POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Applying rule-based controllers and reinforcement learning to control a general purpose robot: the Air Hockey challenge case

Tesi di Laurea Magistrale in
Computer Science & Engineering - Ingegneria Informatica

Author: **Francesco Minnucci**

Student ID: 996059
Advisor: Prof. Marcello Restelli
Co-advisors: Amarildo Likmeta, Alessandro Montenegro
Academic Year: 2022-23

Abstract

Closing the reality gap generated by hardware limitations, physical constraints and uncertainty of dynamics and sensors, represents a challenging task in modern days highly dynamic robotic environments. Air hockey is an example of this type of environments, since it is a game composed of different tasks requiring fast planning and immediate reaction to environmental changes. It this thesis, it will be shown how to design a policy for a general purpose robot, in order to make it play a complete air hockey game. A combination of a rule-based approach and reinforcement learning will be used to train the developed agent in executing specific tasks of air hockey, such as: hitting, defending and preparing. Finally, all the trained tasks will be combined in a hierarchical agent, deployed on a simulated robot to let it play a complete game against an opponent, while trying to avoid constraints violations.

Keywords: Reinforcement Learning; Policy Gradient Methods; Robotics; Air Hockey; Rule-based policies; Policy Optimization; Explainable Artificial Intelligence.

Abstract in lingua italiana

Colmare il cosiddetto *reality gap*, generato da limitazioni di hardware, limiti fisici ed incertezze dovute sia alla dinamica dell'ambiente che al rumore proveniente dai sensori, rappresenta una grande sfida per i manipolatori robotici che operano in contesti ad alta dinamicità. L'Air Hockey rappresenta un esempio di questo tipo di ambienti, poichè si tratta di un gioco composto da task diversi che richiedono un'elevata capacità di pianificazione e un'immediata reazione ai rapidi cambiamenti ambientali. In questa tesi verrà mostrato come modellare una politica per un robot generico, non ideato per risolvere un task specifico; in particolare verrà utilizzato un manipolatore a 7 gradi libertà. L'obiettivo finale sarà quello di sviluppare un agente in grado di partecipare ad una partita completa di Air Hockey. Una combinazioni di politiche a regole e reinforcement learning verrà utilizzata per addestrare l'agente sviluppato, in modo da permettergli di eseguire compiti specifici tipici di una partita di Air Hockey, come: colpire il disco, difendere la porta da un attacco, riposizionare il disco. Successivamente, tutte le singole funzioni allenate saranno combinate all'interno di un agente gerarchico, per poi essere integrate in un ambiente simulato al fine di testare il robot, in modo da permettergli di giocare una partita completa contro un agente di default, il tutto tentando di violare il minor numero di vincoli strutturali e di gioco possibili.

Parole chiave: Reinforcement Learning; Policy Gradient Methods; Robotics; Air Hockey; Rule-based policies; Policy Optimization; Explainable Artificial Intelligence.

Contents

Abstract i

Abstract in lingua italiana iii

Contents v

1 Introduction 1
1.1 Motivation . . . 1
1.2 Goal . . . 2
1.3 Thesis Structure . . . 3

2 Preliminary backgorund 5
2.1 Markov Decision Processes . . . 5
2.2 Reinforcement Learning . . . 6
2.2.1 Policy Optimization . . . 9
2.2.2 Parametric policies . . . 10
2.2.3 Policy Gradient with Parameter-based Exploration (PGPE): . . . 13
2.2.4 Actor Critic approaches . . . 16

3 Robot Air Hockey Challenge 19
3.1 What is Air Hockey . . . 19
3.2 Challenge Motivation . . . 20
3.3 Challenge organization . . . 20
3.3.1 Warm-Up . . . 21
3.3.2 Qualifying . . . 22
3.3.3 Tournament . . . 23
3.4 Framework . . . 24
3.4.1 Environments . . . 25
3.5 Agents . . . 27
3.5.1 Planar Robot - 3 Degrees of Freedom . . . 27

3.5.2 KUKA iiwa14 LBR Robot . . . 28
3.5.3 Evaluation metrics . . . 29
3.6 Constraints . . . 29

4 Methodology 33
4.1 Air Hockey as an MDP . . . 33
4.1.1 State Space . . . 33
4.1.2 Action space . . . 34
4.1.3 Reward function . . . 34
4.2 Rule-Based Agent . . . 35
4.2.1 Hit task . . . 37
4.2.2 Defend task . . . 41
4.2.3 Prepare task . . . 42
4.2.4 Default Position task . . . 44
4.2.5 Noise filtering . . . 44
4.3 Hierarchical Agent . . . 45
4.3.1 Switcher . . . 45
4.3.2 Finite State Machine (FSM) . . . 46

5 Experimental results 49
5.1 Initial experiments . . . 49
5.2 Gradient analysis . . . 51
5.3 Training results . . . 53
5.4 Challenge outcome . . . 54

6 Related Works 59

7 Conclusions 63
7.1 Future Works . . . 63

Bibliography 65

List of Figures 69

List of Tables 71

List of Acronyms 73

1 | Introduction

1.1. Motivation

Closing the reality gap represents one of the most challenging tasks in the field of Robot Learning, in order to reach a real embodied intelligence. To deploy an effective learning algorithm on a real world operating robot, careful considerations of practical factors must be kept into account: sensors noise, observation delays, actuator limitations and physical feasbility. The factors just cited are only some of the multiple issues encountered in the development process. To better understand the described limitations, a more restricted dynamic enviroment can be analyzed. In this thesis, such an environment is AirHockey.

AirHockey is a game characterized by a 2D constrained environment, the game field, and by high uncertainty. The latter, in particular, originates from multiple sources: the air flowing through small holes in the filed, producing an uneven airflow that causes fast and uncertain movements; the collision between the puck and the mallet, or the table's borders, results in higly uncertaint trajectories. AirHockey game itself is further characterized by a set of rules that the agent must follow to play a fair match.

While coping with the aforementioned challenges, a task-specific robot represents a possibility, however employing *general-purpose* robots to face dynamic tasks is desirable if the task requirements are blurry.

In this work, in order to allow a general purpose robot to perform a particular task, in this case playing the AirHockey game, a specific controller was developed. The robot controller was trained by means of a combination of Reinforcement Learning [RL, 21] and *parametric rule-based policies*. In RL an agent learns a correspondence, commonly called *policy*, between observations and actions. The learning process is carried out by interacting with the environment. The learning path is steered by a *reward function*, which serves as an intuitive measure of the agent's local performances. Due to the automatic nature of the policy-determination process, an RL controller tends to be robust since it is able to generalize to unpredicted situations. The main issue of using an RL controller resides in the possible lack of results interpretability, that is heavily influenced by the

model utilized to link observations to actions. A *parametric rule-based policy* is, as stated by Likmeta et al. in [10], “a rule-based controller in which the rules are defined in terms of a set of parameters, whose values are not manually set, but learned by interacting with the environment using an RL algorithm”.

The described RL framework materializes in a parameter-based RL method called Policy Gradients with Parameter-Based Exploration for Control [PGPE, 20], that was employed in order to learn the aforementioned parameters of the rule-based policy.

