POLITECNICO DI TORINO

Master's Degree in Aerospace Engineering

Master's Degree Thesis

Safe Exploration with Safety Layer and reward shaping

Supervisor

Prof. Manuela BATTIPEDE

Tutor

Andrea LONZA

Ing. Enrico BUSTO

Candidate

Alessia BASLER

April 2021

Abstract

The purpose of this Master Thesis is to investigate and improve one of the state-of-the-art Safe Reinforcement Learning algorithms. The studied algorithm consists in the application of a Safety Layer to classical Reinforcement Learning algorithms in order to accomplish a Safe Exploration during learning phases, that would open up the doors of real-world training to intelligent agents.

Safety Layer algorithm shows good performances in environments where the danger is located on the edges, but worsens when used in environments where the hazards permeate the space in an heterogeneous way. To improve the performances in such peculiar situations, reward shaping has been introduced, in order to reinforce the safety action of Safety Layer.

In the first chapters of the thesis an introduction to Artificial Intelligence, Deep Neural Networks, classic and Deep Reinforcement Learning will be presented. This aims to make the reader familiar with the main topics and algorithms that will be probed and deeply analyzed later in this document.

A chapter is dedicated to the explanation of Safe Reinforcement Learning process, in what it differs from classic Reinforcement Learning, its goals and main challenges.

The last chapters will treat the implementation of the chosen algorithm and experiments results, with an eye towards the issues encountered and the solutions proposed. In these chapters it will be also inserted a presentation of the technical features inherent to the experiments performed.

Eventually, some conclusions will be deduced about the improvements obtained, showing that reinforcing the Safety Layer action with reward shaping helps to achieve Safe Exploration in environments with heterogeneous danger distributions, that are more plausible representations of real-world.

Table of Contents

List of Tables

List of Figures

Acronyms

AI	Artificial Intelligence
CMDP	Constrained Markov Decision Process
CPO	Constrained Policy Optimization
DDPG	Deep Deterministic Policy Gradient
DDN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GPI	General Policy Iteration
MC	Monte-Carlo
MDP	Markov Decision Process
ML	Machine Learning
MSE	Mean Squared Error
NN	Neural Network
QP	Quadratic Program
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
SL	Safety Layer
SRL	Safe Reinforcement Learning
TD	Temporal-Differences

Chapter 1

Introduction

During the last century, the world has been witnessing a techonlogical development never seen in its history. Medicine, engineering, biology, physics and so on, every aspect of human knowledge has improved, enjoying of progress that scientific community has performed and shared with the whole world.

In particular, mathematics and computer science combined their efforts on a specific research field, which showed its potentialities already in early 90's and experienced an exponential growth in interest since then: **Artificial Intelligence** (AI). AI is currently used in a variety of real-world applications: from face recognition in social media to incredibly efficient online translators, from computer game bots to ad-personam e-commerce advertisements, from robots playing football to cybersecurity.

In this introduction a brief story of AI will be presented, focusing on the state-of-the-art reached and one of the most modern approaches developed, that will be covered in this thesis: **Reinforcement Learning** (RL).

1.1 Pigmalione and his offspring

The very first time that AI has been conceived, it can be associated with Greek mythology. A sculptor named Pigmalione carved the statue of a beautiful women - Galatea nowadays - and fell in love with her, even if she was just a piece of marble. Pigmalione did not want to surrender, so he slept close to his lover every night and pried Afrodite to give intelligence to his creation, in order to have the possibility to live his life with her. Afrodite, touched by the willingness of the artist, agreed to fulfill his wish: Galatea become a living and thinking being, the first kind of "artificial intelligence" ever conceived, in a certain naive way, from the human thought.

This story testify that since the ancient times, men have always been fascinated

by the idea of creating intelligence or intelligent beings themselves, although only in the last century this aspirations began to become reality. In particular, the first modern and real concept of artificial intelligence is dated 1957, when the American psychologist Frank Rosenblatt presented a report entitled "The Perceptron: a perceiving and recognizing automaton" [1] to the Cornell Aeronautical Laboratory commission in New York. The Perceptron was a an ancestry of modern softwares capable of recognising from different marked cards. It was necessary to tune hundreds of potentiometers to adjust parameters and find the optimal combination to fulfill the task. [2] explicates brilliantly and concisely how perceptron used to work

NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

Figure 1.1: The New York Times article about perceptron, 1956

Perceptron was anything but a linear approximator, the ancestor of what nowadays is called *neuron* and is the basic element of *Neural Networks* (NN), that will be presented in depth in Sec. 3.1.

The creation of perceptron gave birth to the first golden era for AI, during which the research on this field had a consistent growth that lasted until the 70's. At this point, expectation were too high to be satisfied by the weak computers of that age and, trivially, the enormous efforts put in training machines were not justified by the poor performances obtained. Funding decreased and AI research started to be heavily criticised in computer science field; this period is called the *first winter of*

AI.

After winter there is spring and so it happened: in the following twenty years the research about AI lived flashes of high trust against periods of poor funding and expectations, but eventually confirmed itself - from the 90's onward - as one of the most promising technology development in computer science and engineering.

Nowadays there are innumerable applications suitable for intelligent agents, especially for what concerns *Supervised Learning*. Supervised Learning is the field of *Machine Learning* - the name coined for the most promising branch of AI - in which the intelligent agent learns by observing a huge dataset, previously labeled by a supervisor that in this way addresses the learning process. Supervised Learning is used in image recognition, object detection, translation of texts that result much more readable than the ones obtained by not-intelligent translators. Clearly a push to the development of this branch of Machine Learning came from the possibility for modern computers to use and work with *big datas*, that enable training processes never achievable back in time.

1.2 What about Reinforcement Learning?

The purpose of AI has always been the one to emulate human brain, in order to obtain an agent that could not only recognize images or patterns, but that could also decide autonomously which action to perform.

«The cognitive process resulting in the selection of a belief or a course of action among several possible alternative options» ([3]) is known in psychology as *decision-making* and it is exactly the goal of one of the most modern Machine Learning techniques: Reinforcement Learning. The idea behind these algorithms is to emulate humans not only in the outcomes - the decision-making - but in the learning process too.

But how do humans learn? It can be useful at this point proceed with some examples: going back to the earliest day of childhood, every person has experienced at least once the reproaches of an adult after a mischief, the words of praise when performing a worthy action or, more pragmatically, the pain after a reckless ride ended up in an unfortunate fall. This is how humans learn: they make mistakes, they receive a prize or a punishment, they feel joy or pain and, after a lot of trials, they understand their mistakes and try to act differently to avoid them. At least, that's the theory.

In a very similar way, in Reinforcement Learning there is an agent that moves in an environment, it gains a reward or a penalty for each action performed and, after multiple trials, it learns the optimal way to act.

But, as in reality, the training phase could be very dangerous: learning to ride a bike will cost a couple of falls, probably. Therefore, some researchers tried to

investigate more in depth the possibility to train agents in a safe way, without exposing them to the dangers of the environment or, in a better way, to accomplish a *safe exploration* of the environment in order to obtain a safe learning phase. This branch of Machine Learning will be the main topic of this thesis: **Safe Reinforcement Learning**.

Chapter 2

Reinforcement Learning Problem

As previously presented in Sec. 1.2, Reinforcement Learning methods are based on the concept of an *agent* that moves in an *environment* and gains a *reward* for each action performed following the current *policy*. In the followings it will be explicated in details the basic elements introduced.

2.1 Basics

2.1.1 Environment

It is the world that surrounds the agent and can manifest itself in different states, the collection of which constitutes the *state space* and can be discrete or continuous. The environment interacts with the agent at each iteration, modifying itself based on the action it performed and providing it with an *immediate reward*. Different problems are described by different environments, with respect to the possible configurations in which the agent can find itself and the different kind of interactions that describe the dynamics involved.

2.1.2 Agent

It is the decision-maker, which observes the current state of the environment and tries to solve a particular problem taking the best action possible in the set of the available actions, the *action space*, that can be discrete or continuous. The agent interacts with the environment observing its reactions to the action taken, namely the next state and the immediate reward obtained. The agent follows the current *policy* and tries to optimize it upon the experience gained on the road. Different

agents characterize different approaches to solve the stated problem, with respect to the features of the dynamics involved.

2.1.3 Reward

It can be distinguished in *immediate reward* and *cumulative (discounted) reward*, indicated in the followings as *return*.

2.1.3.1 Immediate Reward

It consists in a numerical value obtained by a problem-specific function, which represents the effects of the action taken at that timestep. It can be positive, negative or equal to zero and usually has large values in correspondence with termination states of the environment – it tells if the agent has reached the goal or a critical end.

2.1.3.2 Return

It is the sum of all the rewards obtained during a *trajectory* (sequence of states). Since there can be infinite trajectories, it is useful to define the return as the sum of the *discounted* rewards, to avoid infinite returns and introduce a factor that permits to weight differently the rewards obtained early and late on the trajectory.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

In Eq. 2.1 G_t is the total return obtained from timestep t, r_{t+1} is the immediate reward obtained by the agent when it takes an action at timestep t that leads to the next state at timestep $t+1$ and γ is a *discount factor* between 0 and 1. A γ close to the unity is used in most of the RL problems.

2.1.4 Value Function

The *value function* represents the goodness of the state in which the agent founds itself. It is a numerical value associated with every state of the state space, it can be positive or negative and it provides an estimate of the expected return that the agent could gain from that timestep to the termination of the trajectory. At this point, it is crucial to understand the difference between *return* and *value function*. The *return* is the certain sum of (discounted) rewards obtained by the agent at the end of the trajectory, the *value function* is an estimation – more or less accurate, as we will see later – of what the agent expects to gain from that moment on. The *return* is calculated at the end of a trajectory, the *value function* is present at each timestep and for every state of the state space.

2.1.5 Policy

A *policy* is the mapping from a particular state to the probability of each action to be performed by the agent. The RL problem consists in finding the policy that leads the agent to achieve the best results (the highest return). The policy can be represented as a table in which to each state-action combination corresponds a probability, in case of discrete state and action spaces. When these spaces are extremely large – or approximately continuous – it is convenient to represent the policy as a *distribution probability function* based on a set of parameters. In this case, the objective of the RL problem is to find the parameters that describe the best policy, or to optimize an objective function (usually some version of the return) w.r.t. these parameters.

2.2 Markov Decision Processes

The Reinforcement Learning problem can be formalised using a concept that puts together the basic elements presented in the previous section: the *Markov Decision Process* (MDP). An MDP is a decision making model for discrete-time stochastic control processes, it can be uniquely defined with a tuple: *<S, A, P, R>*

- S is the *state-space*, a collection of states of the environment
- A is the *action-space*, a collection of actions that the agent can take
- P is the *success probability* of the decision-making process of the agent, in other word the probability that the action performed is, indeed, the one chosen by the agent (*transition matrix/function*)
- R is the *immediate reward* obtained by the agent after having performed an action

An MDP satisfies Markov property.

Definition 2.2.1 (Markov Property)**.** The Markov property states that given the present, the future is conditionally independent of the past. That is, the future state in which the process will be is dependent only from the current state.

In MDPs, as in reinforcement learning, the objective is to find the optimal policy that permits the agent to gain the maximum cumulative reward possible.

2.2.1 Bellman Equation

To obtain this optimal policy it comes to help the *Bellman equation*:

$$\begin{aligned} v_\pi(s) &= \mathbf{E}_\pi[G_t|S_t = s] && \text{(2.2a)}\\ &= \mathbf{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|S_t = s] && \text{(2.2b)}\\ &= \mathbf{E}_\pi[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + ...)|S_t = s] && \text{(2.2c)}\\ &= \mathbf{E}_\pi[r_{t+1} + \gamma G_{t+1}|S_t = s] && \text{(2.2d)}\\ &= \mathbf{E}_\pi[r_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] && \text{(2.2e)} \end{aligned}$$

$$v_\pi(s) = \sum_{a\in A} \pi(a|s)(r_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s')) \tag{2.3}$$

Where $P_{ss'}^a$ is the probability of passing from state s to state s' taking the action a, $\pi(a|s)$ is the probability to take action a given the state s, r_s^a is the reward gained taking action a in state s. If MDP was a deterministic process, $P_{ss'}^a$ would be equal to 1 only for a particular s' (we are sure that if we are in state s and take action a, there will be no accidents and we will arrive at state s' with probability of 1), so the summatory will disappear and only $v(s')$ for that particular s' will remain. The term $P_{ss'}^a$ appears to deal with stochasticity of the process. The main goal is to find a parametrization for π that provides the highest probability for the action that will lead to maximum return.

$$\pi_*(s) = \begin{cases} 1 & \text{if } a = arg\,max_{a\in A}(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s')) \\ 0 & \text{if otherwise} \end{cases} \tag{2.4}$$

The following is the *Bellman optimality equation* to be solved to find the optimal policy.

$$v_*(s) = \max_a R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s') \tag{2.5}$$

Since there is a non-linear term – the maximization – it is not possible to solve this equation directly, but iterative methods are needed. In the followings only tabular MDPs will be considered, i.e. MDPs where state and action spaces are finite and can be described in a tabular value. Later on it will be presented how to deal with infinite and dense spaces.

2.3 Dynamic Programming

2.3.1 Value Iteration

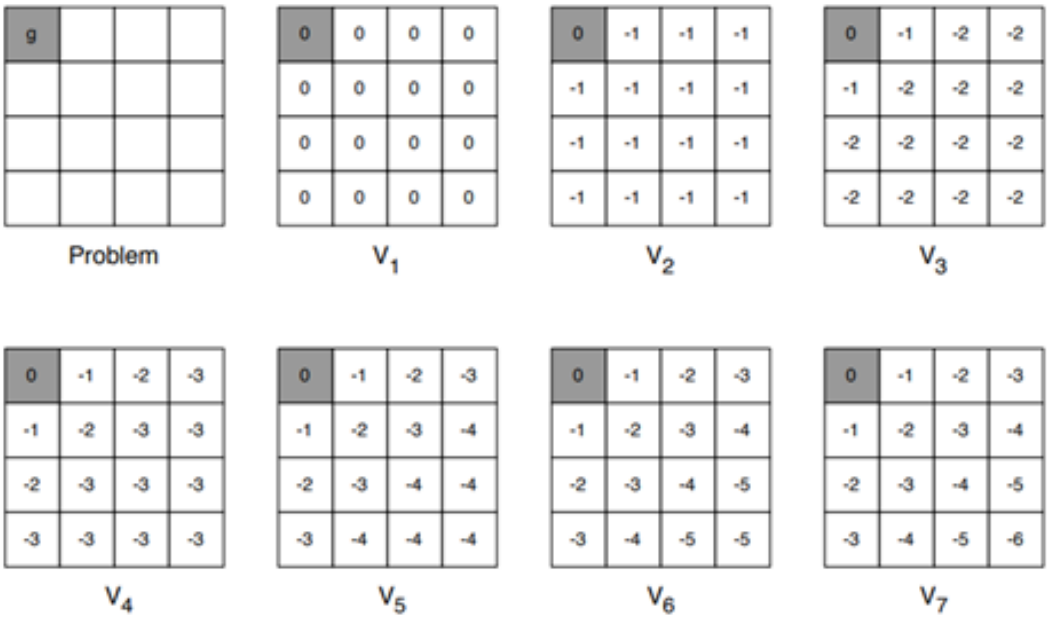
The Bellman equation is the core of the value iteration algorithm for solving an MDP. The objective is to find the value function for each state and it is done following the following steps:

1. start with arbitrary initial values for the value function (usually zeros)
2. calculate the utility of a state using the Bellman equation and assign it to the state
3. repeat step 2 until convergence (proved to converge)

Policy acts greedily with respect to value function, trying to always go in direction of a state with higher value.
Here an example is presented to help understanding the method: the goal is to find the optimal policy that guides the agent to the target (top-left corner) in fewer moves possible. The environment is basic with this features:

- *Determinist policy* $\rightarrow P^a_{ss'} = 1$ for the given action (if acting up , the agent goes up with probability 1)
- *Undiscounted reward* $\rightarrow \gamma = 1$
- *Reward* $\rightarrow -1$ always, 0 when reaching the target
- value function initialized to all zeros

Figure 2.1: Value Iteration Process [4]



$$v_*(s) = \max_a R^a_s + \gamma \sum_{s' \in S} P^a_{ss'} v_\pi(s')$$

$v_2(1) = \max(-1 + 1 * 1 * 0 = -1[down], -1 + 1 * 1 * 0 = -1[right], -1 + 1 * 1 * 0 = -1[left]) = -1$
$v_3(1) = \max(-1+1*1*-1 = -2[down], -1+1*-1*-1 = -2[right], -1+1*1*0 = -1[left]) = -1$
$v_2(2) = \max(-1 + 1 * 1 * 0 = -2[down], -1 + 1 * -1 * 0 = -2[right], -1 + 1 * 1 * 0 = -1[left]) = -1$
$v_3(2) = \max(-1+1*1*-1 = -2[down], -1+1*-1*-1 = -2[right], -1+1*1*-1 = -2[left]) = -2$

Starting from state 9 and following the final value function, the optimal policy leads the agent to the target within 3 moves *[up, up, left]*, the shortest path possible. It is important to understand that eventually the algorithm provides a mapping from states to value function, not an explicit mapping to actions; actions will be extrapolated from the value function itself.

2.3.2 Policy Iteration

The value iteration algorithm represents a way to estimate the goodness of each state, the policy is not expressed directly, but mainly follows the value function. There is another way to obtain the same result, providing an explicit operator for the policy. The objective is to find for each state explicitly the action that gets the highest return and it is done following the following steps:

1. start with an arbitrary initial policy (usually random distribution over actions, for each state)
2. evaluate the policy calculating the expected return (aka value function) of each state following the current policy

$$\mathbf{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|S_t = s]$$

3. improve the policy acting greedily with respect of the returns obtained at step 2, i.e. subscribe the old policy with a new one
4. repeat from step 2 until convergence

To explain Policy Iteration method, it will be applied to the example previously presented. Initial policy set is a uniform distribution.

$$\pi(up|\cdot) = \pi(down|\cdot) = \pi(left|\cdot) = \pi(right|\cdot) = 0.25$$

Figure 2.2: Initial Random Policy [4]

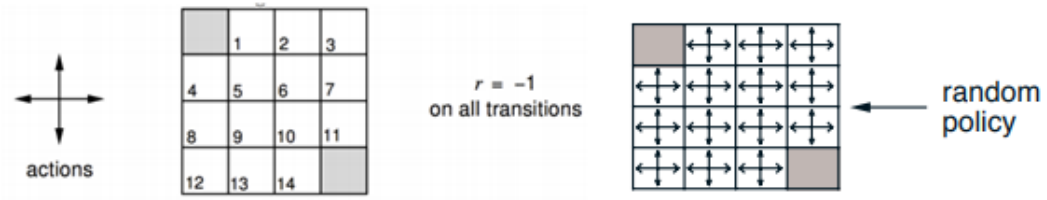
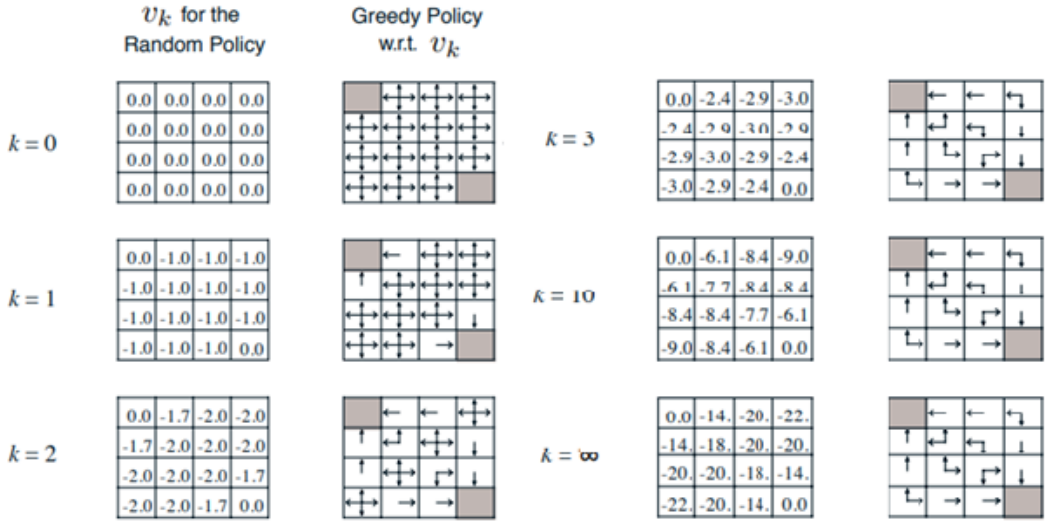


Figure 2.3: Policy Iteration Process [4]



Starting from state 9 and following the final value function, the optimal policy leads the agent to the target within 3 moves *[up, up, left]*, the shortest path possible. In contrast with what happened with value iteration, in this case eventually the algorithm provides a direct mapping from states to actions: the policy is explicit. Furthermore, policy iteration does not involve non-linear functions: maximum operator disappears since the policy is fixed and the value function refers to that particular policy; this leads to the possibility to solve analytically the problem, but it will not be object of this study and it will be omitted.

