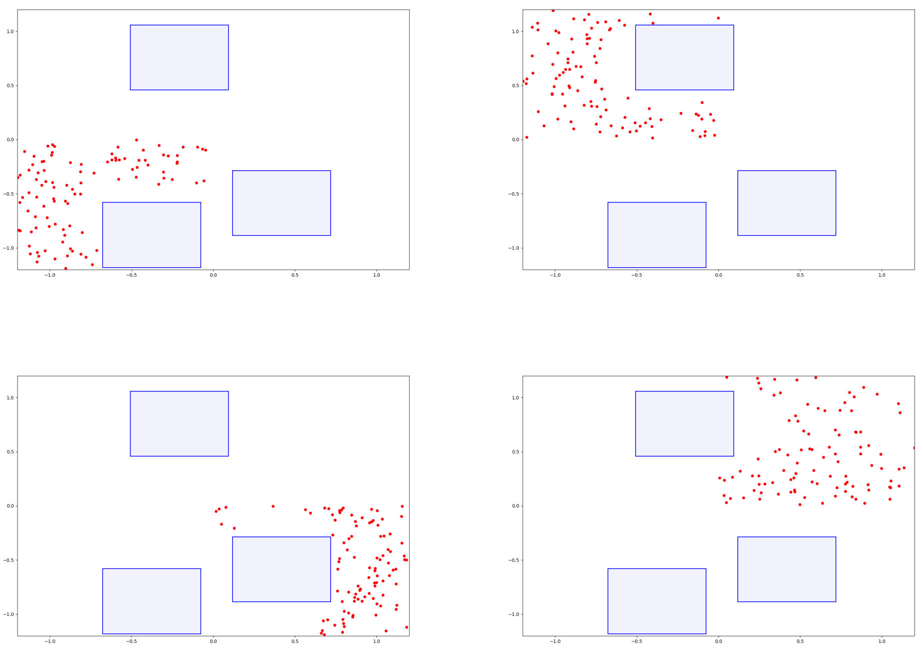


Figure 5.6: Example of sample selection from M sub-areas generated in *Environment-1*.

of samples S_m. For *Environment-3* S was significantly greater with respect to the first experiments and it was considered an advantage given the unsatisfactory results achieved in precedence.

The time limit was also modified since the procedure became iterative, so potentially M times slower. The experiments in the test environments showed that the algorithm could find valid solutions in very short time (a few seconds). For this reason *timeout* was set to 180 and the number of iterations was set to 8. In this way the time limit for the first two environments was almost halved with respect to the previous experiments. The maximum time interval for *Environment-3* increased, but from the experiments revealed that the effective execution time decreased from the first experiments in the same environment.

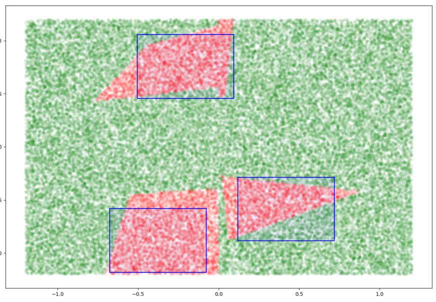
Table 5.4 reports the results achieved by the iterative version of the constraint learning procedure. All the parameters except from the ones regarding amount of training samples and time limit were kept as in the first set of experiments. The results in terms of F1-score were significantly improved with respect to the ones of the original constraint learning procedure. On average there was an increase of 0.21 on the F1-score. The improvement was especially seen in *Environment-3* were the original method hardly ever succeeded in producing a formula that would even partially identify unsafe zones. Moreover the performance of the new learning model were more stable, in fact the standard deviation for the values of F1 scores was lowered, never exceeding 0.02.