Participation to the Air Hockey Challenge

As stated before, AirHockey represents a valid environment to test the capabilities of a general purpose robot subject to constraints. This thesis is based on a competion, *Robot Air Hockey Challenge*, organized by the *“Technische Universitat Darmstad (TUD)”*. The challenge is structured as follows.

The challenge consists in three simulated stages: Warm Up, Qualifying and Tournament. In each of this stages different tasks are required for robot air hockey. The developed agents should be able, not only to perform a particular task, but also be robust and capable of adapting to changes. The agents are evaluated in a simulated environment to mimic the simulation-to-real gap. In the first two phases the evaluation is based on the success rate of the specific task, either Hit, Defend or Prepare, and on the violated constraints. At the end of the qualifying stage, only the first 16 teams, based on the described evaluation, are allowed to access the tournament. In the tournament phase the agent must be able to play a complete game and, as the name suggests, mathces between participants are organized, to produce a leaderboard.

A more detailed description of the challenge can be found in chapter 3, the following is the official website of the challenge: Air Hockey Challenge website.

1.2. Goal

The goal of this work is to develop an agent capable of controlling a general purpose robot, in order to play an Air Hockey match. The agent should be capable of exploiting the whole potential of the robot, e.g. by shooting the puck as fast as possible, without violating contstraints relative to robot and game requirements; while dealing with noisy observations.

Thesis outcome

The devolped agent, succesfully passed the qualifyng stage cut-off accessing the tournament. After two rounds of tournament matches the agent ranked fifth out of fourty-six teams enrolled in the challenge.

1.3. Thesis Structure

The work will proceed in Chapter 2, revising all the background essentials necessary to a better understanding of the thesis. Chapter 3 will describe what is AirHockey and the organization of the challenge in its phases: warm-up, qualifying and tournament. Moreover a description of the framework provided by the organizers will be given, followed by a further explanation on the constraints to which the developed agent had to adhere. In Chapter 4, the thesis will delve into the methods used to solve the challenge, describing the formalization of the problem as a Markov Decision Process [MDP, 16]. The details of the developed Rule Based Agent will also be provided. The experimental results will be described in Chapter 5 reporting the whole training process which led to the final agent. Chapter 6 will contain an overview of the related works, paying special attention to the ones coping with the constraints. Finally, in Chapter 7, conclusions of the work will be drawn, and some possible future development cues described.

2 | Preliminary backgorund

This chapter will describe the background knowledge necessary to fully understand the work of this thesis.

It will start by describing a foundamental framework exploited during the challenge, the concpet of Markov Decision Process [MDP, 16]. A description of Reinforcement Learning will follow, contextualizing it in the broader setting of *learning by interaction*, a concept that forms the foundamental basis of almost all theories regarding learning and intelligence. In this setting, in particular, Reinforcement Learning aims at learning to control the state of the environment.

The chapter will proceed with an overview over the concept of policy optimization and will describe various types of policies, together with an algorithm focused on parameter-based exploration. A shallow description of Actor Critic approaches will also be provided at the end of the chapter.

2.1. Markov Decision Processes

While dealing with Sequential Decision Making problems, a Markov Decision Process [MDP, 16] can be used to formalize the idea of environment, where the agent operates. This environment can be either stationary or not. In the first case its dynamics remain the same over time while in the second one, they might change. A Markov Decision Process can take place within a finite or infinite time horizon. In particular, the first assumes that the decision process takes place over a finite, predetermined, number of steps. In what follows, an infinite time horizon with discounted reward is considered.

An infinite time horizon, discounted, MDP is defined as a tuple $M := \langle \mathcal{S}, \mathcal{A}, p, r, \gamma, \mu_0 \rangle$ where:

- $\mathcal{S}$ is the continuous state space;
- $\mathcal{A}$ is the continuous action space;
- $p : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the state transition model. $p(s'|s, a)$ specifies the probability

that the next state is s', assuming that action a was taken in state s;

- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. $r(s, a)$ expresses the istantaneous reward if the agent is in state s and selects action a;
- $\gamma \in [0, 1]$ is the discount factor, it specifies how important future rewards will be while learning the policy: if close to 0 it leads to myopic evaluations, if close to 1 leads to a far-sighted evaluation. It can be interpreted as the probability that the interaction will last one step more, conversely, $1 - \gamma$ is the probability that such interaction will stop in the next step;
- $\mu_0 \in \Delta(\mathcal{S})$ represents the initial probability distribution of states. $\mu_0(\cdot)$ gives the probability that the MDP starts with state s.

At time t, a state $s_t \in \mathcal{S}$ can be described as a function of the history as follows:

$$s_t = f(s_1, a_1, r_1, \ldots, s_{t-1}, a_{t-1}, r_{t-1}).$$

In an MDP, the Markovian property holds. A stochastic process X_t is said to be Markovian if and only if:

$$\mathbb{P}(X_{t+1} = j | X_t = i, X_{t-1} = k_{t-1}, \ldots, X_1 = k_1, X_0 = k_0) = \mathbb{P}(X_{t+1} = j | X_t = i),$$

therefore, if the probability is stationary (time invariant), it is possible to write:

$$p_{ij} = \mathbb{P}(X_{t+1} = j | X_t = i) = \mathbb{P}(X_1 = j | X_0 = i).$$

The above property states that the future is independent from the past given the present, i.e., the current state itself is enough to capture all the information coming from the history of interactions. As a simple example, it is possible to think of a Rubik's cube: knowing what have been the previous actions is not necessary to solve it, only the current state is needed.

2.2. Reinforcement Learning

Interaction protocol In the following paragraph it is assumed to deal with a stationary environment. This means that its dynamics remain the same over time and each action has a fixed expected value, while each reward observation is a noisy realization of it.

Each learning process, either human or artificial, arises from the interaction of two elements:

- The **Agent**: it has the ability of performing actions on the environment and observes the outcomes of its actions;
- The **Environment**: it is evertyhing outside the agent; it is characterized by a state and is solicited by the agent with actions, to which it emits an observation and a reward.

The aforementioned reward, represents a notion of utility, that the agent will try to maximize over time while interacting with the environment.

From now on, a *fully observable MDP* will be considered, therefore the observation will coincide, at each step, with the real state.

Reinforcement Learning [RL, 21] is a sub-field of Machine Learning [ML, 1] that copes exactly with the described situation. The agent's goal is to identify a mapping, called *policy*, between observations and actions. Further details about the policy will be provided later in the chapter.

A general RL framework can be seen in Figure 2.1.