2.4 Model-free Prediction and Control

Until now MDPs have always been *fully know*, meaning that every element of the tuple <*S, A, P, R*> was in agent's possession. In Reinforcement Learning typical problems this is not true, but often there is no knowledge of the *model* of the environment: how it reacts to the action taken by the agent. This model corresponds to the P of the former tuple, namely the *transition function*, and the lack of this element does not allow the use of the Dynamic Programming algorithms presented in Sec. 2.3. This is the starting point of proper Reinforcement Learning, where an agent has to learn how to act without any prior knowledge of the environment. Two main families of algorithms arise from this problem: *model-free* and *model-based* Reinforcement Learning. In this work only the former will be used and analysed, for this reason the latter will not be mentioned anymore. Beside this division of Reinforcement Learning algorithms, it is important to explicate the distinction between prediction and control phases. Prediction phase is the first part of the Reinforcement Learning process, comparable with the evaluation step of policy iteration. Control phase is the policy optimization part, where the policy is improved.

2.4.1 Prediction

In this phase the agent starts acting within the environment, gains knowledge of it and approximates a value function of the *partially unknown MDP*. There are different methods to accomplish this result, in the following the most important ones will be introduced.

2.4.1.1 Monte-Carlo Learning

Monte-Carlo prediction consists in learning the value function from episodic experience and then averaging the results obtained above a large number of episodes. The process is the following:

1. the agent starts following the current policy in the environment until it reaches an ending state or a maximum number of steps is performed

2. the return is calculated as the weighted sum of the rewards obtained along the episode and it is assigned to that state

3. repeat step 1-2 for a large N number of episodes

4. average upon the number of episodes N the sum of returns stored in every states

The average return correspondent to every state is the approximated value function of that state; it is proved that, for infinite episodes, the function converges to the correct value function with respect to the current policy π (consequence of the *law of large numbers*).

2.4.1.2 Temporal-Difference Learning

Similar to Monte-Carlo, Temporal-Difference consists in learning the value function from episodic experience, but in contrast to the former it does not use the *empirical mean* over complete episodes as estimator, but it learns from incomplete episodes by bootstrapping. *Bootstrap* is a statistical method that uses an estimate to calculate other estimates by resampling, in other words it "learn a guess from a guess" :

1. initialize to zero the value function
2. the agent performs an action in the environment, gaining an immediate reward R
3. use Eq. 2.2e to approximate the value function of the current state using the current approximation of the value function at the next state

$$v_\pi(s) = E_\pi[r_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

4. update $v(s)$ with the difference between the old approximated value and the new approximated value

$$v\pi(S_t) \leftarrow v(S_t) + \alpha[r_{t+1} + \gamma v_\pi(S_{t+1}) - v(S_t)] \tag{2.6}$$

5. repeat steps 2-4 until a terminal state is reached
6. repeat steps 2-5 for a N number of episodes

The main differences between MC and DP prediction are the followings:

- TD can learn online after every step, while MC must wait until end of episode before return is known
- TD can learn from incomplete sequences, while MC can only learn from complete sequences
- TD works in continuous (non-terminating) environments, while MC only works for episodic (terminating) environments
- TD has low variance because the estimate is made upon a target that is close in the future and depends upon one or few actions, but it is biased since the target is not the true value of what we are trying to estimate, but an estimation itself (*bootstrap*)

- MC has high variance because the estimate is made upon a target that is far in the future and depends upon many action, but it has zero bias because the target is the true value of what we are trying to estimate ($V_\pi(s)$ is not $G_t(s)$)

2.4.2 Control

This is the phase in which the optimization process takes place, aiming at outputting an improved policy. Basically it is done applying the GPI mechanism:

- evaluation of the policy done with MC or TD algorithms
- optimization using some technics to assure a good balance between exploring the environment and exploiting the current policy

As already seen in Sec. 2.4.1.1, for its nature MC control can be done only at the end of each episode, in an off-line manner. Hence TD learning has several advantages over MC:

- lower variance
- on-line learning, i.e. learning performed at every timestep
- use of incomplete sequences

For this reason, only TD based control algorithms will be presented in the followings.

2.4.2.1 SARSA

SARSA algorithm owes its name to the tuple $< S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1} >$ that defines the elements used for the learning process:

- $S_t \rightarrow$ state of the agent at timestep t
- $A_t \rightarrow$ action taken following the current policy π at timestep t
- $R_{t+1} \rightarrow$ reward gained by the agent at timestep $t+1$, after having performed the action A
- $S_{t+1} \rightarrow$ state of the agent at timestep $t+1$, after having performed the action A
- $A_{t+1} \rightarrow$ action taken following the current policy π at timestep $t+1$

It is evident that there is no trace of the *transition model* in the optimization process elements, so *SARSA* is a model-free algorithm. Another characteristic of this learning process is that it does not rely on a *state-value function*, but on the novel concept of *action-value function*. «The action-value function $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π.»[4]

$$q_\pi(s, a) = \mathbf{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.7}$$

As it can be seen there are some similarities between Eq. 2.2a and Eq. 2.7, the difference between the two is that *state-value function* $v_\pi(s)$ depends only on the current state and provides knowledge about the goodness of that state, while *action-value function* $q_\pi(s, a)$ depends both on the current state and the action taken by the agent following the current policy, returning a notion of the goodness of that action taken in that state. To explain deeper the nature of the *action-value function*, it could be possible that the same action, e.g. going *up*, in a particular state is the optimal one, in another state returns very low performances. Similarly to Eq. 2.2e, *action-value function* can be re-written as:

$$q_\pi(s, a) = \mathbf{E}_\pi[r_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{2.8}$$

The process is similar to the one presented in Sec. 2.3.1, but instead of using Eq. 2.6 to calculate the state-value function, the following one is used:

$$Q_\pi(S_t, A_t) \leftarrow Q_(S_t, A_t) + \alpha[r_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) - Q_\pi(S_t, A_t)] \tag{2.9}$$

It is now clear why in addition to S_t, A_t, R_{t+1} also S_{t+1}, A_{t+1} are required. Summarizing all the steps of the algorithm, it looks like:

- move one step selecting a_t from $\pi(s_t)$
- observe $R_{t+1}, S_{t+1}, A_{t+1}$
- update the state-action function $Q_\pi(s_t, a_t)$ using Eq. 2.9
- update the policy $\pi(s_t) \leftarrow \underset{a}{\text{argmax}}\ Q(s_t, a_t)$

Exploiting vs Exploring *SARSA* is assured to converge to the optimal policy for the *Russel and Norvig Greedy in the Limit of Infinite Exploration (GLIE)* [5] theorem, meaning that every state has to be visited an unbounded number of times in order to be sure that every possible trajectory is explored and the optimal one is found. This is the main challenge of reinforcement learning technique: balancing the exploitation of a known policy that is proved to be good and the exploration of the environment, in order to be sure that the current policy is indeed the optimal one.

This requirement cannot be fulfilled following always greedily the current policy, since this is mere exploitation and it misses some exploration of the environment. In fact, it could happen that the optimal policy is hidden, so only an high level of exploration of the environment could find it. To overcome this limit different exploration strategies are known in literature, the simplest one is the ϵ-greedy policy: the agent follows the current policy with a probability of $1 - \epsilon$, while with probability ϵ it takes a random action.

The structure of the algorithm is the following:

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 2.4: SARSA structure [4]

2.4.2.2 Q-Learning

Q-learning algorithm is similar to *SARSA*, with slight differences that have heavy impact on the whole optimization process. The structure is the following:

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Figure 2.5: Q-Learning structure [4]

The main difference to be seen is in the update rule:

$$Q_\pi(S_t, A_t) \leftarrow Q_(S_t, A_t) + \alpha[r_{t+1} + \gamma \max_{a \in A} Q_\pi(S_{t+1}, A_{t+1}) - Q_\pi(S_t, A_t)] \quad (2.10)$$

In Eq. 2.9 the target of the optimization always follows the current policy π (net of ϵ probability to chose a random action), in Eq. 2.10 the target is the maximum value function available, without caring if it corresponds to the one indicated by the current policy. This permits to untie the learning process from the current policy: in this way it is possible to optimize a policy while exploiting a different one. This process is called *off-line learning* and is very useful since it «dramatically simplifies the analysis of the algorithm and enables early convergence proofs»[6] Q-learning is at the base of the most important algorithms in **Deep Reinforcement Learning**, that will be discussed in the next Chapter.

Chapter 3

Deep Reinforcement Learning

Until now, both value function and policy function have been represented by a *lookup table* in which every state maps a value $V(s)$ or an action $\pi(s)$, or a state-action pair maps an action-value $Q(s)$. This representation leads to several issues when scaling up to large or continuous problems:

- heavy memory consumption when dealing with huge number of action or states
- slowdown of learning process

Another way to represent the main function involved in Reinforcement Learning is to find some good approximators, in order to greatly reduce the amount of data to store and use for calculations. In this way only the parameters of the approximator would be saved and a full family of optimization processes could be introduced in the learning process. Deep Reinforcement Learning is the evolution of classic Reinforcement Learning, that happens when the main functions involved are obtained using Deep Learning techniques. In particular, the approximators chosen are *Neural Networks*.

3.1 Neural Networks

Neural Networks (NN) are among the most powerful known function approximators and are the foundation of modern Machine Learning. Neural Network have a structure similar to the one of a brain and their behaviour mimics the learning process of intelligent beings. In particular, NN basic element is the *neuron*, which is a value that, connected with other values of other neurons, shapes a network. The network of neurons is constituted by at least three layers:

- *input layer* → it receives the input values
- *output layer* → it provides the processed values
- *hidden layer(s)* → each neuron of this kind of layers is a linear combination of some (or every) neurons of the previous layer, then it is *activated* by a non-linear function and passed to the next layer

$$output = f_{activation}\left(\sum_{\#neurons} input_i + bias\right) \tag{3.1}$$

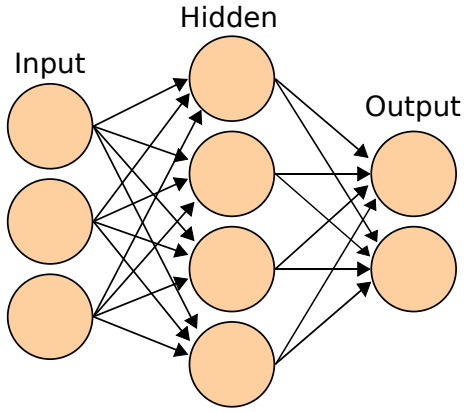


Figure 3.1: Neural Network basic structure

Therefore, Neural Networks are composed of different layers of linear combinations, that alone could only approximate linear relations; this is the reason that leads to the use of non-linear *activation functions* as already stated. Theoretically whatsoever non-linear function could be an activation function, but in concrete only particular ones are mostly used:

ReLU	$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} i = 1, ..., J$
tanh	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$

Table 3.1: Most used activation functions

Neural Networks are used to approximate unknown functions, given an input and the correspondent output. The learning phase consists in a minimization of the loss between the predicted and the real output (Eq. 3.2 is MSE Loss) performed by backpropagation (Fig. 3.2), with respect to weight and bias of each layer. The purpose is to obtain the parameters describing the function that better predicts the output, given the input.

$$\frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N} \tag{3.2}$$

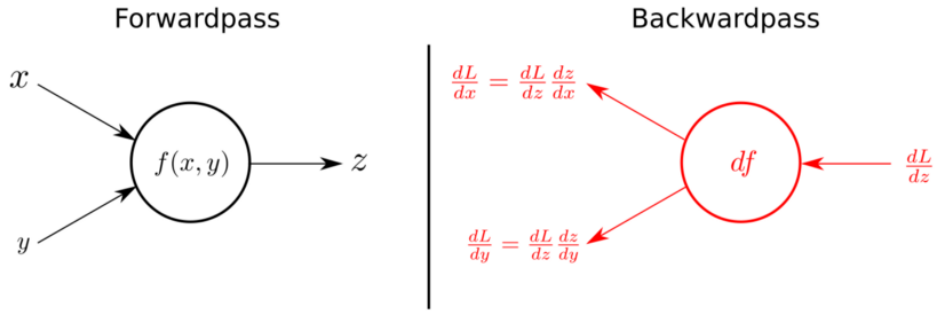


Figure 3.2: «The forward pass on the left calculates z as a function f(x,y) using the input variables x and y. The right side of the figures shows the backward pass. Receiving dL/dz, the gradient of the loss function with respect to z from above, the gradients of x and y on the loss function can be calculate by applying the chain rule.» [7] Figure from [8].

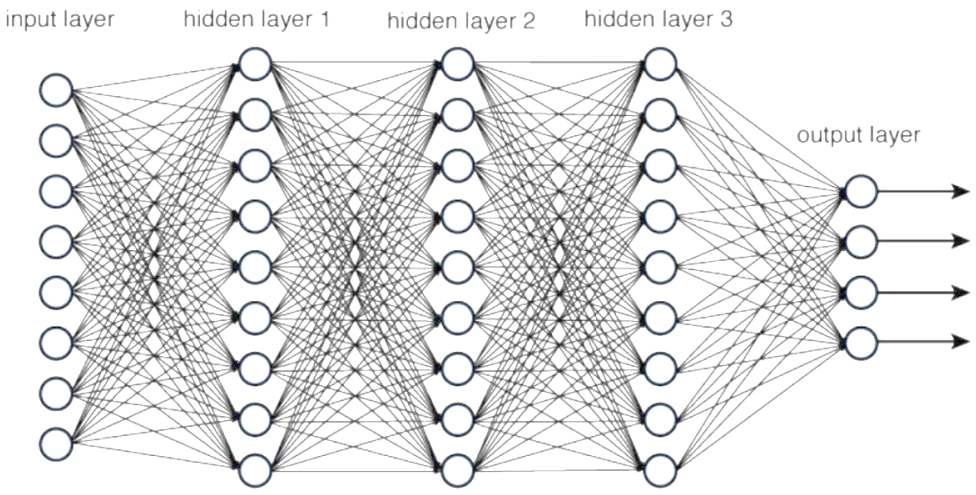


Figure 3.3: Deep Neural Network

Considering the case where input and associated outputs are perfectly know and labeled and the purpose is to find the function connecting them, this branch of Machine Learning is called *supervised learning*. When Neural Networks are provided with multiple hidden layers, they take the name of *Deep Neural Networks* (DNN). When DNN are used to approximate value function, action-value function or directly policy function, Reinforcement Learning takes the name of Deep Reinforcement Learning.

3.2 Value-based Algorithms

Value-based algorithms rely on Value Iteration process presented in Sec. 2.3.1 and on Q-Learning presented in Sec. 2.4.2.2. In particular, the basic evolution of Q-Learning in Deep Learning field is the *Deep Q-Network*

3.2.1 Deep Q-Network

Deep Q-Network uses a Neural Network to approximate the action-value function that guides the agent through the environment. Without deepening too much in the illustration of this algorithm, it is useful to present some elements that will be largely used later. The structure of the algorithm is reported in Alg. 1.

In DQN the concept of *replay buffer* is introduced, in which $< S_t, A_t, R_{t+1}, S_{t+1} >$ tuples are stored in order to perform the learning process on both new and old experience. This replay buffer aims to solve instability problems that may occur when using non-linear function approximators. Learning phase is performed after taking an action, observing the results and storing the obtained tuple in the replay buffer; action-value function network weights are updated in order to minimize the indicated loss. Another interesting feature of DQN is how the target of the loss and the loss are calculated:

$$y_i = r_{i_t} + \gamma \max_{a \in A} \hat{Q}(s_{i_t}, a_{i_t}) \tag{3.3}$$

$$L = \frac{\sum_{i=1}^{N} \left(Q(s_{i_{t+1}}, a_{i_{t+1}}) - y_i \right)^2}{N} \tag{3.4}$$

In Eq. 3.3 $\hat{Q}$ is the *target network* used only to estimate the target of the loss function, which weights are substituted by the ones of the main network only every C steps (Alg. 1). In Eq. 3.4 the difference is between the value estimated from the target network $\hat{Q}$ and the one provided by the current Q network, that is the one followed as policy. In this way, target and current action-value are not correlated, improving stability and convergence capability of the algorithm. Minimization

Algorithm 1 Deep Q-Network

Initialize network Q, target network $\hat{Q}$ and empty replay buffer $\mathcal{R}$
$s_t \leftarrow s_0$ ▷ Reset the environment
while not converged **do**
    Set ϵ ▷ ϵ-decay
    Select action a_t following ϵ-greedily $\max(Q(s_t))$
    Execute action a_t and observe next state s_{t+1}, reward r_t and d_t done signal
    Store transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{R}$
    if enough experiences in $\mathcal{R}$ **then**
        Sample a random minibatch from $\mathcal{R}$
        for every element in minibatch **do**
            if done **then**
                $y_i = r_i$
            else
                $y_i = r_{i_t} + \gamma \max_{a_{i_{t+1}} \in \mathcal{A}} \hat{Q}(s_{i_{t+1}}, a_{i_{t+1}})$
            end if
        end for
        Calculate the loss $\mathcal{L} = \frac{\sum_{i=1}^{N} \left(Q(s_{i_t}, a_{i_t}) - y_i\right)^2}{N}$
        Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every C steps, copy weights from Q to $\hat{Q}$
    end if
end while

of the loss with respect to network weights is performed by *Stochastic Gradient Descent*.

3.3 Policy-based Algorithms

Aside of value-based, there is another family of algorithms: policy-based algorithms aim at finding the optimal policy function directly and without using any value-function. This is a genuine result of the use of function approximators (being them NN or other non-linear functions), that enable the learning of a function that has a state as input and outputs directly an action or the probability distribution over the available actions (the best action has the largest probability). Therefore policy-based algorithms are optimization processes, in which the objective is to maximize an objective function with respect to some parameters (that shape the policy). The fundamental algorithm of this family is *REINFORCE*.