Environment-1			
Experiment	F1 score	Recall neg.	Precision neg.
Point1-env1	0.821	0.755	0.701
Point2-env1	0.861	0.810	0.732
Point3-env1	0.863	0.778	0.769
Statistics	$\mu =$ **0.848** $\sigma = 0.019$	$\mu = 0.781$ $\sigma = 0.022$	$\mu = 0.734$ $\sigma = 0.028$
Environment-2			
Experiment	F1 score	Recall neg.	Precision neg.
Point1-env2	0.752	0.755	0.636
Point2-env2	0.799	0.702	0.655
Point3-env2	0.762	0.710	0.645
Statistics	$\mu =$ **0.771** $\sigma = 0.020$	$\mu = 0.722$ $\sigma = 0.023$	$\mu = 0.645$ $\sigma = 0.008$
Environment-3			
Experiment	F1 score	Recall neg.	Precision neg.
Point1-env3	0.743	0.789	0.523
Point2-env3	0.774	0.870	0.551
Point3-env3	0.789	0.881	0.567
Statistics	$\mu =$ **0.769** $\sigma = 0.019$	$\mu = 0.847$ $\sigma = 0.041$	$\mu = 0.547$ $\sigma = 0.018$

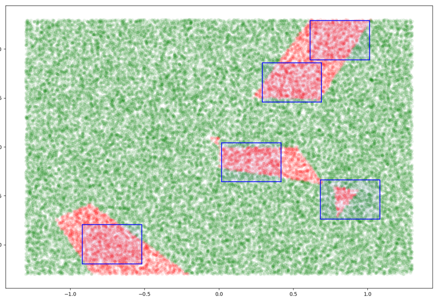
Table 5.4: Results in terms of global F1-score and negative class recall and precision achieved by the execution of the constraint learning procedure iterative variation. Aggregated statistics (mean and standard deviation) were calculated for the results in experiments with the same configuration.

In the table, recall and precision scores are reported for the negative class. The values for the precision were quite lower than the recall scores, especially when increasing the number of hazards in the environment. This indicates that the regions identified as unsafe by the formula were a bit wider than the original ones and probably not completely centered in the position of the hazards. This result was due to the non-uniform distribution of the training data which did not allow to have a perfect representation of the safe areas.

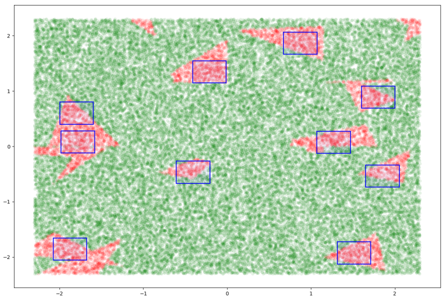
Considering the training data quality, the restrictions in terms of time and computational resources, the results of the constraint learning procedure using subsets were evaluated to be satisfactory. A visualization of the predicted constraint theories in different environments is provided in 5.7.



(a) *Environment-1*



(b) *Environment-2*



(c) *Environment-3*

Figure 5.7: Formulae $\hat{\chi}$ learned in three different environments using the iterative variation of the constraint learning procedure.

5.3 Training of The Agent with Prior Knowledge

For the second agent the environment's configuration was kept identical to the original one, in terms of layout and coefficients for cost and reward, to have a fair comparison. The penalty coefficient P was instead fixed by considering the reward parameter. In order to have a significant contribution of the penalty but still allow the agent to learn to achieve the goal, P was set to $\frac{1}{10}R_s$, resulting in $P = 0.1$.

The training time of an epoch for the second agent, with penalty computation for each step, only had a small increase (between $10 - 15\%$). This was possible due to an evaluation in batch of the neighborhood datapoints which mapped them to satisfied or unsatisfied according to the predicted formula $\hat{\chi}$. Since the addition of previous information was expected to lead to faster training of the agent, the number of epochs, and therefore of interactions with the environment, have been reduced in half compared to the first agent.

All the environments created in the first RL experiments were replicated to train new agents with prior knowledge. The goal was to improve the agent's ability to avoid harmful areas. Aggregated statistics about the "safety" of the agent's exploration while testing were used, in order to compare the performance of the agent before and after the addition of prior knowledge. In particular, the measures used as metrics were:

- *AverageTestCost*: average cumulative episodic cost;
- *PercentageSafe*: percentage of episodes with safe trajectories.

5.3.1 Results

The safety results achieved by the agents with prior knowledge and the comparison with the first experiments are reported in Table 5.5. The results improved both in terms of average episodic cost and percentage of zero-cost episodes in all the environments. It can be observed that the average costs are quite uniform among the different environments and, especially, they are decidedly lower than the safety limit set ($z = 5$). The average percentage difference in cost assigned to the agents with prior knowledge was over -30% for *Environment-3* and around -50% for the other two.