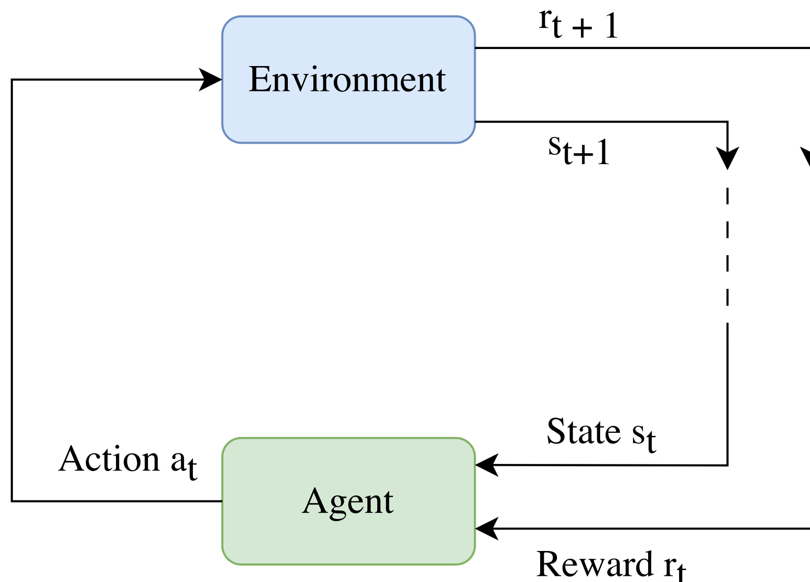


Figure 2.1: The ongoing interaction between the agent and the enviroment comprises the agent's selection of an action and the subsequent response of the environment, providing a new observation (state) and a reward.

Each action might influence not only the immediate state, but also the next ones, therefore also the long term rewards. Since the agent aims at maximizing the total utility in a state, called *cumulative reward*, it is necessary to express the desired goal in terms of the reward function. This formalization of the goal is described by the so called *Sutton hypothesis* [21]:

> *All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

As can be seen in Section 2.1, using this scalar reward signal is a keystone in the definition of an MDP. While designing the reward function, particular attention should be kept on rewarding the sub-goals, since the agent might learn to achive them, ignoring the main goal. Moreover, the defined reward function will steer the learning process, specifying how well the agent is performing locally.

RL algorithms taxonomy According to Sutton et al. ([21]), Reinforcement Learning algorithms can be *model-based*, if they use the collected experience in order to build an approximate model of the environment, from which is possible to compute the optimal policy, or *model-free* if they directly compute the optimal policy from the samples they collected. While dealing with *model-free* algorithms there are two main branches of learning which can be distinguished:

- **On-policy learning**: learn about policy π from experience sampled from π. It learns action values for a near-optimal policy that still explores.
- **Off-policy learning**: learn about policy π from experience sampled from $\bar{\pi}$. The *target policy* is different form the *behavioural policy*, the one used to interact with the environment. This type of learning aims at reusing past experience.

A further classification divides RL algorithms into:

- *Value-based*: focused on estimating the optimal value function, from which the optimal policy is then derived;
- *Policy-based*: they directly search in the policies space for the one that maximizes the expected return;
- *Actor-critic*: an hybrid and more general approach that combines evaluation functionalities of value-based and the search strategies of policy-based approaches. The policy is usually referred to as the actor while the value function as the critic.

Policy

A policy π, is the objective of the learning phase and defines the behaviour of the agent, given a state in which it is.

Formally, it defines a distribution over the actions $a \in \mathcal{A}$, given the state $s \in \mathcal{S}$:

$$\pi : \mathcal{S} \to \Delta(\mathcal{A}).$$

For each state s, given an action a, the policy expresses the probability that the agent will select action a while in state s. A specific case is represented by the so called *determinstic policy*: in each state $s \in \mathcal{S}$ it will always suggest the same action. More formally:

$$\forall s \in \mathcal{S} \; \exists! \bar{a} \in \mathcal{A} \text{ s.t. } \pi(\bar{a}|s) = 1,$$

and, $\forall a \neq \bar{a}$ it holds that:

$$\pi(a|s) = 0.$$

Objective

The main goal of the agent is to learn the *optimal policy* π^*, i.e., the policy that maximizes the *expected return*, which is the expected discounted sum of future rewards:

$$J(\pi) := \mathbb{E}^{\pi} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) \right], \tag{2.1}$$

where, $\mathbb{E}^{\pi}[\cdot]$ is computed considering the randomness of both the environment and the policy.

Here, T represents the horizon of the learning problem, and can possibly be infinite. In the latter case, in particular, few words should be spent on the discount factor γ. Since its value ranges in [0,1], if $T = +\infty$ then γ cannot be 1, in this case it is mandatory to consider the rewards in a far future less important.

2.2.1. Policy Optimization

As stated in Section 2.2, one of the various types of RL algorithms is the so called *policy-based*, which directly operates in the policies space. A *Policy Optimization* (PO)

algorithm, directly searches over the policies space, ignoring the definition of the target policy as a function of another learned object, like a value function.

Sometimes the term *Policy Search* is used as a synonym of Policy Optimization.

While comparing policy optimization and value-based approaches various advantages of the PO arise. In particular, as reported by Papini in [14]: convergence guarantees, continuous actions, robustness to noise, partial observability, use of prior knowledge, safety and explainability. Explainability represents an important property for this work, since the developed agent was trained by means of a combination of RL and *parametric rule-based policy*. The policy can be designed with a high level of human engineering, therefore it will be highly explainable, and PO will be responsible only for discovering the best parameters; after the optimization the final controller will be the same, except with different parameters.

One of the main drawbacks of policy optimization arises from the policy design itself. The explainability advantage comes at the cost of possibly restricting the set of feasible policies, since they will be tight to the designed one, therefore it is possible that the unexpectedly good polices are ruled out. This drawback makes PO generally more suited for the fine-tuning of already existing controllers, rather than developing new ones from scratch.

2.2.2. Parametric policies

A policy class of particular interest is the one containing *parametric policies*. Letting $\Theta \subseteq \mathbb{R}^n$ be a parameter space, for some $n \in \mathbb{N}$; a policy class is parametrized by Θ if it belongs to $\Pi_\Theta = \{\pi_\theta \in \Delta_{\mathcal{A}}^{\mathcal{S}} | \theta \in \Theta\}$. The elements of Θ are real-valued n-dimensional vectors, called *policy parameters*, the elements of Π_Θ are the parametric policies.

Rule-Based Policies

As suggested by Likmeta et al. in [10], a deterministic parametric rule-based policy can be denoted as $\pi_\theta : \mathcal{S} \to \mathcal{A}$, a function that takes a state s as input and produces, as output, a control action $a = \pi_\theta(s)$ parametrized by θ, a d-dimensional vector, such that $\theta \in \Theta \subseteq \mathbb{R}^d$, where Θ is called *parameter space*.

While dealing with parametric policies, the main goal becomes to find the best parameter in Θ, therefore maximizing the expected return (consider for conciseness $J(\theta) = J^{\pi_\theta}$):

$$\max_{\theta \in \Theta} J(\theta) = \mathop{\mathbb{E}}_{\substack{s_0 \sim \mu \\ s_{t+1} \sim p(\cdot|s_t, \pi_\theta(s_t))}} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, \pi_\theta(s_t), s_{t+1}) \right], \tag{2.2}$$

where T is the horizon of the task.

The agent will interact with the environment while executing policy π_θ, collecting N independent episodes $\{\tau_i\}_{i=1}^N$ and replacing the expected reward with a corresponding sample mean, providing the estimator of the expected return:

$$\widehat{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \gamma^t r(s_{i,t}, \pi_\theta(s_{i,t}), s_{i,t+1}). \tag{2.3}$$

It is essential to test multiple values of θ to determine the optimal parameters, each of which needs an evaluation of the objective, therefore an interaction with the environment, possibly resulting in suboptimal outcomes due to a varying parameterizations.