3.3.1 REINFORCE

REINFORCE - or Monte-Carlo Policy Gradient - updates its parameters by *stochastic gradient ascent*, in order to maximize the *return* of each episode. The optimization step is performed at the end of every episode, leveraging on the *policy gradient theorem*

$$\nabla_\theta J(\theta_t) = \mathbf{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s_t, a_t) G_t] \tag{3.5}$$

and updating the parameters with:

$$\Delta_{\theta_t} = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t \tag{3.6}$$

The structure of the algorithm is the following:

Algorithm 2 REINFORCE

Initialize θ arbitrarily
for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for t=1,T-1 **do**
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t$
  end for
end forreturn θ

Though policy-based algorithms, such REINFORCE, are characterized by high variances, they have some advantages over value-based ones:

- better convergence
- better performances for continuous action spaces
- can learn stochastic policies

3.4 Actor-Critic Algorithms

Actor-critic algorithms aim at solving the problem of high variance that characterizes policy-based learning. In order to fulfill this objective, a *critic* is introduced in the optimization process to evaluate the goodness of the current policy. As already presented in Sec. 2.3.1, an object that has this as main objective is the value function; from here the idea of inserting it in policy gradient algorithms to improve them. In actor-critic algorithms, the *actor* is the policy function and the *critic* is the action-value function, which evaluates the current policy and suggest to the actor a direction towards to address the optimization process. Differently from

REINFORCE and other policy-based algorithms, actor-critic algorithms follow an *approximate* policy gradient:

$$\nabla_\theta J(\theta) = \mathbf{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w] \tag{3.7}$$

$$\Delta_\theta = \alpha \nabla_\theta \log \pi_\theta(s,a) Q_w \tag{3.8}$$

In Eq. 3.8 and 3.7 Q_w is the approximate action-value function that is used instead of the return G_t. The actor is updated by SGD, while the critic is updated using TD error presented in Sec. 2.4.1.2.

The structure of a generic actor-critic algorithm is the following:

Algorithm 3 Q Actor Critic

Initialize parameters s, θ, w
for t=1,T-1 **do**
  Select action $a_t \sim \pi_\theta(s)$
  Observe next state s_{t+1}, reward r_t
  Select next action $a_{t+1} \sim \pi_\theta(s_{t+1})$
  $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \log \pi_\theta(s,a)$ ▷ Update policy
  $\delta_t = r_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s,a)$ ▷ TD error for action-value function
  $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$ ▷ Update Q-function
  $s_t \leftarrow s_{t+1}$
  $a_t \leftarrow a_{t+1}$
end for

From actor-critic baseline, a whole family of algorithms arise. Expaining the peculiarity of each exceeds the purposes of this work, hence in the followings only the algorithm chosen to perform experiments will be presented.

3.4.1 Deep Deterministic Policy Gradient

DDPG is an actor-critic based algorithm that also exploits some features introduced by DQN, such replay buffer and target network. DDPG structure is introduced here and explained further:

Algorithm 4 Deep Deterministic Policy Gradient

Initialize policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{R}$
$\theta_{targ} \leftarrow \theta, \phi_{targ} \leftarrow \phi$ ▷ Target parameters equal to main ones
while *episode* $< N$ **do**
  $s \leftarrow s_0$ ▷ Reset environment state
  Initialize random process $\mathcal{N}$ ▷ For action exploration
  $t \leftarrow 0$ ▷ Timestep set to 0
  while $t < T_{max}$ **and** not done **do**
    Select action $a_t = clip(\mu_\theta(s) + \mathcal{N}_t, a_{low}, a_{high})$
    Execute action a_t and observe s_{t+1}, r_t and d done signal
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
    $s_t \leftarrow s_{t+1}$
    if time to update **then**
      for m=1,M **do**
        Sample a random minibatch from $\mathcal{R}$
        $y_i = r_{i_t} + \gamma(1-d)Q_{\phi_{targ}}(s_{i_{t+1}}, \mu_{\theta_{targ}}(s_{i_{t+1}}))$
        ▷ Target network used for target estimation
        Update Q_ϕ using ▷ Stochastic Gradient Descent
        $\nabla_\phi \frac{1}{|B|} \sum_{(s_{i_t}, a_{i_t}, r_{i_t}, s_{i_{t+1}}) \in B} (Q_\phi(s,a) - y_i)^2$
        Update μ_θ using ▷ Stochastic Gradient Descent
        $\nabla_\theta \frac{1}{|B|} \sum_{(s) \in B} Q_\phi(s, \mu_\theta(s))$
        $\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi$
        $\theta_{targ} \leftarrow \rho\theta_{targ} + (1-\rho)\theta$ ▷ Update target networks
      end for
    end if
    $t \leftarrow t + 1$
  end while
  episode $\leftarrow$ *episode* $+ 1$
end while

In Alg. 4 it can be seen that a replay buffer is also present, as it is in DQN, and four NN are initialized:

- actor network
- critic network
- actor target network
- critic target network

The last two networks are used in the optimization phase of the process, in particular:

- actor target network μ_{target} provides the action a_{t+1} as input to the critic target network
- critic target network Q_{target} is used to obtain the target of the loss to be minimized

The form of the gradient is due at off-policy updates with batches of experience, hence it is necessary to take the mean of the sum of gradients calculated from the batch. Target networks parameters are softly-updated with optimal networks parameters at each step. The noise present when selecting the action to perform is required in order to obtain a good exploration, in particular for DDPG the *Ornstein-Uhlenbeck Process* [9] is frequently used.

Chapter 4

Safe Reinforcement Learning

In Sec. 1.2 some hints about Safe Reinforcement Learning have been introduced using a naive example about how to learn to ride a bike. Translating it to Machine Learning applications, it could be the case of a robot learning on a work bench that should stay away from the borders in order to avoid falling down. Another example could be the one of a robotic arm that has to work in strict contact with some humans without risking to hurt them.

In general, *safety* could be inherent with multiple aspects and multiple phases of the development of an optimal policy:

- *safe exploration* can be referred to the case of performing a safe learning, meaning to respect some peculiar conditions during the learning process
- *safe policy learning* can be referred to the case of eventually obtaining a safe policy, namely a policy that respects some peculiar conditions as outcome of the learning process (without the latter being safe)

This Thesis is inherent with the former option, hence the experiments will seek the purpose of obtaining a policy that satisfies stated conditions either during or at the end of the learning process.

In this chapter, a detailed explanation of differences and commonalities between Safe and classical Reinforcement Learning will be presented, State-of-the-art will be treated and the chosen algorithm will be explored in depth.

4.1 Basics

Safe Reinforcement Learning inherits the basic structure from classical Reinforcement Learning, in particular the following elements are the same:

- *state space*
- *action space*
- *reward function*

From these basics, different branches of Safe Reinforcement Learning algorithms may be pursued.

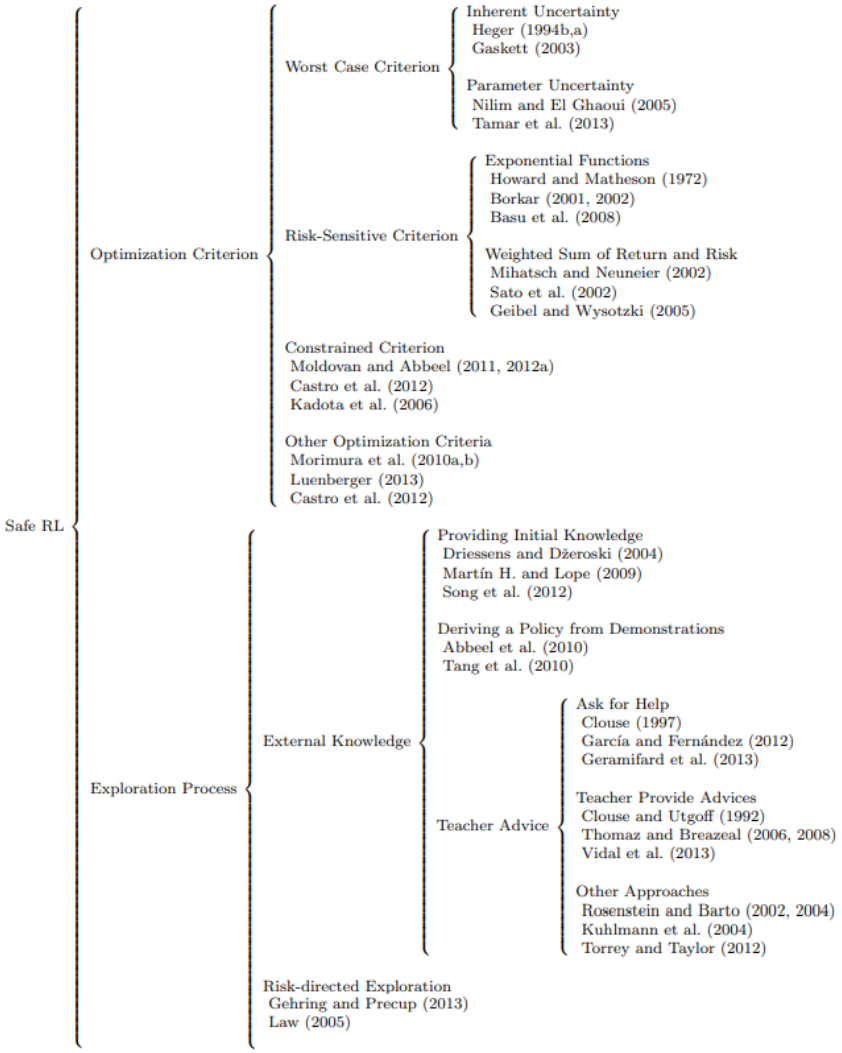


Figure 4.1: Comprensive Survey of Safe Reinforcement Learning algorithms [10]

It is not within the purposes of this work to explore in detail all of these branches, but an exhaustive presentation can be found in [10]. In Fig. 4.1 there is a graph

extrapolated from the previous paper, in an effort to make the reader familiar with the topic. Without going into too much detail, it can be seen that SRL algorithms can be divided into two major categories.

Optimization Criteria modification regards processes in which the agent explores the environment in the same way it would do in classical Reinforcement Learning. The improvement in terms of safety comes from a change of the optimization process. In this kind of algorithms, the optimization is subject to some constraints, such as:

- *worst case constraint*, where the *return* obtained in the worst case by the policy learned has to be above a threshold
- *risk-sensitive criterion*: where there is some parameter that can sense the *risk* and the optimization is performed in order to optimize a combination of *risk* and *return*
- *constrained criterion*, that will be analysed deeper in Sec. 4.2.1

Exploration Process modification concerns processes in which the learning approach is the same as in classical Reinforcement Learning, but the agent explores the environment in a safer way. These algorithms are the most indicated for environments where an unconstrained exploration could lead to severe consequences. In particular, there are different ways to modify the exploration process:

- *external knowledge* of the environment provided at the beginning or during the learning process, in order to give information to the agent that prevent it from exploring risky states
- *risk-directed exploration*, that will be analysed deeper in Sec. 4.2.2

4.2 State of the Art

It is worth focusing the attention on some particular branches of Safe Reinforcement Learning, the ones that concern more than others the algorithm selected for this work. In particular, the selected algorithm fits in the fields of **Constrained Criterion** and **Risk-directed exploration**.

Although in the Fig. 4.1 they are located in different subsections, they still share some peculiarities. Primarily, in both of them it is required a specific element to measure the dangerousness of the state in which the agent finds itself or could come across in close future. This element is called *cost function* or *risk function* and the goal of these algorithms is to keep it below a certain threshold.

4.2.1 Constrained Criterion

If classical Reinforcement Learning can be summarized as a problem of unconstrained optimization, Constrained Criterion SRL algorithms are constrained optimization problems: to optimize the return while keeping cumulative cost below some thresholds. Another kind of Markov Decision Process is required to describe exactly the newfound problem, which includes constraints: **Constrained Markov Decision Process** (CMDP). The general structure of a Constrained Criterion SRL algorithm is the following [10]:

Definition 4.2.1 (Constrained Criterion)**.** In the constrained criterion, the expectation of the return is maximized subject to one or more constraints, $c_i \in C$. The general form of this criterion is shown in the following

$$\max_{\pi \in \Pi} \mathbf{E}_\pi(R) \quad subject \quad to \quad c_i \in C, c_i = \{h_i \leq \alpha_i\} \tag{4.1}$$

where c_i represents the i-th constrain in C that the policy π must fulgill, with $c_i = \{h_i \leq \alpha_i\}$ where h_i is a function related with the return and α_i i the threshold restricting the values of this function. Depending of the problem, the symbol $\leq$ in the constraints $c_i \in C$ may be replaced by $\geq$.

4.2.1.1 Constrained Policy Optimization

One of the first algorithms developed in the frame of SRL is Constrained Policy Optimization [11], that is a Constrained Criterion process. In this algorithm the optimization of the policy leverages on some approximations that permit to treat the problem presented in Eq. 4.1 as a convex one and solve it using duality. This procedure permits to solve the equations with respect of fewer variables than the original ones (i.e. the parameters of the policy network, usually a large number).

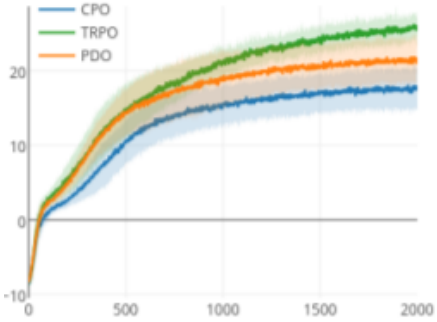


Figure 4.2: Return

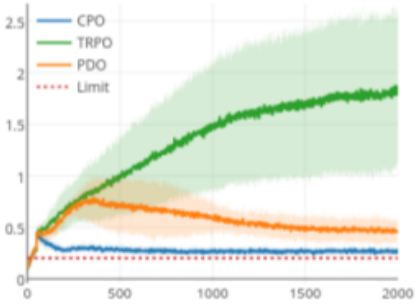


Figure 4.3: Constraint violations

Figure 4.4: CPO (blue) results against two unsafe algoithms (orange, green). Images from [11].

CPO is an effective method of learning a safe policy, but it has several problems:

- slow convergence (Fig. 4.2)
- degradation of the reward obtained (Fig. 4.2)
- violation of constraints during the early stages fo the training (Fig. 4.3)

For these reasons, new algorithms and strategies need to be explored in order to obtain a safe exploration, without impacting on agent performances.

4.2.2 Risk-directed Exploration

Passing now to Risk-directed exploration SRL, it can be seen in Fig. 4.1 that these methods lie in the category of *Exploration Process* modification, which is what makes them different from classical RL. These algorithms still require a cost function, but use it in a different way compared with Constrained Criterion processes.

Specifically, Risk-directed exploration SRL aims to leverage on the cost function in order to modify the exploration space, in the optical of avoiding risky states. In Constrained Criterion SRL it is necessary to explore unsafe states in order to learn about their dangerousness and successively act against it (in the same way it is necessary to explore good states to obtain knowledge about reward). In Risk-directed exploration algorithms, a part of hazardous exploration is avoided using the cost function as a signal.

4.3 Safety Layer

Safety Layer [12] is an algorithm that lies in between the categories presented in Sec. 4.2.1 and Sec. 4.2.2. It consists in the application of a Safety Layer to classical Reinforcement Learning methods, that corrects dangerous actions to safe ones before the agent performs them.

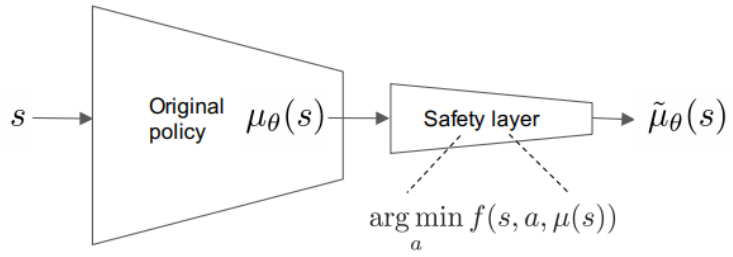


Figure 4.5: Safety Layer algorithm structure

In Fig. 4.5 it can be seen that the inputs are the state s, the action selected by the Neural Network $\mu_\theta(s)$ and the perturbed action a. Current state and action are the inputs of the Safety Layer, which eventually corrects $\mu_\theta(s)$ and provides $\tilde{\mu}_\theta(s)$, the new safe action to be performed.

Therefore, it would be correct to categorize the algorithm as belonging to Risk-Constrained exploration SRL, but focusing on how an action is marked as safe or dangerous, the Constrained Criterion (Fig. 4.1) can be recognised. Indeed, an action is marked as dangerous only if it brings the agent to a state s' associated to a *cost function* $c(s')$ that violates a threshold C.

$$c(s') \leq C \tag{4.2}$$

In Constrained criterion SRL, the agent should visit the whole state space to be able to associate a cost function to every possible state, but this is not possible - as already stated in Sec. 3 - for continue state/action spaces. Moreover, it would necessarily lead to numerous violations during training, as explained in Sec. 4.2.2.

Safety Layer algorithm solves both these problems using an approximation of the cost function instead of the real one, in order to be able to predict the value of $c(s')$ without having visited the state s' not even once. Moreover, in some specific situations it is possible to obtain a closed form solution to the problem of action correction performed by the Safety Layer, as explained in the following Sections.

4.3.1 Safety Layer approximation

In Sec. 4.3, the concept of *approximated cost function* has been introduced. It is, indeed, the most important element of the algorithm discussed in this thesis. Neural Networks are powerful approximators, as treated in Sec. 3.1, therefore the choice easily falls on them.

The initial idea would be to use a Neural Network to associate at each state s' in input an approximated cost function $\bar{c}(s')$ as output.

In Safety Layer algorithm something different happens. First of all, the cost function is modeled as a first-order approximation of $c(s')$ with respect to the action a.

$$\bar{c}(s') \triangleq c(s, a) = \bar{c}(s) + g(s; w)^\top a \tag{4.3}$$

In Eq. 4.3 s' is the state reached starting at s and taking action a, $\bar{c}(s)$ is the approximated cost function at the state s and $g(s; w)$ is something that represents how much the cost function is sensible to variations caused by actions, at state s. In particular, the Safety Layer Neural Network will approximate the $g(s; w)$ function, where it is made explicit the dependency from network parameters w.

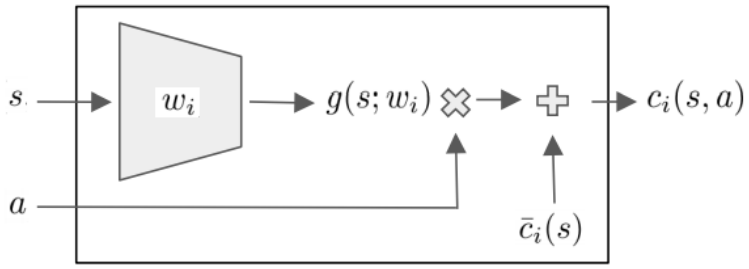


Figure 4.6: Each safety signal $c_i(s, a)$ is approximated with a linear model with respect to a, whose coefficients are features of s, extracted with a NN [12]

4.3.2 Safety Layer pre-training

As stated in Sec. 4.3, Safety Layer is applied to a classical Reinforcement Learning algorithm. The training of the neural network g is done in a phase that precedes the learning process of the classical Reinforcement Learning algorithm, for this reason it is called *pre-training*.