In order to analyse the training process and establish if the agents were also learning how to reach the goal in addition to avoiding harmful areas, graph showing the cost and reward values trends during the agent's training were inspected (Figure 5.8). The graphs represent the performance of the RL training in the three environments with aggregate values of the experiments.

The cost trend was lower on average for agents with prior knowledge throughout the training. In all the environments the average cost for the new agents never exceeded the safety limit z after the first 25 epochs, while

Environment-1				
	Prior Knowledge Agent		Results Comparison	
Experiment	AvgTestCost	%Safe	ΔAvgTestCost	Δ%Safe
Point1-env1	4.016	79.4	-1.517	11.5
Point2-env1	3.466	82.8	-3.766	19.4
Point3-env1	1.510	90.2	-2.200	12.8
Average	2.997	84.1	**-2.494**	**14.6**
Environment-2				
	Prior Knowledge Agent		Results Comparison	
Experiment	AvgTestCost	%Safe	ΔAvgTestCost	Δ%Safe
Point1-env2	2.836	79.8	-1.714	7.8
Point2-env2	3.298	76.6	-1.838	15.8
Point3-env2	2.992	79.4	-5.614	23.6
Average	3.042	78.6	**-3.055**	**15.7**
Environment-3				
	Prior Knowledge Agent		Results Comparison	
Experiment	AvgTestCost	%Safe	ΔAvgTestCost	Δ%Safe
Point1-env3	3.248	79.2	-1.676	7.1
Point2-env3	3.856	77.0	-1.724	8.2
Point3-env3	2.842	80.4	-1.830	10.8
Average	3.315	78.9	**-1.743**	**8.7**

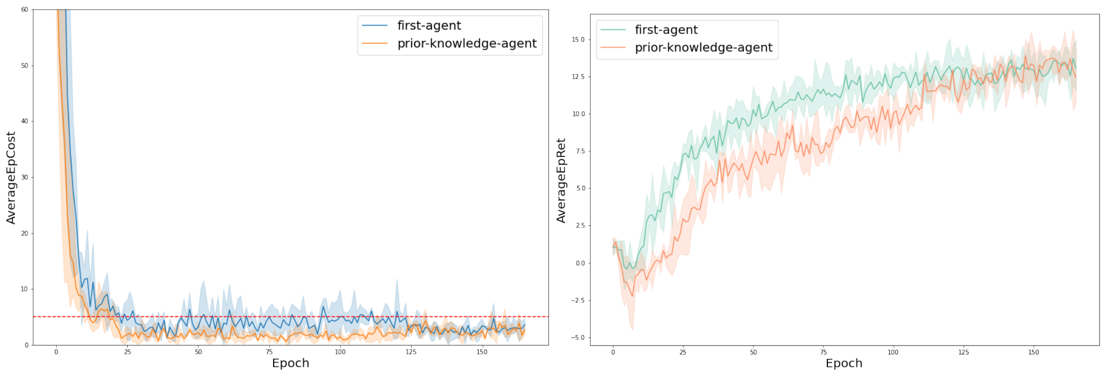
Table 5.5: Performance of the agent with prior knowledge in respecting safety specifications. The results were averaged throughout 500 test episodes and compared with the ones achieved by the first experiments without the integration of prior knowledge.

in the previous experiments the costs were more oscillatory and sometimes over z also at the end of the training. The greatest decrease in terms of costs was found in *Environment-2*, where the original agents were unable to comply with the safety specifications, probably due to the high concentration of hazards in space.

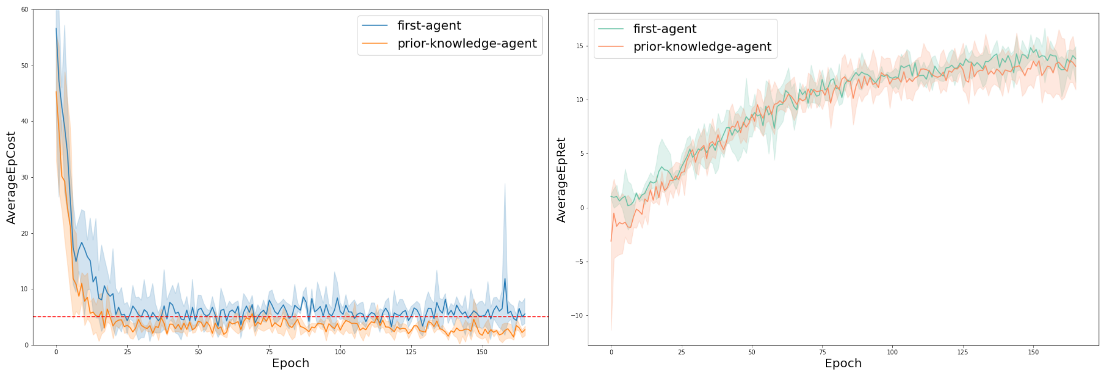
While satisfaction of safety constraints has been significantly improved, agents have not lost the ability to reach the goal. The reward values achieved by adding prior knowledge are on average very similar to those of the original training. In *Environment-3* the addition of the penalty function, even made the agent's learning more stable throughout the training, thus increasing the reliability of the RL model.