The optimization of Equation (2.3) can be carried out using two different classes of policy based methods: *action-based* policy optimization, also called *policy gradient methods*, and *parameter-based* policy optimization. The first method, updates the parameters according to the improving direction of the gradient ([15, 22]):

$$\theta \leftarrow \theta + \alpha \widehat{\nabla}_\theta J(\theta), \tag{2.4}$$

where $\alpha > 0$ is the learning rate and $\widehat{\nabla}_\theta J(\theta)$ is an estimation of the policy gradient. This methods require that the policy π_θ is not only differentiable w.r.t. the parameters θ, but also stochastic, so that it can guarantee a sufficient degree of exploration.

A deterministic rule-based policy cannot adhere to this requirements: it can be seen as a *parametrized decision tree*, therefore it is not not, in general, differentiable and surely not stochastic. Furthermore, stochastic policies prevent enforcing the traceability of the decision making process.

Another significant problem with policy gradient methods, arises from the algorithms employed: they tend to show a slow convergence, due to the high variance in their gradient estimates. The main cause of this problem relies in the repeated sampling from a probabilistic policy which translates into an injection of noise in the gradient estimates after each step. Moreover, the variance increases linearly with the length of the history, since each new state depends on the whole sequence of previous samples.

Due to all the aforementioned drawbacks of action-based policy optimization, as an alternative, it is possible to resort to parameter-based policy optimization methods, further described in Section 2.2.3.

In what follows, to simplify some formulas and avoid numerical instability, the *log-trick* ($\nabla f = f \nabla \log f$) will be used. While working with small probabilities or likelihoods, this mathematical trick consists in working with the logarithms of these quantities, rather than with the probabilities themselves. This necessity arises form the fact that probabilities can be small numbers, leading to precision issues in floating-point arithmetic. By working with the logarithms, products of probabilities are turned into sums of logarithms, leading to more manageable and stable values.

Gaussian Policies

While dealing with continuous action spaces, like the one of the AirHockey challenge, a common choice for the parametric policy is the *Gaussian* one. Considering scalar actions, therefore $\mathcal{A} = \mathbb{R}$, a Gaussian policy is defined as:

$$\pi_\theta(a|s) = \frac{1}{\sqrt{2\pi}\sigma_\theta(s)} \exp\left(-\frac{(a - \mu_\theta(s))^2}{2\sigma_\theta^2(s)}\right), \tag{2.5}$$

here π, with no subscripts, represents the mathematical constant; $\sigma : \Theta \times \mathcal{S} \to (0, \infty)$ is the *standard deviation* function of the policy (σ^2 is the *variance*), while $\mu : \Theta \times \mathcal{S} \to \mathbb{R}$ is the *mean* function of the policy. In this type of policies the stochasticity amount is regulated only by σ_θ.

To sample from a Gaussian policy one can proceed as follow:

$$a = \mu_\theta(s) + \sigma_\theta(s)\eta, \tag{2.6}$$

where $\eta \sim \mathcal{N}(0, 1)$ is a standard normal random variable.

As reported by Papini [14], for a Gaussian Policy, the *score*, i.e., the gradient of log-likelihood with respect to the policy parameters, can be computed as:

$$\nabla_\theta \log \pi_\theta(a|s) = -\frac{\nabla_\theta \sigma_\theta(s)}{\sigma_\theta(s)} + \frac{a - \mu_\theta(s)}{\sigma_\theta^2(s)} \left(\nabla_\theta \mu_\theta(s) + \frac{a - \mu_\theta(s)}{\sigma_\theta(s)} \nabla_\theta \sigma_\theta(s)\right). \tag{2.7}$$

2.2.3. Policy Gradient with Parameter-based Exploration (PGPE):

As anticipated in Section 2.2.2, *parameter-based* policy optimization, represents an alternative for the optimization of Equation (2.3). This type of approach, also known as *policy gradient with parameter-based exploration* [PGPE, 20], moves the exploration problem to a higher level, the parameters one, allowing the use of non-differentiable and deterministic policies.

A *hyperpolicy* ν_ρ is defined, depending on a parameter vector $\rho \in \mathcal{R} \subseteq \mathbb{R}^p$; the parameters θ will be sampled from this hyperpolicy. It is necessary that ν_ρ is stochastic and differentiable w.r.t. ρ. The exploration goes on by testing parameters sampled from ν_ρ, therefore the policy π_θ can also be deterministic and non-differentiable. As shown by Likmeta et al. in [10], it is convenient to redefine the expected return as a function of ρ:

$$
\begin{aligned}
J(\rho) &= \mathop{\mathbb{E}}_{\theta\sim\nu_\rho}\left[J(\theta)\right] \\
&= \mathop{\mathbb{E}}_{\theta\sim\nu_\rho}\underbrace{\left[\mathop{\mathbb{E}}_{\substack{s_0\sim\mu \\ s_{t+1}\sim p(\cdot|s_t,\pi_\theta(s_t))}}\left[\sum_{t=0}^{T-1}\gamma^t r(s_t,\pi_\theta(s_t),s_{t+1})\right]\right]}_{J(\theta)}.
\end{aligned}
$$

The following algorithm, from [10], describes the learning process:

Algorithm 2.1 PGPE

Input: N number of sampled policy parameters
M number of episodes per policy parameter
Ite number of iterations
$(\alpha^h)_{Ite-1}$ learning rate schedule

Initialize the hyperpolicy parameters ρ^0 arbitrarly
for $h = 0, 1, \dots, Ite-1$ **do**
  Sample N policy parameteers $\{\theta_i^h\}_{i=1}^N$ independently from v_{ρ^h}
  Collect M trajectories $\{\tau_{i,j}^h\}_{j=1}^M$ independently for each $\{\pi_{\theta_i^h}\}_{i=1}^N$
  Update the hyperpolicy parameters $\rho^{h+1} = \rho^h + \alpha^h\widehat{\nabla}_\rho J(\rho^h)$
end for

N parameters $\{\theta_i\}_{i=1}^N$ are sampled independently from the hyperpolicy ν_ρ. For each of them, the rule-based policy π_{θ_i} is run, collecting M independent episodes $\{\tau_{ij}\}_{j=1}^M$, (Figure 2.2). After these steps, all the NM episodes are used to estimate the objective:

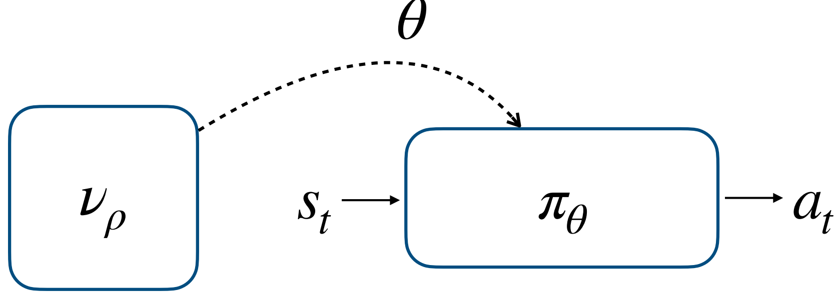


Figure 2.2: Graphical representation of parameter-based methods.