Ideally, the purpose of the writers of [12] was to execute the learning process upon an existing database of tuples $< s, a, s' >$ (where the cost function $c(s)$ is considered as part of the state s), in order to perform the pre-training on offline data rather than simulating new ones. While this is applicable and even advised when a database is effectively available, usually it is not the case. Therefore, to simulate the required data, the strategy is to run a *random policy* and store in a buffer the single-step tuples $< s, a, s' >$. In this way, the randomness of data present in a database is properly simulated. In fact data can be obtained and saved in very different times and manners, hence the neural network has to be able to learn from this random distribution of data.

Pre-training consists in applying Supervised Learning presented in Sec. 3.1 to the data stored in the buffer D.

$$\underset{w}{\operatorname{argmin}} \sum_{(s,a,s') \in D} (\bar{c}(s') - (\bar{c}(s) + g(s; w)^\top a)) \tag{4.4}$$

Eq. 4.4 shows that the objective of pre-training phase is to find the parameters w that minimize the difference between the predicted cost function $(\bar{c}(s) + g(s; w)^\top a)$ and the real one $\bar{c}(s')$, where the latter is considered as part of the state s' present in the tuple. Once pre-training is complete, the policy learning process can start.

4.3.3 Action correction during training

Every classical Reinforcement Learning algorithm can be used to train the policy, since the action of the Safety Layer does not influence the optimization process.

In fact, Safety Layer only performs a correction on the output of policy neural network, but never interacts with the parameters describing it or other functions involved in the policy learning process. In [12] DDPG (Sec. 3.4.1) has been chosen, hence in this thesis the same algorithm is kept as the preferred one.

During each step in the environment, the policy function $\mu_\theta(s)$ takes a state s as input and outputs an action a. Associated with the state s there is a cost function $\bar{c}(s)$ that describes the dangerousness in which the agent finds itself. At this point, the following optimization process takes place:

$$\begin{aligned} a^* = \underset{a}{\operatorname{argmin}} \ & \frac{1}{2}\|a - \mu_\theta(s)\|^2 \\ & \text{s.t.} \bar{c}_i(s) + g(s; w_i)^\top a \leq C_i \forall i \in [K] \end{aligned} \tag{4.5}$$

where the subscript indicates the constraint number i. Eq. 4.5 shows that the Safety Layer corrects the original action as little as possible in the Euclidean norm in order to make it safe, i.e. compliant with the constraints. Eq. 4.5 is a quadratic problem, therefore it is possible to find the global solution. In the paper a QP-solver [13] is indicated, but its application exceeds the goals of this thesis. Going one step further and making the assumption that *no more than a single constraint is active at a time* it is possible to reach a closed-form analytical solution for Eq. 4.5. This assumption is suitable in many physical domains, for example the ones in which the agent has to avoid some obstacles (since it can be too close just to an object per time). The solution found is the following (proof of its validity can be found in [12] and will be omitted):

$$a^* = \mu_\theta(s) - \lambda^*_{i^*} g(s; w_{i^*}) \tag{4.6a}$$

$$\textit{where} \quad \lambda^*_{i^*} = \left[\frac{g(s; w_{i^*})^\top \mu_\theta(s) + \bar{c}_i(s) - C_i}{g(s; w_{i^*})^(s; w_{i^*})} \right]^+ \tag{4.6b}$$

$$\textit{and} \quad i^* = \underset{i}{\operatorname{argmax}} \ \lambda^*_{i^*} \tag{4.6c}$$

If predicted cost function $\bar{c}(s) + g(s; w)^\top \mu_\theta(s)$ is above the threshold C_i, then it means that action $a = \mu_\theta(s)$ leads to a dangerous state. In this case the multiplier λ_i correspondent to that constraint is positive, hence a correction is performed (as in Eq. 4.6a); otherwise the numerator of Eq. 4.6b is negative or null and no correction is activated. In case there is only one constraint, Eq. 4.6c loses sense and only one multiplier λ is present.

4.3.4 Implementation and results

The following images and captions are taken from [12] and describe the environments used to implement the algorithm.

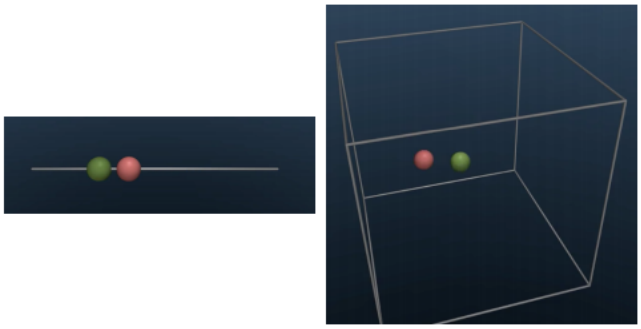


Figure 4.7: Ball1D (left) - Ball3D (right) tasks [12]

«The goal is to keep the green ball as close as possible to the pink target ball by setting its velocity. The safe region is the [0, 1] interval in Ball-1D and the [0, 1] cube in Ball-3D; if the green ball steps out of it, the episode terminates [12]»

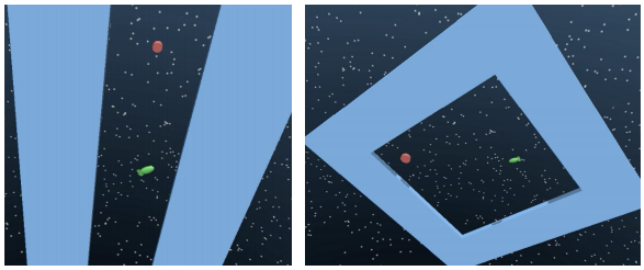


Figure 4.8: Spaceship-Corridor (left) and Spaceship-Arena (right) tasks [12]

« The goal is to bring the green spaceship to to the pink rounded target by controlling its thrust engines. Touching the walls with the spaceship's bow terminates the episode. [12]»

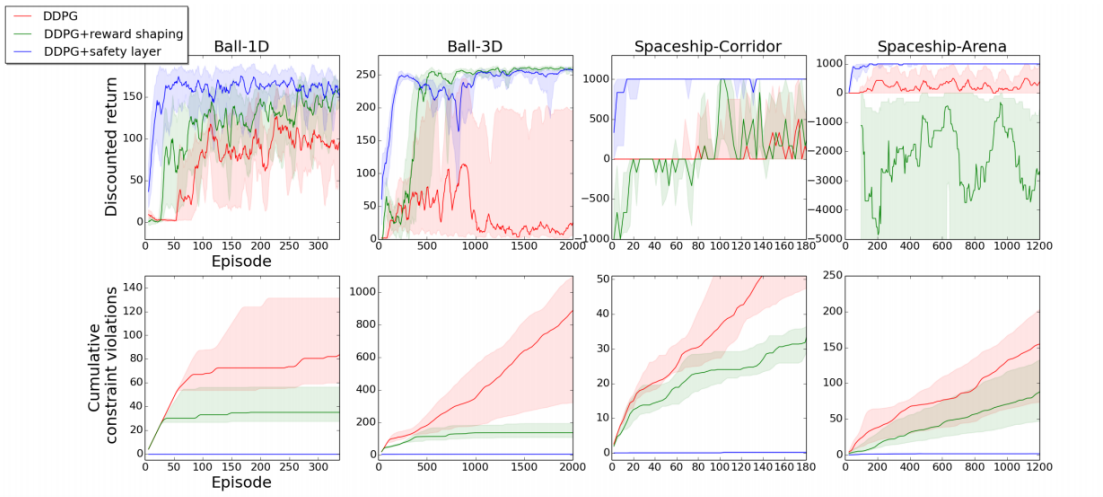


Figure 4.9: Results of [12] per each task. (Top) Sum of episodic discounted rewards. (Bottom) Cumulative constraint violations occurred during training.

Safety Layer is able to ensure safe exploration as presented in Sec. 4. In Fig. 4.9 Safety Layer is compared to unsafe DDPG and DDPG supported by reward shaping. The latter is a modification of the classical algorithm where a penalty is associated to states whose cost function violates the constraints, in order to deter the agent to reach those states later in the future. Safety Layer shows promising results both in terms of safety, both in terms of reward gained. In fact, as it can be seen from the graphs in Fig. 4.9, blue line performs well in all four environments tested, even when classic algorithm demonstrates to struggle. The authors of the [12] explicate this unexpected result in the following way: «[...] our method promotes more efficient exploration – it guides the exploratory actions in the direction of feasible policies. [12]»

This capability of Safety Layer of "guiding" the exploration process is typical of Risk-directed exploration SRL.

Chapter 5

Implementation

Before starting the detailed description of the environments and agents used for the implementation and modification of the Safety Layer algorithm, it is necessary to explicate the reasoning behind the different choices made.

The main purpose of this thesis is to obtain an algorithm that improves state-of-the-art safe exploration in some particular environments that try to represent realistic dangers configurations. These environments are the ones in which dangers fill the space in a heterogeneous way and that will be called, borrowing a mathematical term for informal use, *non-convex* environments. If *convex* environments are perfectly practicable from a large family of agents, for *non-convex* environments this is not the case since they are "drilled" by dangers that the agent should avoid.

The environments explored in the reference paper [12] were *convex* since the only dangers were the external walls and that the target was always within them. This thesis implementation started from a simple environment very similar to the ones used in the reference paper [12] and then passed to a more complex *non-convex* environment, more representative of the real world.

5.1 Environments

The two environments chosen for implementation are:

- **CartPole Continuous** as *convex* environment
- **Safety Gym** as *non-convex* environment

5.1.1 CartPole Continuous

CartPole is an environment provided by OpenAi Gym [14], corresponding to the problem treated in [15]. It consists of a cart with a pole upon it, linked by a

hinge that leaves it free to oscillate around the vertical position (Fig. 5.1); the cart itself can move horizontally towards left or right. The task to be learned by the agent is to keep the pole balanced (its angle with the vertical must remain below a threshold) moving the cart within the edges of the environment. The original version of this environment was discrete in action, meaning that the agent could decide only between -1 ($left$) and 1 ($right$), but for the experiments of this thesis, a continuous version has been used [16], in which the action space is continue and $a \in [-1,1]$. Detailed physical background and equations involved can be found in [15]. The environment is provided of reward function, but neither of

Figure 5.1: CartPole Continuous environment

constraints neither of cost function, therefore it has been necessary to implement them manually. The constraints set for the agent were:

- to not go beyond the *left edge* of the screen (1^{st} constraint)

$$x_{pos} - margin > x_{edge_{inf}} \quad \rightarrow \quad -x_{pos} + margin + x_{edge_{inf}} < 0 \tag{5.1}$$

- to not go beyond the *right edge* of the screen (2^{nd} constraint)

$$x_{pos} + margin < x_{edge_{sup}} \quad \rightarrow \quad x_{pos} + margin - x_{edge_{sup}} < 0 \tag{5.2}$$

The cost function has been rewritten adapting the one used in the reference paper [12]:

$$c_1 < C_1 \quad \text{with} \quad c_1 = -x_{pos} + margin + x_{edge_{inf}} \quad \text{and} \quad C_1 = 0 \tag{5.3a}$$

$$c_2 < C_2 \quad \text{with} \quad c_2 = x_{pos} + margin - x_{edge_{sup}} \quad \text{and} \quad C_2 = 0 \tag{5.3b}$$

In Eq. 5.3a and Eq. 5.3b, the following terms indicate:

- x_{pos}: the position of the cart on the horizontal rail, going from negative values (left part of the screen) to positive values (right part of the screen)
- $margin$: constant value required to consider some "inertia" and give the agent time to counterattack the danger
- x_{edge}: constant value that indicates where the end of the screen is located on the x axes (i.e. the edge not to be crossed)

Fig. 5.2 shows a clarifying visualization of CartPole Continuous constrained environment.

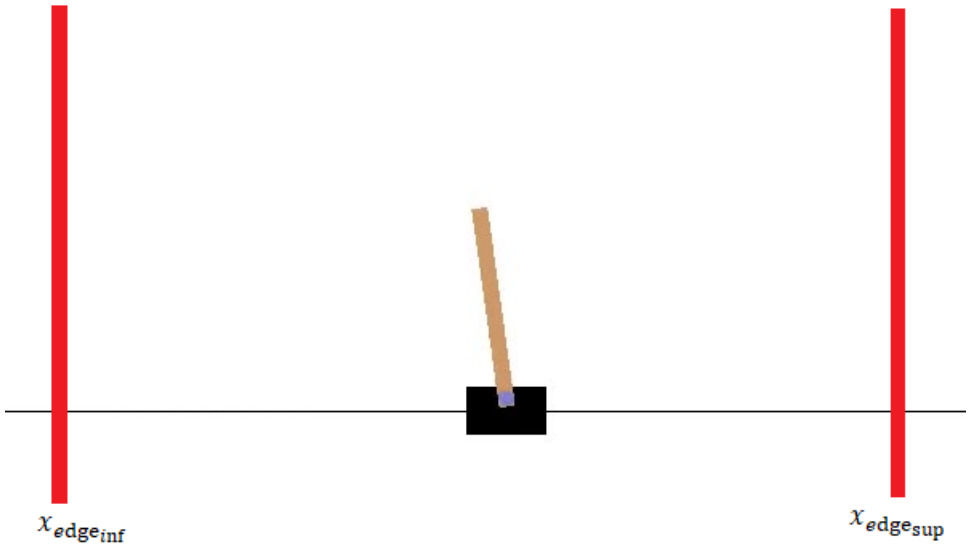


Figure 5.2: CartPole Continuous constrained environment

5.1.2 Safety Gym

The second environment selected to host the experiments has been Safety Gym. Developed by OpenAI, it is specifically designed to support the benchmarking of Safe Reinforcement Learning inherent algorithms.

Safety Gym is a suite that offers several tools and items, configurable in a variety of ways that enable a multitude of experiments. Three tasks built upon three levels offer the baseline configurations for the three agents available, even if every feature and item is customizable. In the following Sections, the different configurations will be presented in detail.

5.1.2.1 Tasks

Goal

The objective is for the agent to reach a *target zone* in the state space as soon as possible, the target changes each time that an episode ends or the agent succeeds.

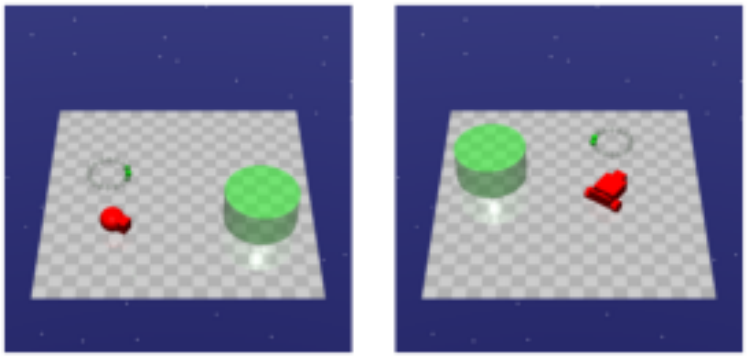


Figure 5.3: Goal task

Button

The objective is for the agent to hit a *target button* while avoiding fake buttons, the target button changes each time that an episode ends or the agent succeeds

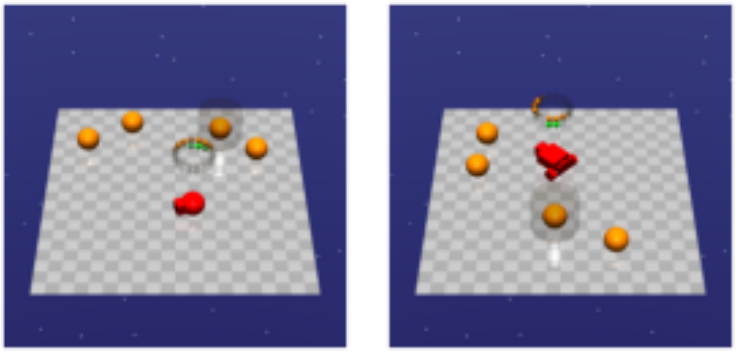


Figure 5.4: Button task

Push

The objective is for the agent to push a *box* towards a *target zone*, the location of the box and of the target zone changes each time that an episode ends or the agent succeeds

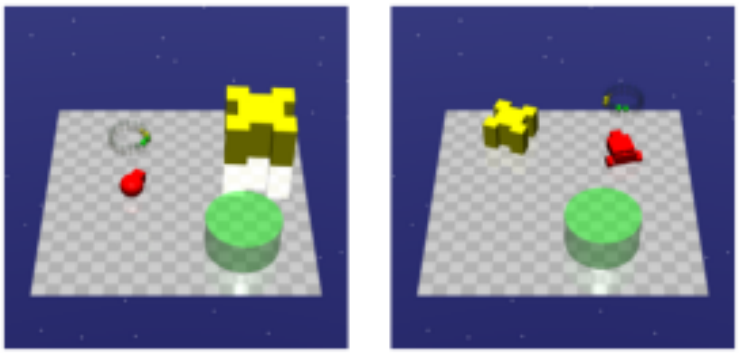


Figure 5.5: Push task

5.1.2.2 Agents

Point

It is the simplest of the three agents available. It is provided of two actuators: one for turning and another for moving forward or backward. The action is bi-dimensional $a = a_1, a_2$ where $a_1, a_2 \in [-1,1]$. This agent is fixed in the plane and does not have possibility to move in z direction.

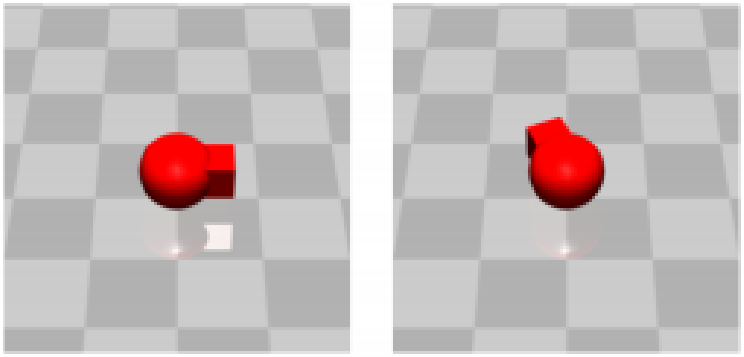


Figure 5.6: Point agent

Car

This agent is more realistic than Point since it is not fixed on the bi-dimensional plane, in fact it can move in z direction, if forced by the environment. It is provided of two actuators: one for turning and another for moving forward or backward. Both the actuators have to be coordinated to permit movement. The action is bi-dimensional $a = a_1, a_2$ where $a_1, a_2 \in [-1,1]$.

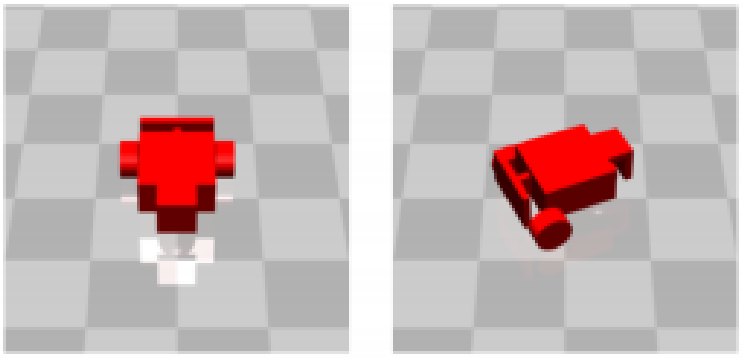


Figure 5.7: Car agent

Doggo

It is the most complex agent available, it has bilateral symmetry and it is controlled by 12 actuators: three for each of the four leg. As for the other agents, it needs to learn how to move before starting learning the task; this objective is already difficult, differently from the other agents.