Some experiments were run also by increasing the value of P to 0.2 or 0.5. The results showed that with these penalty values the agent was learning how to avoid the unsafe areas even better, but the high negative contribution to the objective function was preventing the agent from achieving the goal and it was reducing the training stability. A solution to this could be the application of a non-constant penalty coefficient which adapts to the agent's current state and performance.

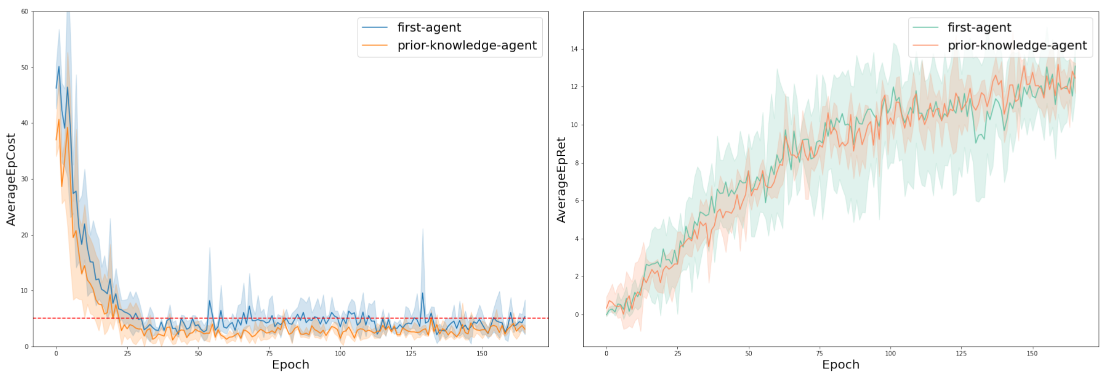
Hence, the addition of a penalty function, based on the learned formula's predictions regarding the safety of the environment, brought significant improvements in the safe exploration task. The safety requirements were on average always satisfied, even in environments with a greater amount of harmful areas where the original RL agents could not learn safe enough behaviours. In addition to this training time was also reduced by almost half with comparable results in terms of reaching the task goal but with a much higher level of safety.



(a) *Environment-1*



(b) *Environment-2*



(c) *Environment-3*

Figure 5.8: Training performances of the original agent compared to the agent with prior knowledge. On the left there are the average costs per epoch and the red dashed line indicates the limit boundary fixed in the constraints specification. The plots on the right represent the average reward achieved throughout the training.

Chapter 6

Conclusion

Given a set of two-dimensional data sampled from the trajectories of an expert reinforcement learning agent, a method was implemented to learn a formula which approximates the support of the training data. The learned constraint theory was used to predict the unsafe areas to be avoided by the agent and to add a penalty to the objective function of the RL system.

An adequate learning of the formula was guaranteed even in the environments with numerous hazards thanks to the partition of the constraint learning process into subproblems considering data belonging to sub-areas of the environment. This variation to the original procedure added scalability to the process, also with the possibility to indicate the extent of dividing the data for greater precision. Some further additions and/or adjustments could be applied in real world applications:

- The use of more powerful computational resources could lead to even more precise results than the ones achieved in this thesis, since the solver would scale faster to more complex representations.
- The constraint learning processes on the data subsets could be parallelized since the final constraint formulation consists in the union of the singular formulas.
- A more trustworthy source providing the trajectories could lead to a more uniform and representative training dataset. An example is the use of human safe demonstrations to create or integrate the current dataset (human experience was already used to add information to a RL system and consequently improve its performance [47]).