$$\widehat{J}(\rho) = \frac{1}{N}\sum_{i=1}^{N}\widehat{J}(\theta_i) \tag{2.8}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\underbrace{\frac{1}{M}\sum_{i=1}^{M}\sum_{t=0}^{T-1}\gamma^t r(s_{ij,t},\pi_\theta(s_{ij,t}),s_{ij,t+1})}_{\widehat{J}(\theta_i)}. \tag{2.9}$$

Exploting the stochasticity and differentiability of the hyperpolicy, it is possible to compute the gradient, taking advantage of the log-trick ($\nabla f = f\nabla \log f$) [19]:

$$\nabla_\rho J(\rho) = \mathbb{E}\left[\nabla_\rho \log \nu_\rho(\theta) J(\theta)\right].$$

After each iteration, the hyperpolicy is therefore updated with a single step of gradient ascent:

$$\rho \leftarrow \rho + \alpha\widehat{\nabla}_\rho J(\rho),$$

where,

$$\widehat{\nabla}_\rho J(\rho) = \frac{1}{N}\sum_{i=1}^{N}\nabla_\rho \log \nu_\rho(\theta_i)\widehat{J}(\theta_i).$$

Where $\alpha > 0$ is the learning rate, $\widehat{\nabla}_\rho J(\rho)$ the estimator of the gradient $\nabla_\rho J(\rho)$ retrieved with the collected episodes $\{\{\tau_{ij}\}_{j=1}^{M}\}_{i=1}^{N}$.

It is common to assume a gaussian hyperpolicy, where ρ consists of a set of means and standard deviations $\{\mu_i, \sigma_i\}$ that determines an independent Gaussian distribution for each parameter θ_i in θ. This Gaussian hyperpolicy would present a diagonal covariance matrix $\nu_{\rho,\theta} = \mathcal{N}(\mu, diag(\sigma))$ having $\rho = (\mu, \sigma)$ as parameters. In such case, the expression

of the gradient will become:

$$\nabla_{\mu_i} \log v_{\mu,\sigma}(\theta) = \frac{(\theta_i - \mu_i)}{\sigma_i^2},$$
$$\nabla_{\sigma_i} \log v_{\mu,\sigma}(\theta) = \frac{(\theta_i - \mu_i)^2 - \sigma_i}{\sigma_i^3}.$$

Sampling with a baseline Given enough samples, it is possible to determine the reward gradient with arbitrary precision. Each sample requires an entire state-action history, which is computationally expensive and time consuming. It is possible to obtain a cheaper gradient estimate by drawing a sample θ and comparing its reward r to a *baseline* reward b, computed as a moving average over the previous samples. If $r > b$ then ρ is adjusted to increase the probability of θ, the opposite is done if $r < b$. Using a step size $\alpha_i = \alpha\sigma_i^2$, with α constant, the following parameter updates are obtained:

$$\Delta\mu_i = \alpha(r - b)(\theta_i - \mu_i),$$
$$\Delta\sigma_i = \alpha(r - b)\frac{(\theta_i - \mu_i)^2 - \sigma_i^2}{\sigma_i}.$$

Natural scores In addition, gradient updates can be done by means of the so called *natural gradient*, when ρ in the gaussian hyper-policy is parametrized with $\log(\sigma_i)$ instead of just σ_i. The parameter udpate is defined as follow:

$$\rho \leftarrow \rho + \zeta F^{-1} \nabla_\rho J(\rho), \tag{2.10}$$

where ζ is the learning rate and F is the *Fisher Information Matrix* of the gradient score, i.e., the variance of the gradient of the log-likelihood function w.r.t. the policy parameters, defined as:

$$F := \mathop{\mathbb{E}}_{\theta \sim \nu_\rho} \left[(\nabla_\rho \log \nu_\rho(\theta))(\nabla_\rho \log \nu_\rho(\theta))^T \right]. \tag{2.11}$$

It quantifies how much information about the unknown parameters is carried by an action sampled from $\pi_\theta(\cdot|s)$.

The matrix will be in the form:

$$F := \begin{bmatrix} \frac{1}{\sigma^2} & 0 \\ 0 & 2 \end{bmatrix}, \tag{2.12}$$

therefore, the inverse used in (2.11) will be:

$$F^{-1} := \begin{bmatrix} \sigma^2 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}. \tag{2.13}$$

Both the baseline sampling and the natural gradient can be added in the PGPE algorithm (Algorithm 2.1), either combined together or one at a time. In particular, the baseline update would be the last operation to perform before completing an iteration of the for loop; on the other hand, the natural gradient update would substitute the standard update of the hyperpolicy parameters. The following modified algorithm presents both the proposed modifications:

Algorithm 2.2 PGPE with baseline and natural gradient update

Input: N number of sampled policy parameters
M number of episodes per policy parameter
Ite number of iterations
$(\alpha^h)_{Ite-1}$ learning rate schedule

Initialize the hyperpolicy parameters ρ^0 arbitrarily
for $h = 0, 1, \dots, Ite - 1$ **do**
  Sample N policy parameteers $\{\theta_i^h\}_{i=1}^N$ independently from υ_{ρ^h}
  Collect M trajectories $\{\tau_{i,j}^h\}_{j=1}^M$ independently for each $\{\pi_{\theta_i^h}\}_{i=1}^N$
    where each reward is taken as $r = [(r^1 - b), \dots, (r^M - b)]^T$
  Update the hyperpolicy parameters $\rho^{h+1} = \rho^h + \zeta^h F^{-1} \nabla_\rho J(\rho^h)$
  Update baseline b accordingly
end for

As a drawback, PGPE cannot perform multiple gradient steps using the same episode more than once, this holds since the hyperpolicy parameters change as an effect of the gradient update.

2.2.4. Actor Critic approaches

As stated in Section 2.2, the *actor-critic* approach combines the evaluation functionalities of value-based approaches with the search strategies of the policy based approaches. Both the presented algorithms are *off-policy*, so they learn a *target policy* while using a *behavioural policy* to interact with the environment.

Soft Actor Critic (SAC)

Soft Actor Critic [SAC, 7], is an off-policy actor-critic Deep RL [9] algorithm, based on the maximum entropy reinforcement learning framework. The actor aims at maximizing the expected reward while also maximizing the *entropy*, i.e., the level of unpredictability in the actions selected by the policy. Indeed, the goal is to succeed at the task while behaving as randomly as possible.

This algorithm is based on three main components: an *actor-critic* architecture, which presents separate policy and value function networks, an *off-policy* formulation, that allows reusing past collected data, increasing efficiency, and *entropy maximization*, which enables stability and exploration. Neural networks are not going to be further analyzed in this work.

Usually, standard RL aims at maximizing the expected sum of rewards. SAC, on the other hand, considers a more general maximum entropy objective, favoring stochastic policies by increasing the objective with the expected entropy of the policy:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t)) \right], \tag{2.14}$$

where α is the *temperature* parameter; it determines the importance of entropy against the reward, therefore controls the stochasticity of the optimal policy. The conventional objective can be retrieved in the limit as $\alpha \to 0$. This new objective incetivizes the policy to explore more widely, while ignoring unoptimizing avenues, resulting in a smaller sample complexity.

For further in-dept analysis of the algorithm one can refer to [7].