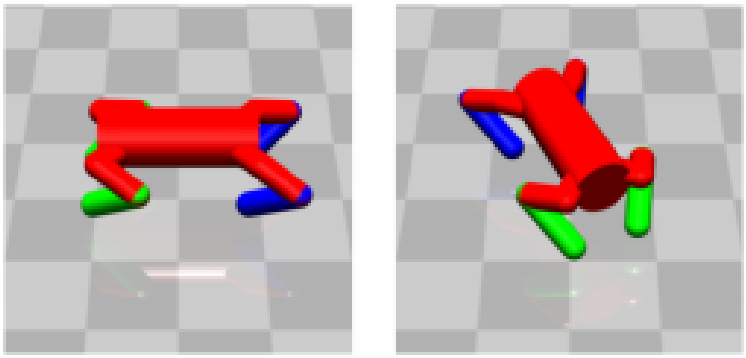


Figure 5.8: Doggo agent

5.1.2.3 Levels ans obstacles

In Safety Gym it is possible to insert several objects as obstacles to the agent, each of them with different customizable features. Also cost function is completely customizable and it can be tuned as sparse or dense. Safety Gym uses sparse cost function as a baseline, meaning that cost is 0 when constraints are not violated and 1 when a violation occurs. For this work, cost function has been set dense and represents the distance of the agent from the closer obstacle (changed of sign and

with some margin as in Sec. 5.1.1).

$$c < C \quad \text{with} \quad c = -distance + margin \quad \text{and} \quad C = 0 \tag{5.4}$$

No obstacles - Level 0

Level "zero" is basic, no objects that hinder the robot are present and the only objective is for the agent to learn the assigned task. This level is useful for debugging purposes and to focus on the agent learning capabilities. Fig. 5.3, Fig. 5.4, Fig. 5.5 represent this level, free of any obstacle.

Single kind of obstacle - Level 1

In level "one" two kinds of obstacles are inserted, but only one is "activated" with constraints. One element of the "inactive" obstacle is present and eight elements of the "active" one. The former is a rigid *box* that can be pushed and moved by the agent and no cost is associated to it, the latter are *hazardous areas* that the agent should avoid while learning the main task, indeed a cost is associated with them. These obstacles are not rigid, meaning that the agent can invade them but cannot move them.

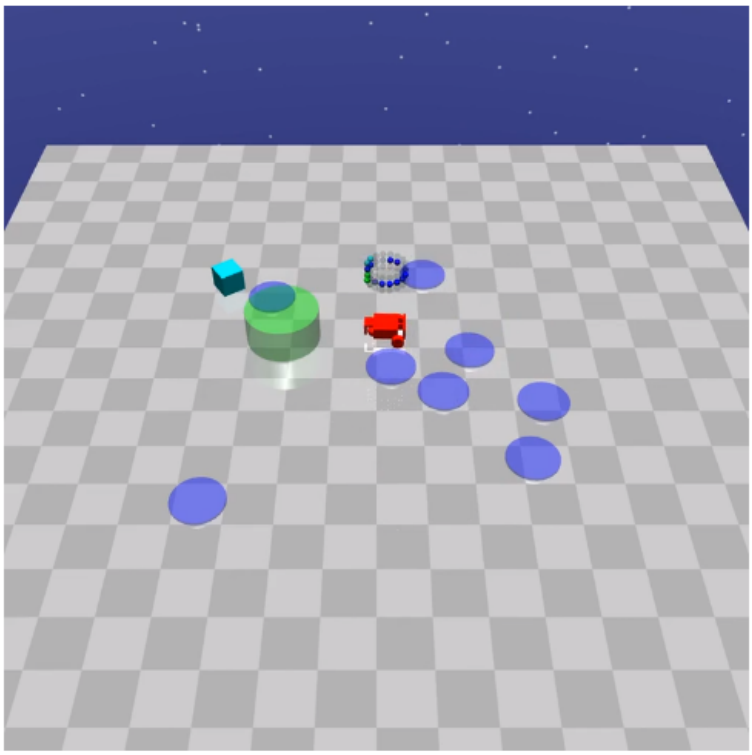


Figure 5.9: Level 1

Different kinds of obstacles - Level 2

This is the most complex level of the pre-configured ones. Several types of obstacles are present, different in kind and numbers for each baselined configuration and main task. For the *goal* task, level 2 consists in inserting ten *hazards* (two more than level 1) and ten *boxes* (nine more than level 2). Every type of obstacle has an associated cost and is constrained, the agent has to avoid each of them while learning the main task.

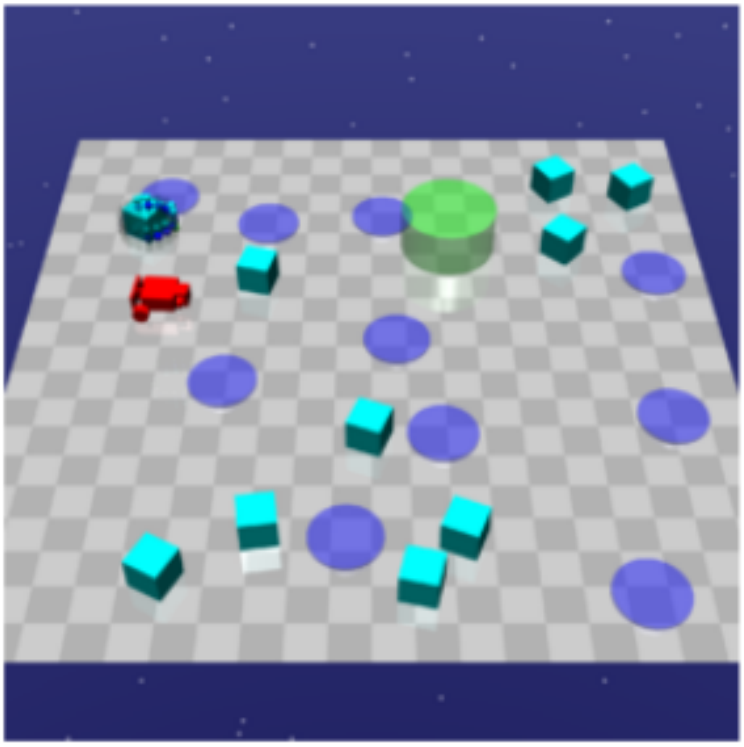


Figure 5.10: Level 2

5.1.2.4 Sensors

Agents are provided of different sensors in order to be conscious of the surrounding environment, each of them measures some features and these values are part of the state vector, which is the input of the policy neural network. The sensors available are:

- *accelerometers*
- *gyroscopes*

- *magnetometers*
- *velocimeters*
- *joint position trackers*
- *joint velocity trackers*
- *compasses* for pointing to goals
- *lidars* (one for each kind of object present in the environment)

Compasses and lidars are renderizable, in order to provide to humans a clear visualization of what the agent is perceiving at that moment.

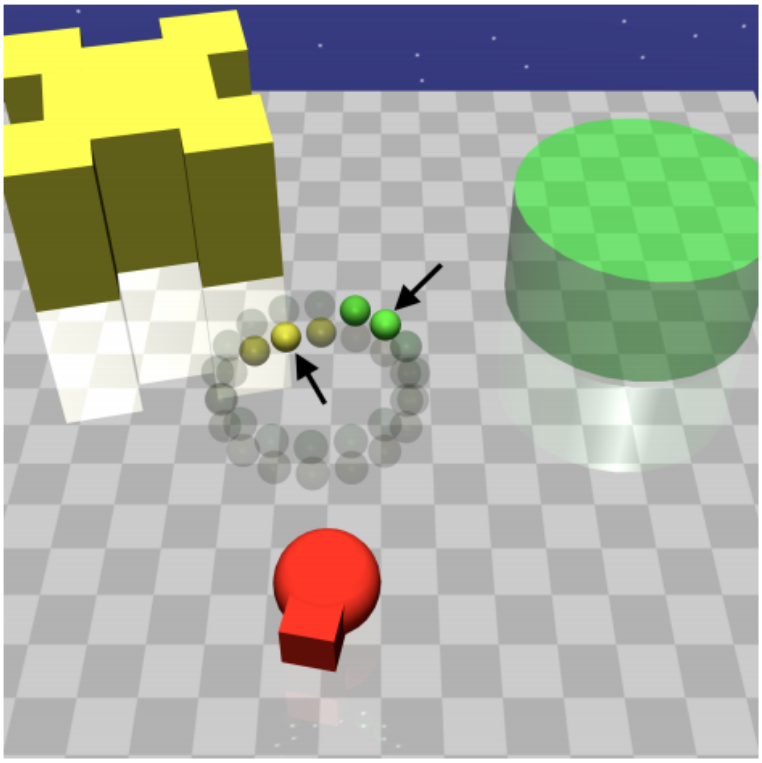


Figure 5.11: Lidar rendering, yellow for the *box* and green for the *target zone*

5.1.2.5 Chosen configurations

Two options have been chosen for the implementation part of this thesis.

In the first configuration (*Configuration 1* from now on) the purposes were to understand limits of the algorithm and its applicability to Safety Gym suite, to identify which metrics to store, to tune specific hyperparameters.

Configuration 1 features are:

- *Task*: Goal
- *Agent*: Point
- *Level*: 1
- *Sensors*: accelerometers, gyroscopes, magnetometers, velocimeters, lidars for target and obstacles

The second configuration (*Configuration 2* from now on) is the one used for the effective improvement phase, in which some features of the algorithm have been modified and the effects of these changes have been analysed.

Configuration 2 features are:

- *Task*: Goal
- *Agent*: Car
- *Level*: 1 (without inactive box)
- *Sensors*: accelerometers, gyroscopes, magnetometers, velocimeters, lidars for target and obstacles

5.2 Algorithm baseline structure

The baseline structure of the algorithm can be resumed as:

- *pre-training exploration*: the agent collects samples about the surrounding environment following a random policy
- *pre-training learning process*: offline optimization of the SL network
- *policy learning*: each step of this phase is also divided in:
  - *action selection*: the policy network takes the current state as input and provides an action
  - *risk estimation*: the SL network takes as inputs the current state, its associated cost function and the action selected and estimates the future cost function
  - *action correction*: if the cost function predicted exceeds the threshold, the SL corrects the action towards a safe one
  - *action execution*: the agent follows the corrected action and prosecutes in the exploration phase

– *policy optimization (if any)*: at determined steps, optimization of the policy parameters is performed

The previous structure is the one used for the experiments performed on *CartPole* environment, *Safety Gym Configuration 1* and as the starting point for *Safety Gym Configuration 2*. In the latter environment indeed, some variations have been actuated in order to improve the performances.

Detailed pseudo-code of the algorithm can be found in Appendix A.

Chapter 6

Results

In this Chapter, the results of the different experiments performed will be presented and analysed. For the experiments presented in Sec. 6.1 and Sec. 6.2 only the baseline algorithm has been applied, whose structure can be found in Appendix A. In Sec. 6.3 the reader will find the explanation of the issues encountered during training and the variations applied to solve them, such as:

- cleansing of pre-training database
- modification of pre-training process
- transition to sparse reward
- insertion of reward shaping, a particular technique that operates on the episodic reward in order to guide the agent in the exploration process

Each curve reported in the following graphs represents the average of three identical experiments with different seeds, the shaded area represents the interval built with standard deviation.

6.1 CartPole Continuous

As previously explained in Chapter 5, the implementation started with *CartPole Continuous* environment to understand potentialities of the algorithm, debug the code, acquire knowledge on the logic behind it, verify functionalities. Since this environment is not provided of its own constraints, it has been necessary to construct some and associate them with a cost function; the constraints were for the cart to not run outside of the borders of the screen and the cost function was dense, representing the distance from the borders changed of sign (detailed explanation in Sec. 5.1.1).

Several experiments have been done, tuning the different parameters of DDPG (learning rates, γ, τ, buffer size), SL neural network and pre-training (size of the NN, batch size, number of samples, optimization epochs). The results found were in line with the ones stated in the SL reference paper [12]: the algorithm converges to an optimal policy, with no violations of constraints during training (Fig. 6.3, Fig. 6.1 and Fig. 6.2). Fig. 6.1 and Fig. 6.2 show the total violations done by

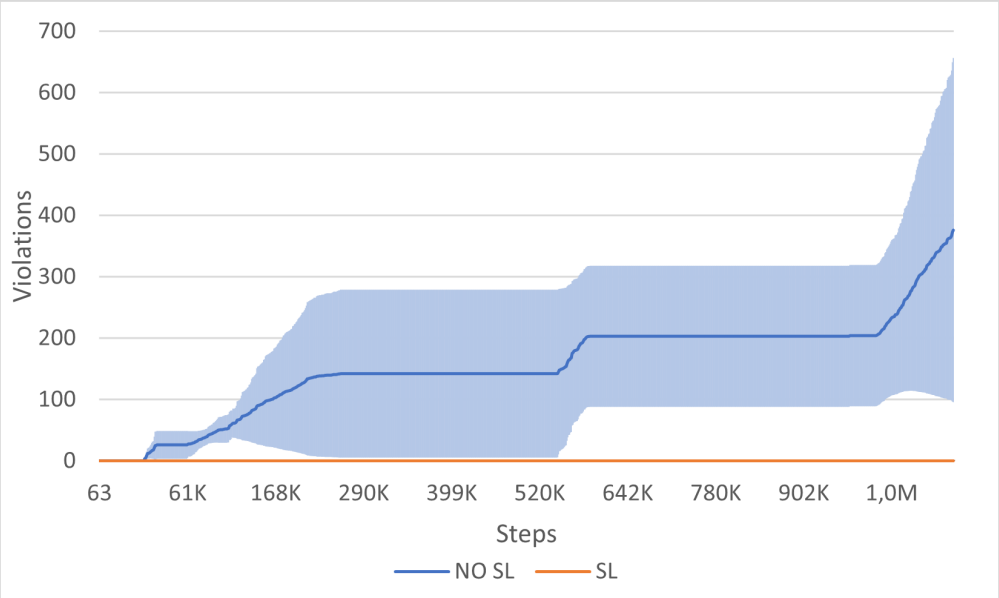


Figure 6.1: CartPole - Cumulative violations of max constraint during training

the agent during the training phase, both with safe and unsafe approaches. It can be seen that without SL, the agent is not protected from taking unsafe actions, indeed it has to commit them in order to learn that they are provided by a bad policy. Indeed, violating constraints means ending the episode and preventing the reward growing higher. For this reason, a policy that does not avert this behaviour cannot be an optimal policy. In unsafe DDPG, the agent eventually learns an optimal policy that avoids the edges of the screen, as can be seen in Fig. 6.3 where both processes end in an optimal behaviour, but it is not because it is conscious about some kind of "safety" of the environment. The agent stays far from the constraints because the optimal policy - keeping the pole vertical for as long as possible - implicates not running outside the screen, in order to not terminate the episode and make more reward. The optimal policy is itself implicitly safe. This incapability of learning implicitly unsafe policy is one of the limit of the baseline algorithm, as it will be explained more in details in further Sections.

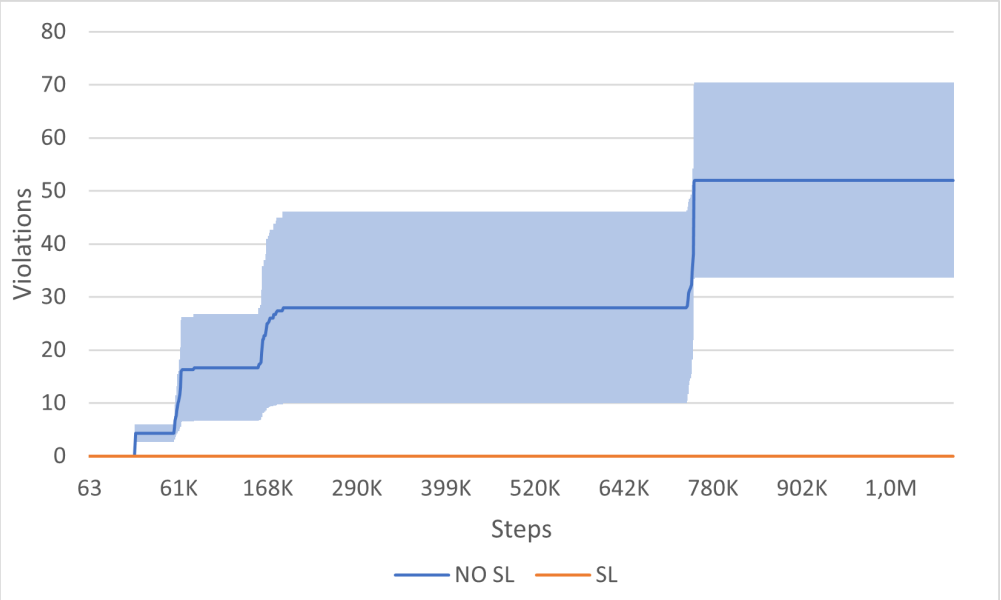


Figure 6.2: CartPole - Cumulative violations of min constraint during training

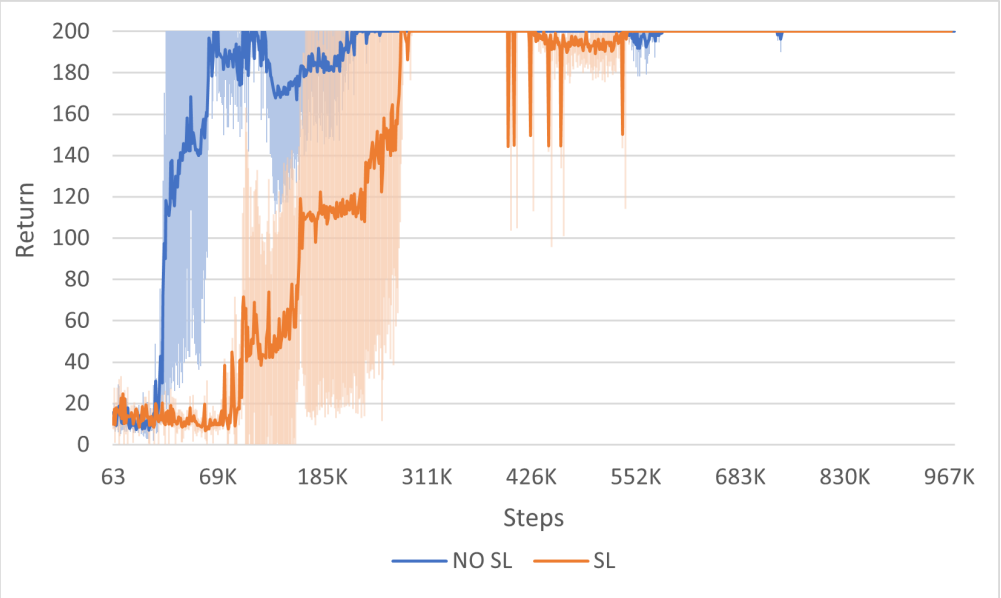


Figure 6.3: CartPole - Episodic reward during training

6.2 Safety Gym - Configuration 1

After having verified the functionalities of the algorithm on the basic *CartPole Continuous*, the SL layer has been applied to a more complex environment, namely *Configuration 1* of *Safety Gym* (*Point* agent). Although this is not the final environment where the majority of the experiments have been performed, due to its low complexity, it has been useful to test the application of SL at this level before deploying it on the definitive configuration. In this way, indeed, the most serious issues have been discovered and in the successive phase it has been possible to start investigating about solutions. In the followings, three processes will be presented:

- **NO SL**: DDPG without Safety Layer
- **SL suboptimal** : DDPG with a suboptimal Safety Layer, where optimization of the *margin* was not tuned properly
- **SL optimal** : DDPG with optimal Safety Layer

It can be useful to remind that the *margin* is a constant to be added to the constraint equation in order to take into account some "inertia" of the agent (Eq. 5.3a for details).