This kind of solutions would lead to results compatible with use in a real case with stringent security needs.

In the last step of the methodology the learned formula was used to identify the portion of unsafe neighbors of the robot in the current position throughout the training process. The penalty coefficient was passed as a static hyperparameter of the RL algorithm. As a future alternative, it would be possible to 1) make this coefficient as a learnable parameter during the training 2) or else apply an updating function which, for example, increases the coefficient value after a predefined number of interactions or after reaching a certain level of safety learning by the agent. These strategies could bring even better performance in terms of safety specifications satisfaction by the agent trained with prior knowledge.

The implemented method was meant to increase the safety level of an agent trained by constrained reinforcement learning. There are many real applications that could benefit from this kind of solution. For instance, a company could decide to integrate the production process with autonomous robots which have to perform locomotive tasks. The robots must be able to navigate obstacles while moving from a location to another. In this case it would be necessary that the autonomous agent is trained so that it will avoid risky areas with a very high confidence.

High safety requirements are present in every system that has an objective of being an autonomous device. In order to be applicable in real life, these models are integrated with experience, additional hand-made rules and/or adjusted data from similar applications. This is the case of self-driving cars, where additional information from experimental data or from autonomous systems in different domains with similar safety concerns can be crucial to ensure a safe behaviour of the vehicles [48].

Bibliography

[1] K. Wang and W. Sun, “Meta-modeling game for deriving theory-consistent, microstructure-based traction–separation laws via deep reinforcement learning,” *Computer Methods in Applied Mechanics and Engineering*, vol. 346, pp. 216–241, 2019.

[2] D. L. Moreno, C. V. Regueiro, R. Iglesias, and S. Barro, “Using prior knowledge to improve reinforcement learning in mobile robotics,” 2004.

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

[4] J. Garcia and F. Fernandez, “Safe exploration of state and action spaces in reinforcement learning,” *Journal of Artificial Intelligence Research*, vol. 45, pp. 515–564, dec 2012.

[5] A. Hans, D. Schneegass, A. Schäfer, and S. Udluft, “Safe exploration for reinforcement learning,” pp. 143–148, 01 2008.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[7] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, “A survey of deep reinforcement learning in video games,” *CoRR*, vol. abs/1912.10944, 2019.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.

[9] M. Coggan, “Exploration and exploitation in reinforcement learning,” *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[10] J. Choi, G. Lee, and C. Lee, “Reinforcement learning-based dynamic obstacle avoidance and integration of path planning,” *Intelligent Service Robotics*, vol. 14, pp. 1–15, 11 2021.

[11] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. K. Yogamani, and P. Pérez, “Deep reinforcement learning for autonomous driving: A survey,” *CoRR*, vol. abs/2002.00444, 2020.

[12] A. Bhatia, P. Varakantham, and A. Kumar, "Resource constrained deep reinforcement learning," *CoRR*, vol. abs/1812.00600, 2018.

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

[14] Y. Zhang, Q. Vuong, and K. W. Ross, "First order optimization in policy space for constrained deep reinforcement learning," *CoRR*, vol. abs/2002.06506, 2020.

[15] L. D. Raedt, A. Passerini, and S. Teso, "Learning constraints from examples," in *AAAI*, 2018.

[16] L. G. Valiant, "A theory of the learnable," in *STOC '84*, 1984.

[17] S. Muggleton and L. de Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19-20, pp. 629–679, 1994. Special Issue: Ten Years of Logic Programming.

[18] N. Beldiceanu and H. Simonis, "A model seeker: Extracting global constraint models from positive examples," in *Principles and Practice of Constraint Programming* (M. Milano, ed.), (Berlin, Heidelberg), pp. 141–157, Springer Berlin Heidelberg, 2012.

[19] S. Paramonov, S. Kolb, T. Guns, and L. De Raedt, "Tacle: Learning constraints in tabular data," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, (New York, NY, USA), p. 2511–2514, Association for Computing Machinery, 2017.

[20] Y. Guo, R. Reichart, and A. Korhonen, "Unsupervised declarative knowledge induction for constraint-based learning of information structure in scientific documents," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 131–143, 2015.

[21] F. Yang, Z. Yang, and W. W. Cohen, "Differentiable learning of logical rules for knowledge base completion," *CoRR*, vol. abs/1702.08367, 2017.