Optimistic Actor Critic (OAC)

Optimistic Actor Critic [OAC, 3] arises as an evolution of SAC, highlighting and solving two issues that prevented efficient exploration. According to [3], combining a greedy actor update together with a pessimistic estimate of the critic, can lead to *pessimisitc underexploration* i.e., the avoidance of actions that the agent does not know yet. Moreover, most algorithms are *directionally uninformed*, it means that they will sample actions in opposite directions from the current mean with equal probability, this behaviour can be wasteful since, typically, actions along certain directions are needed more than others.

The OAC algorithm copes with both the aforementioned issues, approximatig a lower

and upper confidence bound on the state-action value function. By doing so it is possible to apply the principle of *optimism in the face of uncertainty* [2], performing directed exploration bounded by the upper bound, while still using the lower bound to avoid underestimation.

OAC can avoid pessimsitic underexploration because it uses the upper bound to determine exploration covariance; moreover the exploration policy is not constrainted to have the same mean as the target one, so the algorithm is *directionally informed*. The latter property, allows to reduce the wasting arising from sampling portions of the action space that have been already explored by previous policies.

Once again, for a further analysis one can refer to [3].

3 | Robot Air Hockey Challenge

In this chapter the AirHockey challenge will be described. At the beginning there will be an explanation of the air hockey game. After that there will be a delve into the details of the challenge, describing the various phases and the framework provided by the organizers. A particular focus on the constraints will close the chapter.

3.1. What is Air Hockey

Air Hockey is a competitive tabletop game involving two players, equipped with a mallet and a puck. The matches take place on a specifically designed table, the playing surface consists of a smooth surface perforated with numerous small holes, through which the air is pumped, creating a cushion of air that significantly reduces the effect of friction on the puck, allowing it to glide quickly on the table.

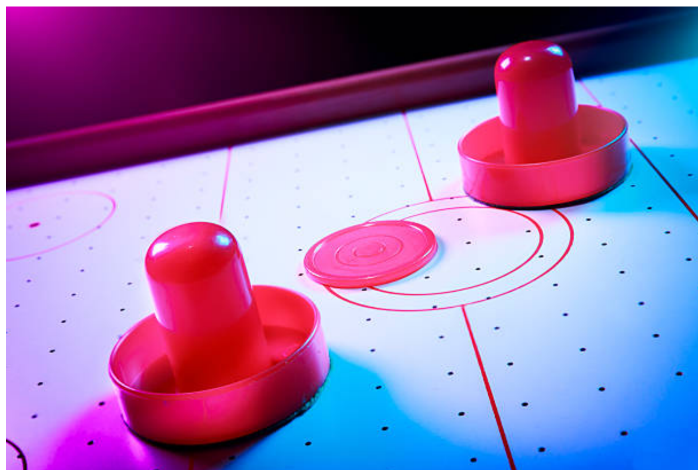


Figure 3.1: A real-world example of puck, mallet and air hockey game field.

The goal of the game is to manouver the puck into the opponent's goal while, at the same time, preventing the opponent from scoring. The players utilize the mallet to hit the puck, aiming to control its trajectory and overrun the opponent's defence. The game

requires strategic positioning, quick reflexing and, at least in a human player, precise hand-eye coordination, to control the puck and succesfully execute defensive and offensive maneuvers.

3.2. Challenge Motivation

As stated in Chapter 1, the principal motivation of this challenge consists in *closing the reality gap*, i.e., the discrepancy between the performance of an RL agent in simulation, and its performance in the real world. Reinforcement Learning models are extensively trained in simulated environments, before being deployed in real world scenarios. This is done since real-world experiments are expensive and can potentially be dangerous, as an example one can think to an agent that is testing a wrong policy in an autonomous driving environment, a wrong action might potentially crash the car. However, model mismatches, observation delays, sensors noise and actuators limitations might lead to a significantly reduction of performance while interacting with the real world.

The Robot Air Hockey Challenge arises in this context, providing a more restricted dynamic scenario to test possible solutions to the aforementioned limitations.

Challenge background

The Air Hockey Challenge has beed organized by the *"Technische Universitat Darmstad (TUD)"*. In a previous work, the organizers showed how to apply a method to learn a robotic task in simulation, while avoiding constraints violation in the learning process, however this method, called [ATACOM, 13] resulted unsuitable for real-world applications since model errors and external disturbances could cause unexpected violations of the constraints. In [12], Liu et al. proposed an approach based on advanced optimization techniques, to show how a general-purpose robot was able to achieve performances similar to the ones of a task specific robotic arm in the hit tasks of the air hockey game. Moreover, this work also showed how two agents were able to play a full game in simulation. However this approach was never tested in a real-world environment.

3.3. Challenge organization

The challenge was structured into three main phases, *Warm-Up*, *Qualifying* and *Tournament*, described in the following paragraphs. For each stage there were different tasks required for robot air hockey. At the end of the challenge a real-robot simulation is expected for the top-three teams.

The participants were provided with an environment that simulated an air hockey game field and the robot, together with all the necessary APIs (Application Programming Interface) needed to control it. Each team developed an agent, testing it locally, and then submitted it to the organizers' cloud server for the evaluation. Except done for the warm-up phase, the submitted agents were evaluated in a modified simulation environment, to mimic the sim-to-real (simulation to reality) gap.

After every submission, the participants were able to download a dataset, collected by the evaluator, to analyze the agent's behaviour during the simulation.

The competition spanned during 41 weeks of 2023, from 20th February to 1st November.

A total of 46 teams participated to the challenge, coming from international universities and research centers.

3.3.1. Warm-Up

In the warm-up phase the participants were provided with an ideal environment (no disturbances added) and a 3-degrees-of-freedom robot. An example of the robot can be seen in Figure 3.2.

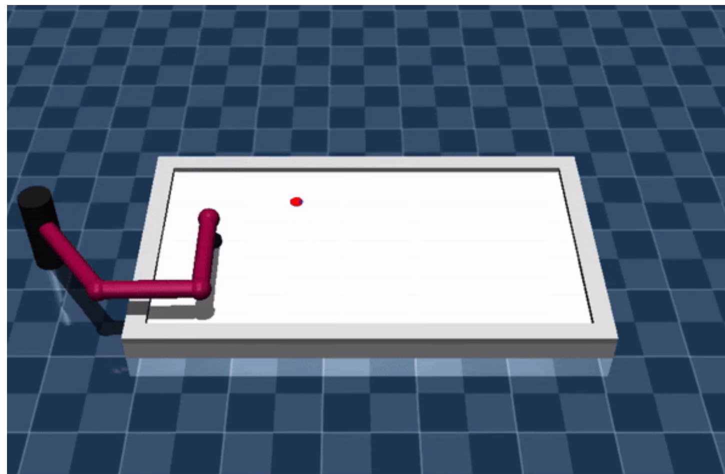


Figure 3.2: 3dof-robot (image taken from the website of the challenge).

As the name suggests, the aim of this phase was to familiarize with the tasks, the environments, and the API. The evaluator in the cloud server was the same provided in the simulator.

During the warm-up, the agent had to perform two tasks:

- **Hit**: the puck was initialized randomly on the left side of the table with initial

velocity equal to zero. The objective was to hit the puck to score a goal as fast as possible.

- **Defend**: the puck was randomly initialized on the right side of the table with a random velocity, heading the left. The task consisted in stopping the puck on the left side of the table and prevent it from getting scored.