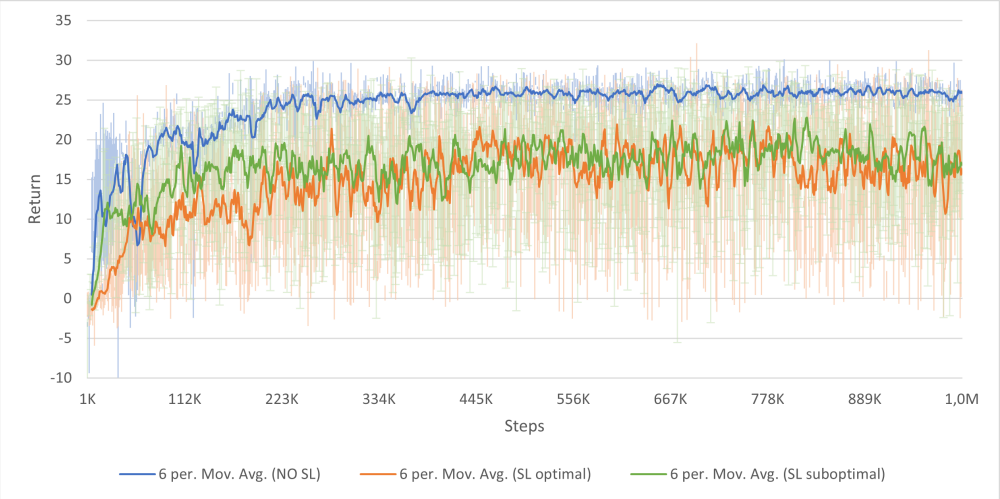


Figure 6.4: Configuration 1 - Episodic reward during training

Fig. 6.4 shows that unsafe DDPG converges to an optimal policy after 300k steps (interactions with the environment), reaching a reward of 25 (i.e. 4/5 catches of the target for each 1000 steps episode). It is evident that the application of the SL affects negatively the agent performances, it is promptly explained in the

followings. By default, the agent would look for the shortest way to get to the target, and it is well know that the shortest way between two points is the straight line that connects them. This is exactly what the agent does in the unsafe case: it turns itself and goes straight to the target, without caring about invading hazardous areas on the road. When the SL is applied, this behaviour is modified by the correction performed, that gives precedence at preventing the agent from violating the constraints (namely stepping inside hazardous areas) rather than leading it towards the target. To confirm this thesis, in Fig. 6.5 it can be seen that during the evaluation episodes, where both the safe and unsafe algorithms are performed without SL, the reward obtained by the agents reaches similar levels. This validates the idea that the correction action of the SL is the responsible of the degradation of performances.

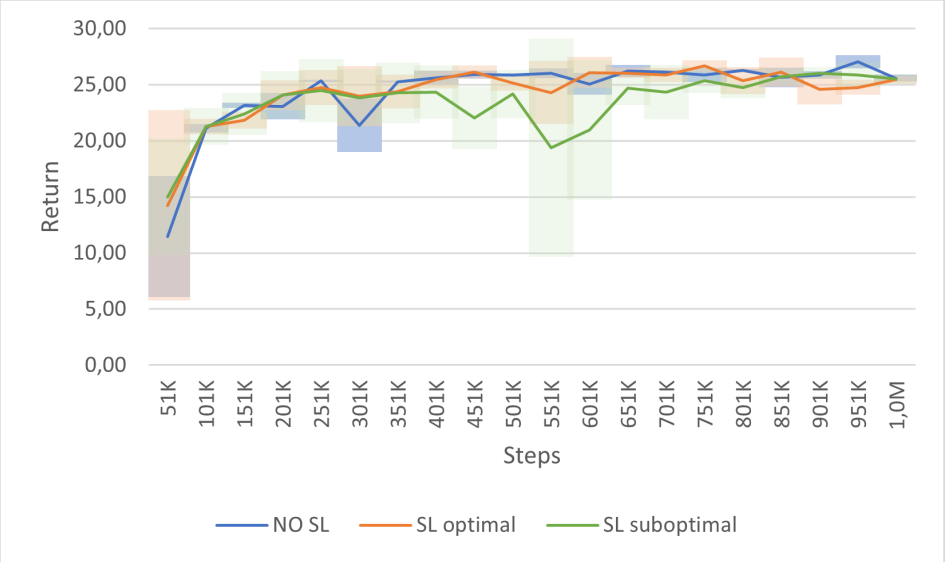


Figure 6.5: Configuration 1 - Episodic reward during evaluation

Having a look on the safety performances, more issues arise. Looking at Fig. 6.6, it can be seen that no mayor differences are present between the safe and unsafe algorithms, meaning that although the SL executes some corrections on the policy actions, this is not enough to significantly prevent the agent from violating constraints. On the other hand, a suboptimal SL can even increase the violations due to its incapability to promptly stop the agent before stepping inside hazardous areas, not taking properly into account inertia.

Investigating more in detail, it can be seen that the effect of SL is more important in the early phases of the training, where instead the unsafe algorithm performs a huge number of violations. In Fig. 6.7 it can be seen that both safe and unsafe processes reach a similar level of violations during the majority of the training, but

in the early stages a decreasing of the 80% is appreciable with an optimal SL.

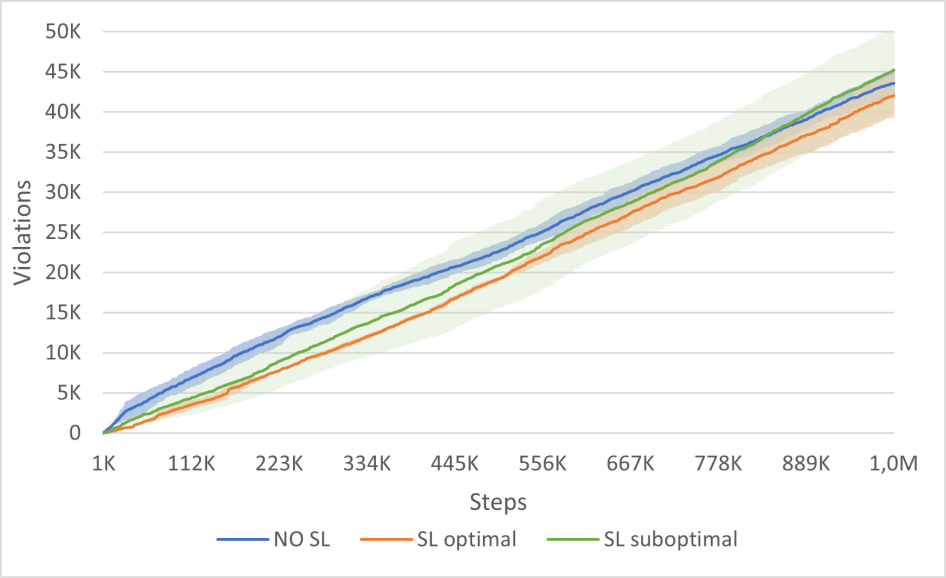


Figure 6.6: Configuration 1 - Cumulative violations of constraint during training

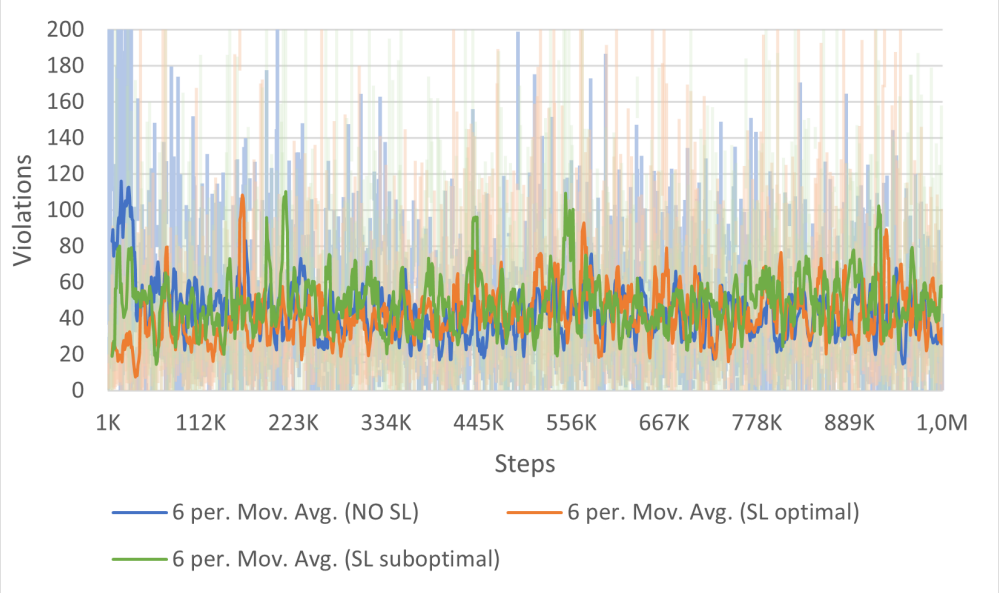


Figure 6.7: Configuration 1 - Episodic violations of constraint during training

Issues and challenges

From the results obtained during experiments on *Configuration 1* environment, several problems arose and were challenged with different tentative solutions. The issues concerned both reward and safety performances and are:

1. learning of an unsafe policy once removed the SL
2. learning of a suboptimal policy
3. scarce dependency of the SL neural network with respect to the current state, leading to almost constant $g(s; w)$ values (from now on, dependency upon parameters w will be omitted in the writing for the seek of simplicity)
4. SL occasionally providing not feasible actions (not in the range $[-1,1]$)

Learning of an unsafe policy

The original idea of the application of a SL during training was to protect the agent until it learned a policy safe enough, then it could have proceeded without needing shielding anymore. It can be seen from Fig. 6.7 that there is no decreasing in the violations pattern during training, meaning that the policy does not learn any safe behaviour. This is the consequence of what already mentioned in Sec. 6.1: SL succeeds in finding an optimal safe policy only when the optimal policy is intrinsically safe. An explanation of this behavior could be that SL does not affect the learning process of the policy, but it just prevents the agent to run in dangerous states during exploration; when learning a policy that naturally leads away from dangers, SL will eventually not be called anymore, but when this does not happen SL will continue to be needed and to impact on main policy performances.

Learning of a suboptimal policy

It can be seen from Fig. 6.4 that, nonetheless safety performances are not significantly improved, there is a degradation in terms of reward obtained when applying the SL. This is a direct consequence of inserting obstacles in the space of the environment: the optimal policy should not lead on the shortest way to the target, but rather to circumnavigate the obstacles and lengthen the trajectory. Without any safe constraints, the agent learns only to reach the target without considering the hazardous areas, which do not impact in any way on the agent. When these obstacles have to be avoided, they become rigid in the point of view of the agent and they start to affect its trajectory, blocking it and precluding some roads that would otherwise be accessible. Therefore, the degradation of policy performances is natural in some limits and should not be considered a consequence of SL application.

Scarce dependency of the Safety Layer upon the current state

In order to help the reader understand this issue, it may be useful to repeat that the SL neural network is the $g(s)$ function, which describes the variation of the cost when a specific action is taken in a specific state (Eq. 4.3). The original idea was to have different values for different states, in order to be able to approximate a large variety of responses to the action provided by the policy and, consequently, a wide range of safety corrections (Eq. 4.6a). The issue arose concerns the output of the $g(s)$ function, which is supposed to variate with respect to the state space, but that during experiments resulted quite constant. The scarce dependency of the $g(s)$ function upon the current state involves a linearity of the cost function with respect to the action, which is not theoretically wrong, but that limits the correction capability of the SL: very dangerous and slight dangerous actions are treated in the same way. For this reason, the behaviour found in the renderings was of an agent that stops itself in proximity of the hazardous areas and gets stuck there, trapped by the same SL that protects it. The SL does not distinguish a slight dangerous action, useful to escape from the edge of the hazardous area, from a very dangerous one, that would lead it inside the said zone. An extreme consequence of trapping the agent on the border of the obstacle is that it will eventually violate it and get stuck there; when this happens it is possible to have spikes in the episodic violations (Fig. 6.7), although those violations are quite small in magnitude.

Safety Layer correction providing not feasible actions

This issue is strongly connected with the previous one, since it depends on the outputus provided by the $g(s)$ neural network. In every environment considered, every component of the action is in the range $[-1,1]$, characteristic that can be easily achieved using hyperbolic tangent (tanh) as activation function of the last layer of the policy network. When the correction of the SL is applied to a dangerous action, the safe output is the solution of Eq. 4.6a, which lies in the range $[-\infty, +\infty]$ and is clipped in $[-1,1]$ in order to provide the agent with a compatible input. Unfortunately, this operation leads to a loss of information, since a value slightly larger than the upper bound is truncated in the same way of a value that strongly exceeds it. The visible consequence of this issue is that when the agent is located in a very dangerous state and it should perform a strong safe action, eventually it executes just the largest action possible, that cannot save it. Hence, in some situations it is very hard for the agent to act towards a safe state.

To solve such problems, different strategies have been followed, impacting both training and pre-training processes, but also environmental structure, reward and cost functions.

6.3 Safety Gym - Configuration 2

In the *Configuration 2* of *Safety Gym* the number and nature of obstacles is the same as in *Configuration 1*, but the agent is *car*. Some preliminary experiments have been done to tune the parameters of the different operators and, after having identified the features of an optimal SL, the modification of the algorithm started, in order to improve performances in terms of reward obtained and safety. This last environment is also the one that has been explored the most, in order to find solutions to the problems explained in Sec. 6.2, whose proposed solutions will be explained in the following Sections.

6.3.1 Scarce dependency of the SL upon the current state

6.3.1.1 Cleansing of the pre-training database

The first modification conducted to solve the issue was about the SL pre-training process, cleaning the database of tuples ($< s, a, s' >$) upon which the optimization of the $g(s)$ network is based. During pre-training, the agent is governed by a random policy, as explained in Sec. 4.3.2, and for this reason it is prone to make a lot of violations if left free to move for long episodes. Hence, with the purpose of reducing this risk, the episodes are truncated in a short time (20/30 steps), allowing the agent to explore the environment without giving it the time to invade hazardous areas. As a matter of fact, this collection of samples was strongly dominated by similar values of cost representing safe states, in which the agent was located for most of the time. This led to the idea that cleaning the database could have helped the optimization process of the SL, making it more variable with respect to the input state.

To find a trade-off between risky exploration and useful samples collection, the number of steps for each episode have been increased and the database has been cleaned of a large number of similar values, resulting in a more uniform one (same number of safe and unsafe states). Unluckily, this cleaning process did not show improvements in performances, and it was found that short episodes are enough to obtain a good approximator, with a number of samples in the order of 10^5.

No modifications have been kept for the future.

6.3.1.2 Dependency of the Safety Layer network upon the action

This modification consisted in reviewing the whole SL concept: while it usually takes only the current state as input, in this upgrade it took also the action provided by the policy.

$$g(s) \rightarrow g(s, a) \rightarrow g(s, \mu(s))$$

In this way it was introduced an indirect dependency of the SL on policy parameters. As a consequence of this modification, $g(s, a)$ outputs are less constant than before and this permits an higher flexibility to the SL, in particular to the correction executed on dangerous actions. It was seen on renderings that the agent does not get stuck on the edges of the hazardous area as often as before, but this is at the cost of the precision of the SL, that is not activated in correspondence of every obstacle and at times lets that the agent invades them. This behaviour leaves the agent freer to explore the environment and find other trajectories to the target. The results are of a better policy in terms of reward obtained, but a poorer one in terms of safety.

Since this work is mostly focused on improving the safety, also in spite of a loss of reward, no modification have been kept for the future.

6.3.2 Learning of unsafe policy

This issue leads to the most critical consequences on the performances of the selected algorithm, it factually states the failure of the SL in granting a safe exploration during training. For this reason, most of the efforts have been focused on trying to solve or at least mitigate this problem.

6.3.2.1 Conversion of hazardous areas into rigid obstacles

The first tentative made has been substituting the dangerous areas with rigid walls (pillars in *Safety Gym*) that cannot be invaded by the agent. In this case, the violation occurred whenever the agent touched the pillar. The idea was to force the agent to avoid the "fastest wrong way" (going straight to the target), prompting it to learn the "fastest correct way" (circumnavigating obstacles without touching them). No evident improvements have been observed during the evaluation episodes, which are identical for each experiments and involve the use of nonrigid obstacles. On the contrary, worst results in terms of violations have been found. When using hazards (no rigid areas), the SL used to stop the agent, that started to turn on itself and get stuck on the edge of the obstacle; with pillars (rigid walls) the SL provide a correction of the action that "rebounds" the agent in order to avoid it to touch the obstacle. In this scenario, the agent learns to leverage on the "rebound" provided by the SL to turn itself towards the target, so it does not learn to avoid the obstacle, on the contrary it looks after them to get the "rebound" it needs; this often leads to larger violations than before.

No modifications have been kept for the future.

6.3.2.2 Continue training of the Safety Layer

Another strategy used to solve the problem of the agent not learning a safe policy was to perform the training of the SL not only in the pre-training phase, but also during the DDPG optimization process. In this way, the SL neural network could benefit of a database of new trajectories, drawn by an optimized policy and not a random one. In this way, performing an optimization of the SL on the current capability of the agent, the hope was to obtain a weakier effect of the correction in the later epochs that could have gradually lead to an independence of the agent from the SL. No improvements were found, on the contrary if during pre-training

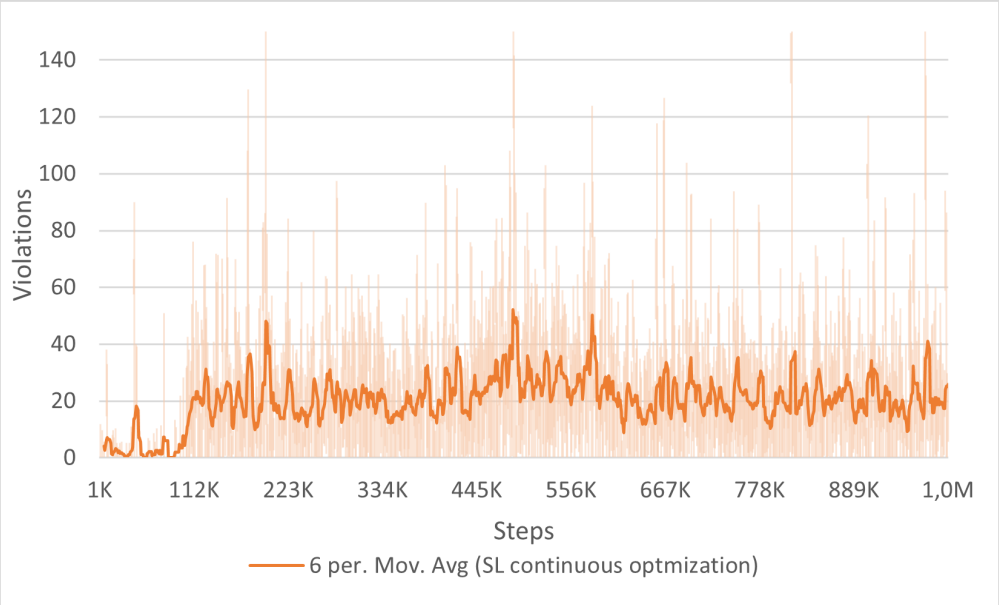


Figure 6.8: Configuration 2 - Episodic violations of constraint during training when continuous optimization of the SL is performed

the $g(s)$ network was optimal, an high risk arose of spoiling it with successive iterations. It can be seen in Fig. 6.8 a consistent increasing of episodic violations in correspondence with an update of the SL during training ($\sim 100k \quad step$).