[22] A. Meló and H. Paulheim, "Automatic detection of relation assertion errors and induction of relation constraints," *Semantic Web*, vol. 11, pp. 1–30, 04 2020.

[23] G. Murphy and P. Allopenna, "The locus of knowledge effects in concept learning," *Journal of Experimental Psychology: Learning Memory and Cognition*, vol. 20, pp. 904–919, July 1994. Copyright: Copyright 2018 Elsevier B.V., All rights reserved.

[24] J. Tenenbaum, T. Griffiths, and C. Kemp, "Theory-based bayesian models of inductive learning and reasoning," *Trends in cognitive sciences*, vol. 10, pp. 309–18, 08 2006.

[25] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach,

R. Heese, B. Kirsch, M. Walczak, J. Pfrommer, A. Pick, R. Ramamurthy, J. Garcke, C. Bauckhage, and J. Schuecker, “Informed machine learning - a taxonomy and survey of integrating prior knowledge into learning systems,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021.

[26] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, “A survey on deep transfer learning,” 2018.

[27] A. Borghesi, F. Baldo, and M. Milano, “Improving deep learning models via constraint-based domain knowledge: a brief survey,” 2020.

[28] S. Roychowdhury, M. Diligenti, and M. Gori, “Regularizing deep networks with prior knowledge: A constraint-based approach,” *Knowledge-Based Systems*, vol. 222, p. 106989, 04 2021.

[29] L. Rieger, C. Singh, W. J. Murdoch, and B. Yu, “Interpretations are useful: penalizing explanations to align neural networks with prior knowledge,” *CoRR*, vol. abs/1909.13584, 2019.

[30] R. Stewart and S. Ermon, “Label-free supervision of neural networks with physics and domain knowledge,” *CoRR*, vol. abs/1609.05566, 2016.

[31] G. Borboudakis and I. Tsamardinos, “Incorporating causal prior knowledge as path-constraints in bayesian networks and maximal ancestral graphs,” in *ICML*, 2012.

[32] M. Sinha and S. Ramsey, “Using a general prior knowledge graph to improve data-driven causal network learning,” 03 2021.

[33] M. Zheng and S. Kleinberg, “Using domain knowledge to overcome latent variables in causal inference from time series,” *Proceedings of machine learning research*, vol. 106, pp. 474–489, 2019.

[34] A. Silva and M. Gombolay, “Neural-encoding human experts’ domain knowledge to warm start reinforcement learning,” 2020.

[35] B. Argall, S. Chernova, M. M. Veloso, and B. Browning, “A survey of robot learning from demonstration,” *Robotics Auton. Syst.*, vol. 57, pp. 469–483, 2009.

[36] S. Badreddine and M. Spranger, “Injecting prior knowledge for transfer learning into reinforcement learning algorithms using logic tensor networks,” 2019.

[37] H.-L. Hsu, Q. Huang, and S. Ha, “Improving safety in deep reinforcement learning using unsupervised action planning,” 2021.

[38] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, “Robust adversarial reinforcement learning,” *CoRR*, vol. abs/1703.02702, 2017.

[39] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control.,” in *IROS*, pp. 5026–5033, IEEE, 2012.

[40] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," *CoRR*, vol. abs/1705.10528, 2017.

[41] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[42] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015.

[43] P. Morettin, S. Kolb, S. Teso, and A. Passerini, "Learning weighted model integration distributions," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 5224–5231, Apr. 2020.

[44] S. Kolb, S. Teso, A. Passerini, and L. D. Raedt, "Learning smt(lra) constraints using smt solvers," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 2333–2340, International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[45] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The math sat 5 smt solver," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, vol. 7795, pp. 95–109, 01 2013.

[46] A. V. Fiacco and G. P. McCormick, "Nonlinear programming;: Sequential unconstrained minimization techniques," 1968.

[47] M. E. Taylor, H. B. Suay, and S. Chernova, "Integrating reinforcement learning with human demonstrations of varying ability," in *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '11, (Richland, SC), p. 617–624, International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[48] P. Koopman, U. Ferrell, F. Fratrik, and M. Wagner, *A Safety Standard Approach for Fully Autonomous Vehicles*, pp. 326–332. 08 2019.