Both the tasks were expected to be accomplished without breaking some constraints, that will be described deeper later.

3.3.2. Qualifying

The qualifying stage was ran on a general purpose 7-degrees-of-freedom robot, in particular on a simulator emulating a KUKA iiwa 14 LBR.

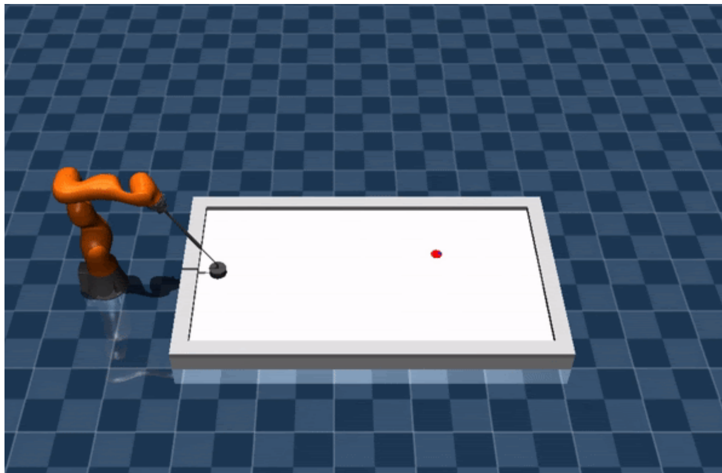


Figure 3.3: 7dof-robot, KUKA iiwa 14 LBR (image taken from the website of the challenge).

The evaluator in the cloud server was modified in order to simulate differenty types of real-world problems including:

- noise in the observation;
- noise in the torque applied to the joints;
- loss of tracking of the puck (i.e., the agents saw the previous state of the puck);
- model mismatch (for example an imperfect knowledge of the robot dimensions).

Each team was able to submit its solution only once per day and could download the dataset at the end of the evaluation. Each evaluation was conducted with 1000 episodes,

i.e., 2.8 hours of real world experiments.

In this phase, the agent had to perform 3 different tasks, one of them also included an opponent:

- **Hit**: an opponent robot moved in a predictable pattern, the puck was randomly initialized with a small velocity. The objective was to score the goal as many times as possible.
  The episode was considered finished if the puck bounced back or was scored, in particular in was considered as a success if the puck was in the opponent's goal when the episode terminated.
- **Defend**: the puck was randomly initialized on the right side of the table with a random velocity, heading the left. The tasks consisted in stopping the puck on the left side of the table and prevent it from getting scored.
  The episode terminated if the puck returned to the opponent's side or was scored or its speed drop below a certain threshold. It was considered a success if the puck was in the range where hits could be made and its longitudinal speed was below a threshold.
- **Prepare**: the puck was initialized close to the table's boundary and was unsuitable for hitting. The task consisted in controlling the puck to move it into a good hit position, while remaining in the agent's side of the game field.
  The episode terminated if the puck crossed the middle line that connected the middle points of the two goals or if the puck was in the opponent's side of the table. It was considered a success if the puck stopped where its could be made and its longitudinal speed was below a threshold.

As described in Section 3.5.3, the evaluation metric kept into account the success rate of each task, and the *deployability*, i.e., the amount of violated constraints. According to this metrics, each agent was categorized into three levels: *Deployable*, *Improvable* and *Non-deployable*. Only "Deployable" and "Improvable" agents were qualified for the next phase.

3.3.3. Tournament

A maximum number of 16 teams was allowed to acces the tournament. In this phase each team had to develop an agent capable of playing a whole game, therefore an additional methodology to allow the agent switching from a task to another had to be developed.

The tournament was divided in two sub-phases, between the twos an adjustment phase

was present to refine the developed agent. A *double round-robin schedule* was applied for the tournament. This means that each team competed against every other participant twice, once at home and once away.

The organizers provided a hard-coded baseline agent to test and validate the developed agents. Each match lasted 15 minutes (45.000 steps), every 500 steps was considered as an episode. Each agent had 15 seconds (750 steps) in order to execute a shoot able to cross the center line, this timer was reset every time the puck entered the player's side. Violating this rule resulted in a *fault*, after 3 faults the opponent's agent was granted with an extra point. Each successful score counted as 1 point. An agent won the game if it collected more points and its deployability score was less then 1.5 times the total number of episodes (i.e 1.5 * 45000 / 500 = 135). For what concerns the final ranking, a match win resulted in accumulating 3 points, a draw 1 point while a loss provided 0 points. Moreover, the agent would have lose the game if it was classified as non-deployable during the match.

The final ranking was determined by the results of the two sub-phases.

3.4. Framework

The Air Hockey challenge was built upon MushroomRL, developed by D'Eramo et al. [4], a Reinforcement Learning library. The general framework of the challenge consisted of two main components, the *Agent* and the environment with which the agent was able to interact thanks to the *AirHockeyChallengeWrapper* interface. A schema of the framework can be seen in Fig, 3.4.

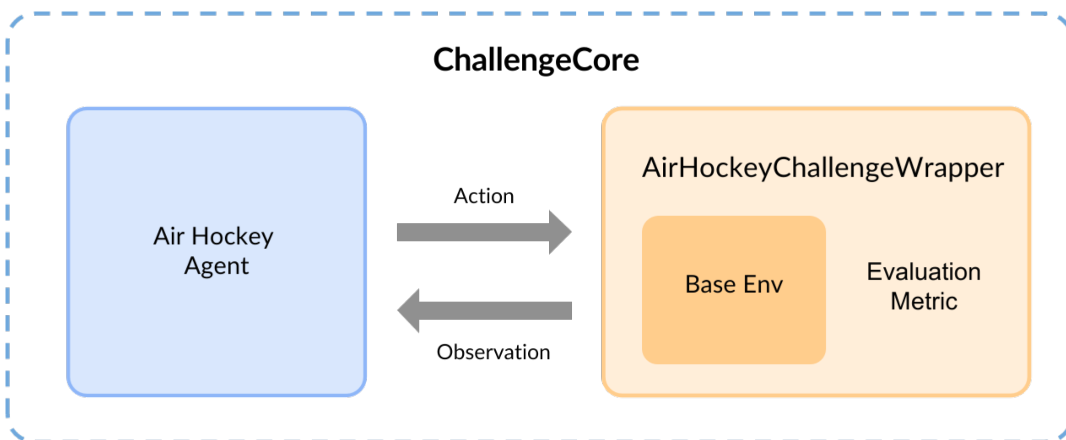


Figure 3.4: Challenge framework (image taken from the website of the challenge).

Participants had to develop the agent that interacted with the enviroment.

3.4.1. Environments

The *AirHockeyChallengeWrapper* provided by the organizers was built around a *Base Env* and processed the information necessary for the challenge evaluation.

At a lower level, the robot was controlled by a *Tracking Controller*, a FeedForward-PD controller, responsible of sending torque commands to the robot:

$$\tau_{cmd} = M(q)\ddot{q}_d + c(q, \dot{q}) + g(q) + K_p(q_d - q) + K_d(\dot{q}_d - \dot{q}). \tag{3.1}$$

The *Trajectory Interpolator*, by default a cubic polynomial, was used to interpolate the trajectory points between two consecutive commands. A *Joint safety limiter* was also added to avoid that the command exceeded the position or velocity limits. A summarizing scheme can be found in Figure 3.5.