No modifications have been kept for the future.

6.3.2.3 Retention of the Safety Layer during the evaluation

Until this moment, SL was always removed during evaluation episodes, in order to analyse the capability of the policy learned to follow safe trajectories. Safe and unsafe algorithms always had very similar trends in terms of reward obtained and also in violations performed: without SL it was evident that the agent alone could

not fulfill safety requirements since it was acting exactly as its unsafe counterpart. It was clear also looking at the various charts of episodic and total violations committed during training: no decreasing patterns were present, meaning that the policy itself did not learn to act safely. For this reason it has been chosen to retain the SL during evaluation episodes as a concrete feature of the final policy to be deployed, in order to provide protection to the agent not only during training, but in general.

No real modification have been executed on the algorithm with this strategy, except that from this point the SL is not considered anymore a temporary addition to the main algorithm (DDPG), but one of its consistent features.

6.3.2.4 Application of reward shaping

Reward shaping is a technique used in classical Reinforcement Learning to guide the agent during the training process. It relies on the idea that some peculiarities of the environment are well known and it consists in manipulating the reward obtained by the agent based on some heuristics. In the case treated in this work, reward shaping consists in the insertion of a penalty on the reward whenever the SL is used, in order to teach the agent to gradually become independent from it. In particular, if SL is used, a penalty of -1 is added once per step to the current reward accumulated by the agent during that episode.

Detailed structure of the modified algorithm can be found in A.

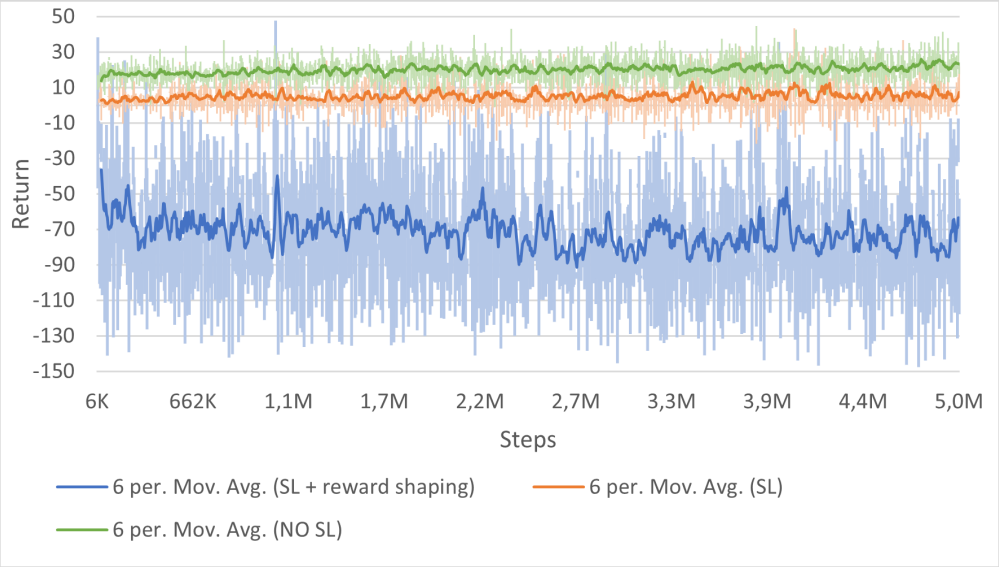


Figure 6.9: Configuration 2 - Episodic reward during training

This led to several changes either in the visualization either in the analysis of

the results: reward during training is heavily penalized by the reward shaping (Fig. 6.9). A clearer representation of agent performance is the *success percentage* during evaluation, consisting in the ratio between targets reached upon total targets of the epoch: each evaluation epoch consist in five episodes (terminating when the target is reached or after 1000 steps), hence five possibility to catch the target; if the agent catches 4 targets, its *success percentage* is of $\frac{4}{5} = 80\%$ (e.g. Fig. 6.14, Fig. 6.16, Fig. 6.18). Furthermore, it can be useful to visualize violations happened during evaluation episodes too, to understand how the retention of the SL and the reward shaping impact on the safety performances of the learned policy.

Three algorithms will be compared in the followings:

- **NO SL**: DDPG without Safety Layer
- **SL**: DDPG with optimal Safety Layer
- **SL + reward shaping**: DDPG with optimal Safety Layer and reward shaping

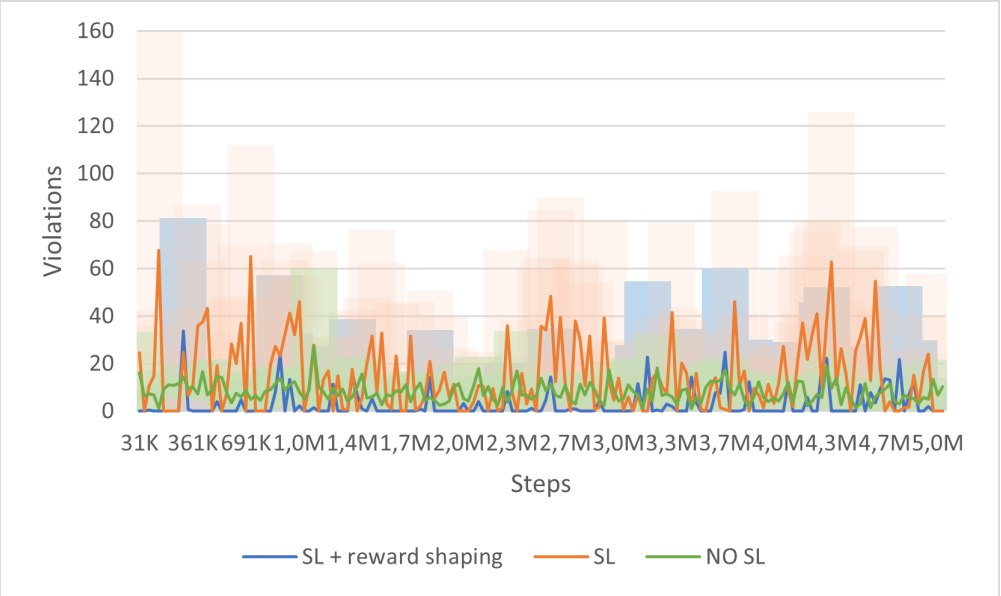


Figure 6.10: Configuration 2 - Average episodic violations of constraints during evaluation

Starting from the episodic violations during evaluation, in Fig. 6.10 it can be seen that unsafe DDPG performs a rather constant number of violations for each episode, while the SL algorithm has a more oscillating trend, that varies from zero to a large number of violations. This behaviour confirms the observation previously done about the agent getting stuck on the edges of the obstacle, that leads to an

increase of violations for that episode. As a last observation, the average of SL and NO SL algorithm episodic violations is similar. Looking instead to the upgraded version of SL algorithm, it can be observed that it is equal to zero always except for just a few occasions, testifying the capability of the SL to keep the agent far from dangers.

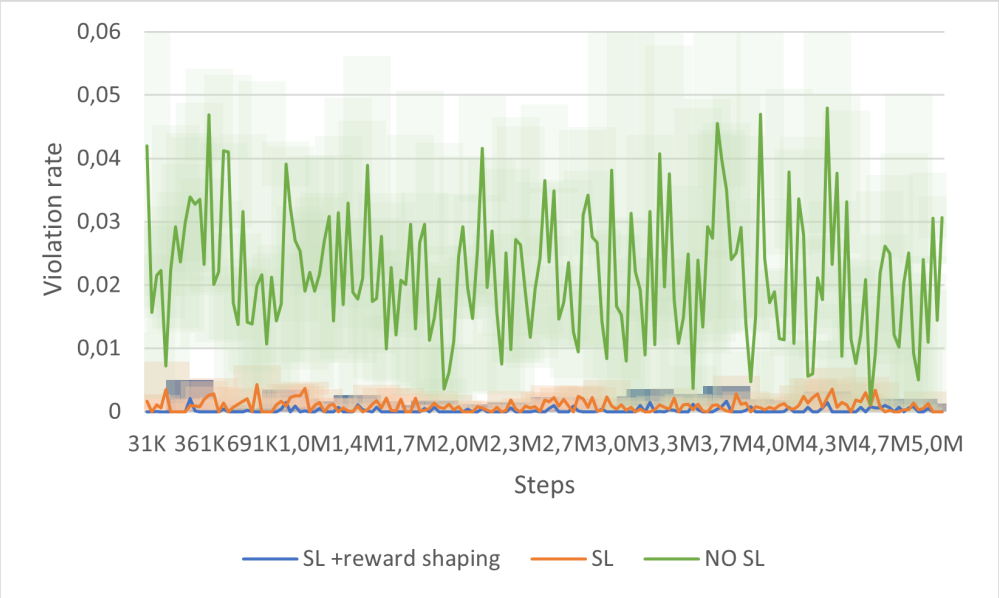


Figure 6.11: Configuration 2 - Average episodic violation rate during evaluation

In Fig. 6.11 a magnitude of the violations executed is reported, representing how heavily the agent invaded the hazardous areas. It is evident that, when applying the SL, this rate hugely decreases, validating the hypothesis that the violations are performed mostly on the borders of the obstacles (i.e. the agent is trapped and keeps turning on itself to get free, occasionally violating the constraints); moreover, the violations with SL and reward shaping are even lighter.

Focusing now on the training part, it can be seen in Fig. 6.12 that a massive decrease of violations during training is obtained when using SL + reward shaping, indeed in some cases a safe exploration with no violations has been achieved. This behaviour is testified by Fig. 6.13, where the episodic violations is represented and that shows no initial spikes in none of the following stages:

- **SL pre-training**: no spike during this early phase, meaning that the SL neural network can be optimized even with a random policy and without the need of performing violations to know the environment
- **DDPG training**: no spike, confirming that the agent do not need to violate the constraints to recognize dangerous states

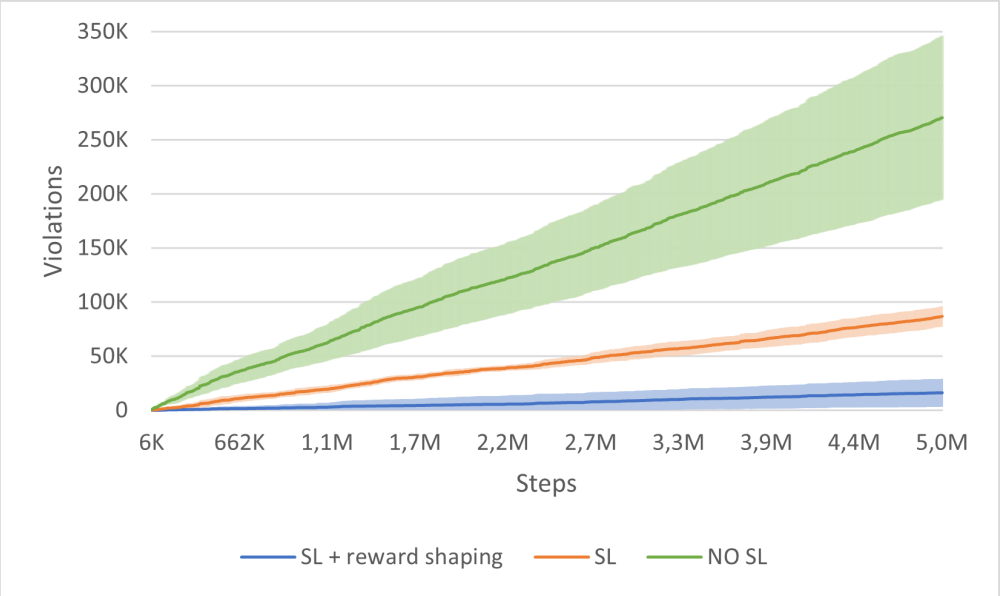


Figure 6.12: Configuration 2 - Cumulative violations of constraint during training

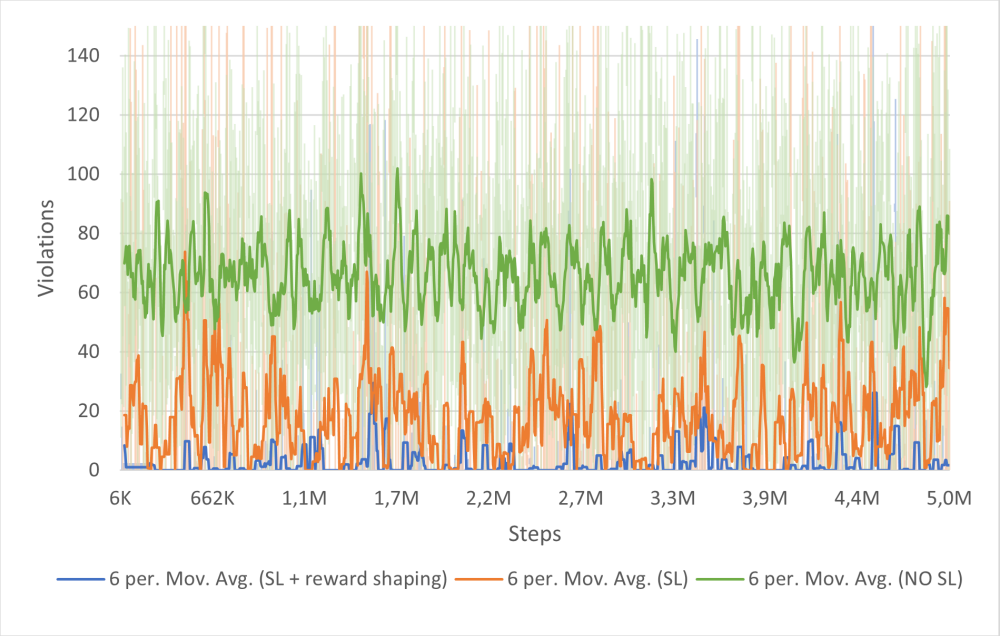


Figure 6.13: Configuration 2 - Episodic violations of constraints during training

At this point, it may be useful to concentrate on the task success, with the purpose to understand if the improvements in terms of safety are followed by actor performances at least as valid as the ones of the original algorithm.

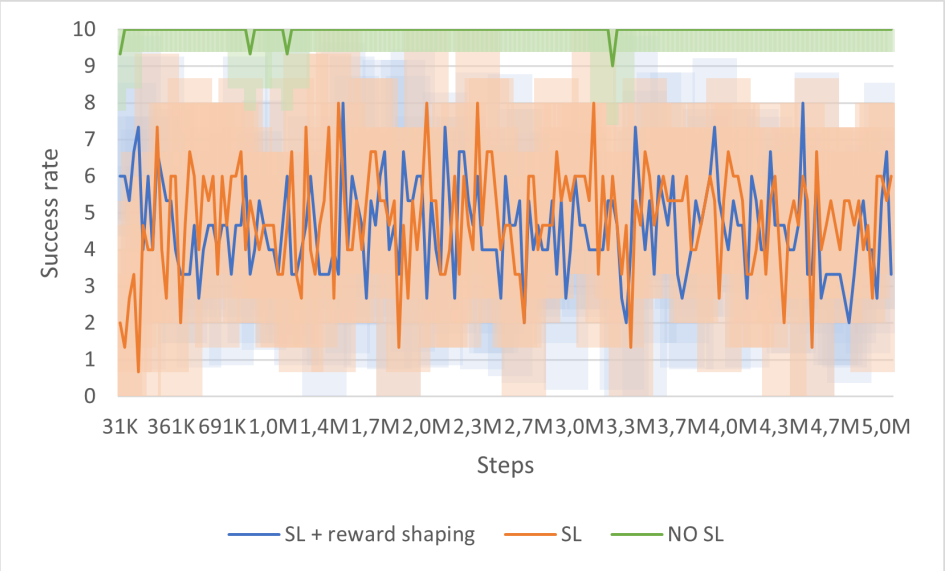


Figure 6.14: Configuration 2 - Average episodic task success rate during evaluation

In Fig. 6.14 it can be seen that unsafe DDPG quickly achieves 100% task success, while original SL and upgraded SL algorithms struggle, not overtaking an average of $55 - 60\%$ of success. Although this could seem an unacceptable loss, it must be noted that, for the configuration of the environment, it is impossible to avoid some decreasing of the performances. As already explained, the introduction of obstacles modifies the structure of the task, causing the lengthen of trajectories to get to the target, that sometimes cannot be reached in the time of an episode. For this reason, a task success of more than 50% is considered satisfactory, especially in light of the improvements obtained on the safety side.

Taking into account the previous considerations, it has been chosen to apply reward shaping for the followings experiments, with the purpose of validate the improvements found and explore more strategies to enhance further the performances.

Transition to sparse reward In order to amplify the effect of the penalty provided by the reward shaping and therefore to encourage the agent in learning a safer policy, some experiments have been performed with sparse reward instead of the dense one. In this configuration, the agent gains a huge reward only when reaching the target ($+1000$), obtaining instead a penalty each times it uses the SL (-0.1). Looking at the episodic violations, reported in Fig. 6.15, it can be seen that the safest configuration is the one with the dense reward.

Looking at the task success, the loss of safety is not justified by an improvement in agent performances: Fig. 6.16 shows clearly that the best policy is the one

obtained with dense reward.

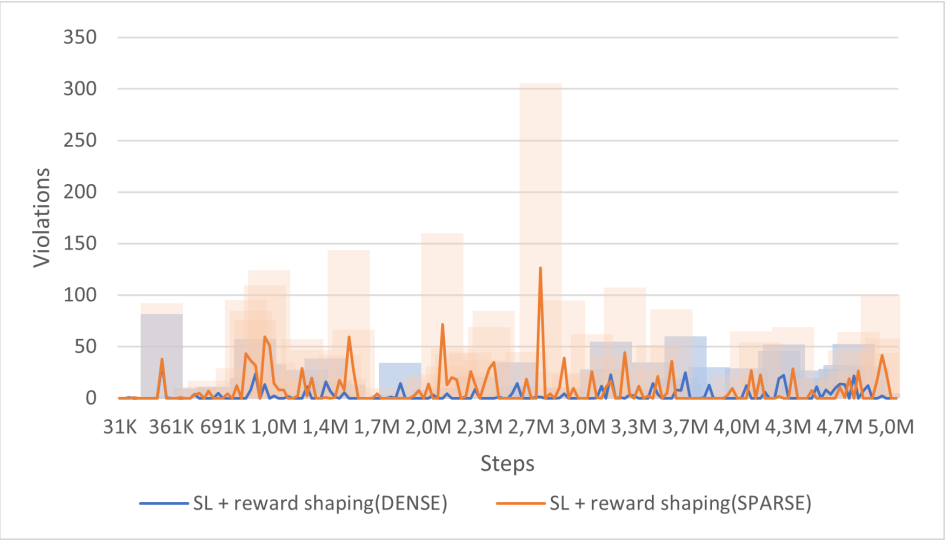


Figure 6.15: Dense vs Sparse reward - Average episodic violations of constraints during evaluation

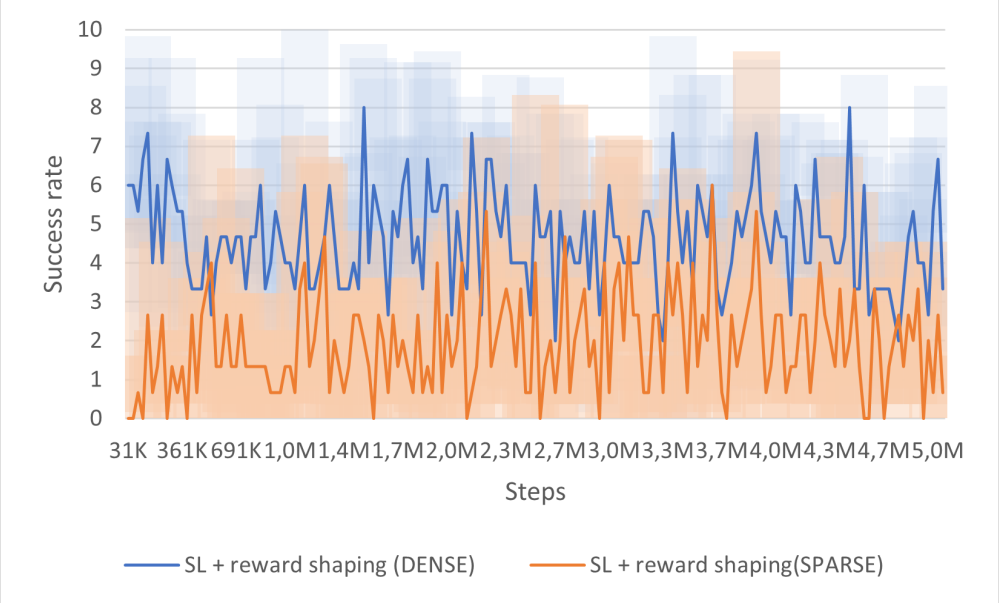


Figure 6.16: Dense vs Sparse reward - Average episodic task success rate during evaluation

Final comparison To assert the reliability of the improvements obtained by using reward shaping with SL, a final comparison have been done.

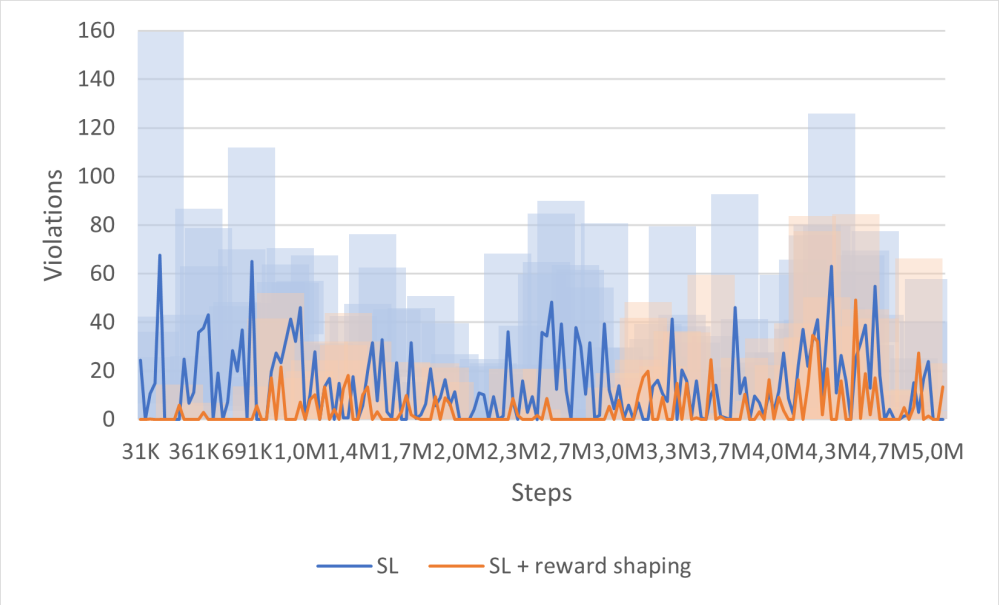


Figure 6.17: Configuration 2 - Average episodic violations of constraint during evaluation

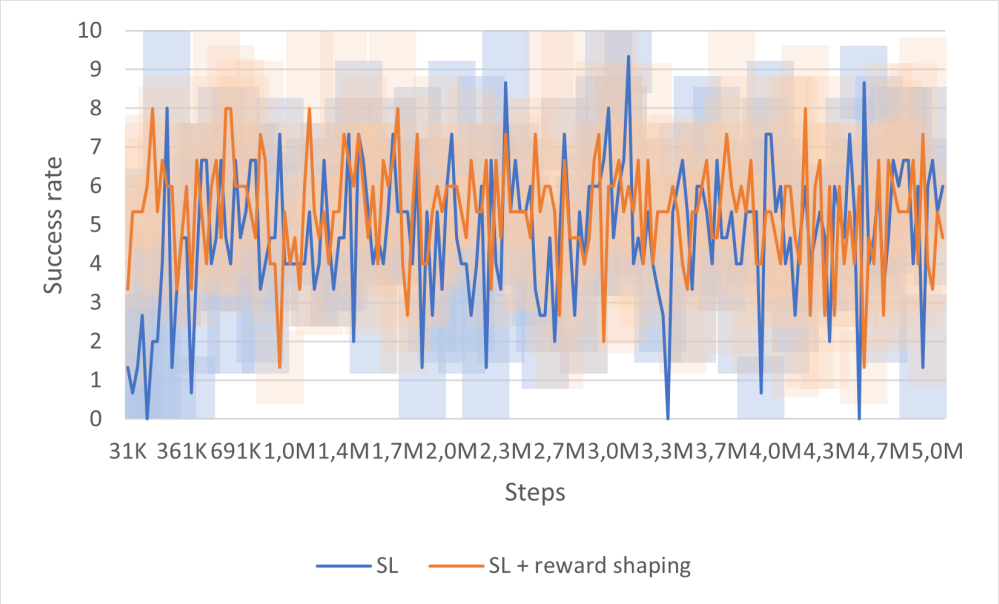


Figure 6.18: Configuration 2 - Average episodic task success rate during evaluation

The model of the best performing SL has been used, without any further training, with and without reward shaping in the optical of having a definitive confrontation between the two alternatives.

In Fig. 6.17 it can be noted the superiority of the algorithm with reward shaping in terms of safety performances, especially in the early stages. Even if an increase of violations can be found later during the training phase, it should be observed in Fig. 6.18 that the learning process could be stopped earlier, since the task is learned long before, already after a million of steps. The same graph also shows that the two algorithms reach a similar satisfactory level in terms of task success.

Chapter 7

Conclusions

The purpose of this work was to find an algorithm that succeeded in:

- performing a safe exploration of the environment during training
- deploying of a safe policy
- learning of a satisfactory policy in terms of task success

To introduce the topic to the reader, a brief presentation of Artificial Intelligence and Reinforcement Learning technique has been provided. After that, the peculiarities of Safe Reinforcement Learning have been treated more in depth, followed by the presentation of the environments and algorithms implemented for the experiments.

The chosen algorithm is called Safety Layer and consists in inserting a correction layer that modifies the dangerous action outputted by the policy into a safe one. To do so, the SL utilizes a neural network to obtain an approximation of the cost function that is linear with respect to the action; the same neural network ($g(s)$) is used in the correction process previously explained.

Although the selected algorithm performs well in environments where the optimal policy is implicitly safe (*CartPole Continuous)*, several issues have been found when developing it in a *non-convex* environment (*Safety Gym*). The main problems are the followings:

- learning of unsafe policy without Safety Layer
- learning of a sub-optimal policy
- scarce dependency of the Safety Layer neural network on the current state
- Safety Layer providing not feasible actions

While for the last two issues no practical solutions have been found, and the second has been classified as an unavoidable consequence of inserting obstacles in the environment, strong improvements have been obtained for the first and most severe problem.

The addition of reward shaping and the retention of the SL as a concrete feature of the policy proved to be a successful strategy in the solution of the issue arose. A significantly safer exploration has been achieved during training, with a massive reduction of violations of constraints. This can be explained considering that the SL protects efficiently the agent in nominal situations, but struggles when it gets stuck on the edges of the obstacles and does not provide any danger knowledge, since it is not explicitly involved into the policy optimization process. Adding a penalty for every time that the SL is called has the effect of discourage the agent in visiting dangerous states (that are the ones in which the SL acts with correction), reducing the time spent close to the borders of the obstacles and therefore leading to a strong improvement of safety. Looking at the same time at the reward obtained, it can be seen that there is no unacceptable loss in performances, since the task success percentage is the same as when not using the reward shaping.

Taking into account the previous considerations, it can be stated that adding reward shaping at the original SL algorithm, a significantly safer exploration of the environment can be achieved, while not having a severe deterioration of agent performances.

7.1 Future Work

Although satisfactory results in terms of safe exploration have been achieved, further improvements may be sought in future work. Some suggestions about strategies to be investigated are the followings:

- use of recurrent neural networks for the policy, in order to provide sort of a long short memory during the training and help the agent to gain more safety knowledge
- to investigate the possibility of a Safety Layer dependent on policy parameters, in the context of finding a $g(s, \theta_\mu)$ network that leads to corrections compatible with the agent (not exceeding the range [-1,1]); an idea could be to have first layers in common between Safety Layer and policy networks
- to improve data collection process during pre-training of the Safety Layer, in order to enhance the efficiency of sample usage in this phase

Acknowledgements

In the course of the development of this Thesis, I had the chance to meet and be constantly in touch with AddFor team. I want to thank Ing. Enrico Busto, who gave me the opportunity to work in his Company and has always been open and available for me, even for the most unexpected requests. I want to thank Sonia Cannavò, that treated me like a friend and reassured me in some particular moments. Most of all, I want to thank Andrea Lonza, that helped me from the very beginning of this work to the last written line, the last experiment performed, the last video-call with my broken webcam and TensorBoard acting crazy. Thank you for having given to me such a great possibility to work on this fascinating field.

I want to thank Professoressa Manuela Battipede, who put me in contact with AddFor, allowing this great experience, and has always guided me in these months of hard work.

I want to thank Nicola, who helped me reach the end of this project in so many ways, also the very pragmatical ones.

I want to say thank you to my family, that during these hard times has always managed to be close to me even when far, even when DPCM and pandemic separated us, even when a video-call was all we had to share.

Last but not least, I need to thank my friends, the ones I know since my first breath and that support me since my first step, my first problem, my first exam, my first thesis, my first work. Thank you for being there since the first "first" of my life.

Finally, to all the people I got to know in the crazy worldwide adventures of the last five years,

grazie, gracias, thank you, danke, ačiū, obrigada.

Appendix A

Pseudo-codes

Algorithm 5 Safety Layer neural network pre-training

Initialize parameters w of $g(s; w)$ (Eq. 4.3), a random policy $\mu(s)$, a replay buffer $\mathcal{D}$
while $episode < N$ **do**
    $s \leftarrow s_0$ ▷ Reset environment state
    $c \leftarrow c_0$ ▷ Reset environment cost
    $t \leftarrow 0$ ▷ Timestep set to 0
    while $t < T_{max}$ **and** not done **do**
        Select action $a_t = \mu(s)$
        Execute action a_t and observe new state s_{t+1} and new cost c_{t+1}
        Store transition (s_t, a_t, c_t, c_{t+1}) in $\mathcal{D}$
        $t \leftarrow t + 1$ ▷ Update timestep
    end while
    $episode \leftarrow episode + 1$ ▷ Update episode
end while
$epoch \leftarrow 0$ ▷ Epoch set to 0
while $epoch < M$ **do**
    Sample *batch* from $\mathcal{D}$
    for *element* in *batch* **do**
        Calculate $c_{pred} = c_t + g(s_t; w)^\top a_t$ (Eq. 4.4)
        Calculate $loss = ||c_{t+1} - c_{pred})||^2$
    end for
    Update parameters w of $g(s; w)$
    $epoch \leftarrow epoch + 1$ ▷ Update epoch
end while
return $g(s; w)$ ▷ Optimized SL approximator is returned

Algorithm 6 Deep Deterministic Policy Gradient with Safety Layer

function SAFETY LAYER PRE-TRAINING
  return $g(s; w)$ ▷ see Alg. 5
end function
Initialize policy parameters θ, Q-function parameters ϕ,empty replay buffer $\mathcal{R}$
$\theta_{targ} \leftarrow \theta, \phi_{targ} \leftarrow \phi$ ▷ Target parameters equal to main ones
while *episode* $< N$ **do**
  $s \leftarrow s_0$ ▷ Reset environment state
  $c \leftarrow c_0$ ▷ Reset environment cost
  Initialize random process $\mathcal{N}$ ▷ For action exploration
  $t \leftarrow 0$ ▷ Timestep set to 0
  while $t < T_{max}$ **and** not done **do**
    Select action $a_t = clip(\mu_\theta(s) + \mathcal{N}_t, a_{low}, a_{high})$
    if $\bar{c}(s) + g(s; w)^\top a_t(s)\, C_i$ **then** ▷ Action a_t is not safe (Eq. 4.5)
      $\lambda^*_{i^*} = \left[\frac{g(s;w_{i^*})^\top \mu_\theta(s) + \bar{c}_i(s) - C_i}{g(s;w_{i^*})^\top (s;w_{i^*})}\right]^+$ ▷ Eq: 4.6b
      $i^* = \underset{i}{\text{argmax}}\ \lambda^*_{i^*}$ ▷ Eq: 4.6c
      $a_t = \mu_\theta(s) - \lambda^*_{i^*} g(s; w_{i^*})$ ▷ Correct the action (Eq: 4.6a)
    end if
    Execute action a_t and observe s_{t+1}, c_{t+1} and d done signal
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
    $s_t \leftarrow s_{t+1}$
    $c_t \leftarrow c_{t+1}$
    if time to update **then**
      for m=1,M **do**
        Sample a random minibatch from $\mathcal{R}$
        $y_t = r_t + \gamma(1-d) Q_{\phi_{targ}}(s_{t+1}, \mu_{\theta_{targ}}(s_{t+1}))$
        ▷ Target network used for target estimation
        Update Q_ϕ using ▷ Stochastic Gradient Descent
        $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_\phi(s,a) - y_t)^2$
        Update μ_θ using ▷ Stochastic Gradient Descent
        $\nabla_\theta \frac{1}{|B|} \sum_{(s) \in B} Q_\phi(s, \mu_\theta(s))$
        $\phi_{targ} \leftarrow \rho \phi_{targ} + (1-\rho)\phi$
        $\theta_{targ} \leftarrow \rho \theta_{targ} + (1-\rho)\theta$
      end for
    end if ▷ Update target networks
    $t \leftarrow t + 1$
  end while
  episode $\leftarrow$ *episode* $+ 1$
end while

Algorithm 7 DDPG with Safety Layer and reward shaping

function SAFETY LAYER PRE-TRAINING
  return $g(s; w)$ ▷ see Alg. 5
end function
Initialize policy parameters θ, Q-function parameters ϕ,empty replay buffer $\mathcal{R}$
$\theta_{targ} \leftarrow \theta, \phi_{targ} \leftarrow \phi$ ▷ Target parameters equal to main ones
while *episode* $< N$ **do**
  $s, c \leftarrow s_0, c_0$ ▷ Reset environment state and cost
  Initialize random process $\mathcal{N}$ ▷ For action exploration
  $t \leftarrow 0$ ▷ Timestep set to 0
  while $t < T_{max}$ **and** not done **do**
    Select action $a_t = clip(\mu_\theta(s) + \mathcal{N}_t, a_{low}, a_{high})$
    if $\bar{c}(s) + g(s; w)^\top a_t(s)\, C_i$ **then** ▷ Action a_t is not safe (Eq. 4.5)
      $\lambda^*_{i^*} = \left[\frac{g(s;w_{i^*})^\top \mu_\theta(s) + \bar{c}_i(s) - C_i}{g(s;w_{i^*})^\top g(s;w_{i^*})}\right]^+$ ▷ Eq: 4.6b
      $i^* = \underset{i}{\text{argmax}}\ \lambda^*_{i^*}$ ▷ Eq: 4.6c
      $a_t = \mu_\theta(s) - \lambda^*_{i^*} g(s; w_{i^*})$ ▷ Correct the action (Eq: 4.6a)
    end if
    Execute action a_t and observe s_{t+1}, c_{t+1} and d done signal
    if $\lambda^*_{i^*} > 0$ **then**
      $r_t \leftarrow r_t - penalty$ ▷ Reward shaping
    end if
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
    $s_t, c_t \leftarrow s_{t+1}, c_{t+1}$
    if time to update **then**
      for m=1,M **do**
        Sample a random minibatch from $\mathcal{R}$
        $y_t = r_t + \gamma(1-d) Q_{\phi_{targ}}(s_{t+1}, \mu_{\theta_{targ}}(s_{t+1}))$
        Update Q_ϕ using ▷ Stochastic Gradient Descent
        $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Q_\phi(s,a) - y_t)^2$
        Update μ_θ using ▷ Stochastic Gradient Descent
        $\nabla_\theta \frac{1}{|B|} \sum_{(s) \in B} Q_\phi(s, \mu_\theta(s))$
        $\phi_{targ} \leftarrow \rho \phi_{targ} + (1-\rho)\phi$
        $\theta_{targ} \leftarrow \rho \theta_{targ} + (1-\rho)\theta$
      end for
    end if ▷ Update target networks
    $t \leftarrow t + 1$
  end while
  episode $\leftarrow$ *episode* $+ 1$
end while

Bibliography

[1] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957 (cit. on p. 2).

[2] Massimiliano Patacchiola. *Dissecting Reinforcement Learning*. URL: https://mpatacchiola.github.io/blog/2016/12/09/dissecting-reinforcement-learning.html. 2016 (cit. on p. 2).

[3] Wikipedia contributors. *Decision-making — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Decision-making&oldid=1012387521. [Online; accessed 17-March-2021]. 2021 (cit. on p. 3).

[4] David Silver. *Lectures on Reinforcement Learning*. URL: https://www.davidsilver.uk/teaching/. 2015 (cit. on pp. 9, 11, 15, 16).

[5] Stuart Russell and Peter Norvig. «Artificial intelligence: a modern approach». In: (2002) (cit. on p. 15).

[6] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998 (cit. on p. 17).

[7] Mayank Agarwal. *Back Propagation in Convolutional Neural Networks — Intuition and Code*. URL: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199. 2017 (cit. on p. 21).

[8] Frederik Kratzert. *Understanding the backward pass through Batch Normalization Layer*. URL: https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html. 2016 (cit. on p. 21).

[9] George E Uhlenbeck and Leonard S Ornstein. «On the theory of the Brownian motion». In: *Physical review* 36.5 (1930), p. 823 (cit. on p. 27).

[10] Javier Garcıa and Fernando Fernández. «A comprehensive survey on safe reinforcement learning». In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1437–1480 (cit. on pp. 30, 32).

[11] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. «Constrained policy optimization». In: *International Conference on Machine Learning*. PMLR. 2017, pp. 22–31 (cit. on p. 32).

[12] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. «Safe exploration in continuous action spaces». In: *arXiv preprint arXiv:1801.08757* (2018) (cit. on pp. 33, 35–40, 52).

[13] Brandon Amos and J Zico Kolter. «Optnet: Differentiable optimization as a layer in neural networks». In: *International Conference on Machine Learning*. PMLR. 2017, pp. 136–145 (cit. on p. 36).

[14] Alex Ray, Joshua Achiam, and Dario Amodei. «Benchmarking Safe Exploration in Deep Reinforcement Learning». In: (2019) (cit. on p. 39).

[15] Andrew G Barto, Richard S Sutton, and Charles W Anderson. «Neuronlike adaptive elements that can solve difficult learning control problems». In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846 (cit. on pp. 39, 40).

[16] Ian Danforth. *CartPole Continuous code*. URL: `https://gist.github.com/iandanforth/e3ffb67cf3623153e968f2afdfb01dc8`. 2018 (cit. on p. 40).