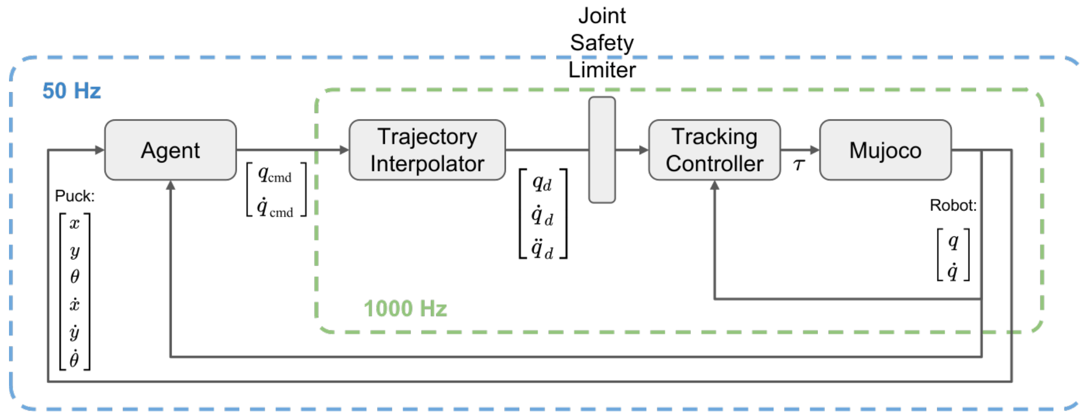


Figure 3.5: Control paradigm (image taken from the website of the challenge).

The simulation was carried out using a *MuJoCo simulator*. The simulation frequency was set to $1000Hz$ while the control one at $50Hz$. The observation provided by the environment was made by:

- Puck Position and velocity: $[x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]$, with θ yaw angle;
- Joints position and velocities: $[q, \dot{q}]$;
- Opponent's Mallet position (if the environment included an opponent): $[x_o, y_o, z_o]$.

Puck position and opponent's mallet position were expressed in $[meters]$, for x and y coordinates, while θ was expressed in $[radians]$. The same holds for the puck's velocities

([*meters/seconds*] and [*radians/seconds*]). The joints position and velocities were expressed in [*radians*] and [*radians/seconds*]. All the above values were expressed in robot coordinates, further specified in Section 3.5.1 and Section 3.5.2.

The desired control action consisted in the desired joints positions and joints velocities, in the warm-up phase; during qualifying and tournament it was customizable, since the participants could express the desired trajectory interpolation order of the *Trajectory Interpolator*. The possible choices were:

- *3, cubic interpolation*: the action command contained the desired [position, velocity]. A cubic polynomial was used to interpolate the intermediate steps;
- *1, linear interpolation*: the action command contained the desired [position]. A linear polynomial was used to interpolate the intermediate steps;
- *2, quadratic interpolation*: the action command contained the desired [position]. A quadratic function used the previous position, velocity and the desired position to interpolate the intermediate steps;
- *4, quartic interpolation*: the action command contained the desired [position, velocity]. A quartic function used the previous position, velocity and the desired position, velocity to interpolate the intermediate steps;
- *5, quintic interpolation*: the action command contained the desired [position, velocity, acceleration]. A quintic function was computed by the previous position, velocity, acceleration and the desired position, velocity and acceleration to interpolate the intermediate steps;
- *-1, linear interpolation*: the action command contained the desired [position, velocity], the acceleration was computed based on the derivative of the velocity. This interpolation was not proper, but was useful to avoid oscillatory behaviours in the interpolation;
- *None*: the agent would send a complete trajectory between each action step, [position, velocity, acceleration] of each joint.

Air Hockey Table

The dimensions of the Table, the puck and the mallet, are specified in the Figure 3.6:

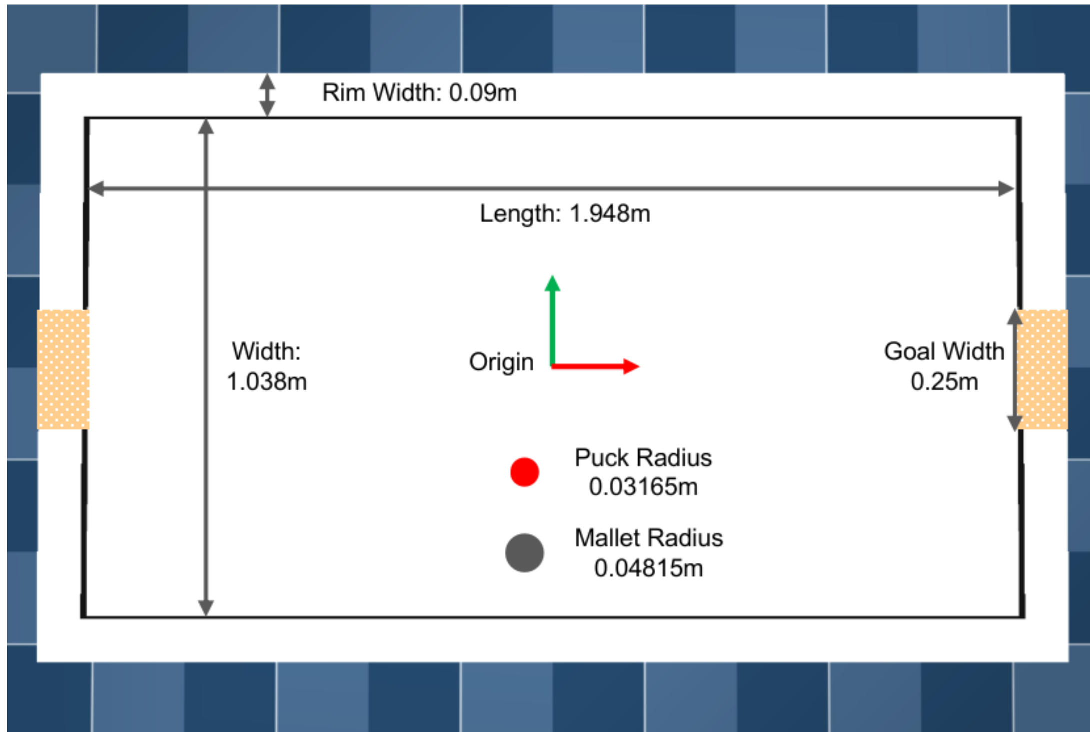


Figure 3.6: Challenge's air hockey table, a smaller version of a standard air hockey table in which also the puck and the mallet got shrunk (image taken from the website of the challenge).

The game table is a smaller version of the standard air hockey table, both the mallet and the puck got shrunk as well. In the center of the table a two-dimensional system of reference was inserted, from now on called *World coordinates*, that could be used to keep track of the puck and mallet positions, observed respect to it.

3.5. Agents

In this section the agents employed in the warm-up, qualifying and tournament stages are going to be described.

3.5.1. Planar Robot - 3 Degrees of Freedom

The base planar robot, used in the warm-up phase, was located at $[-1.51, 0.0, 0.1]$, the orientation was aligned with the world's frame.

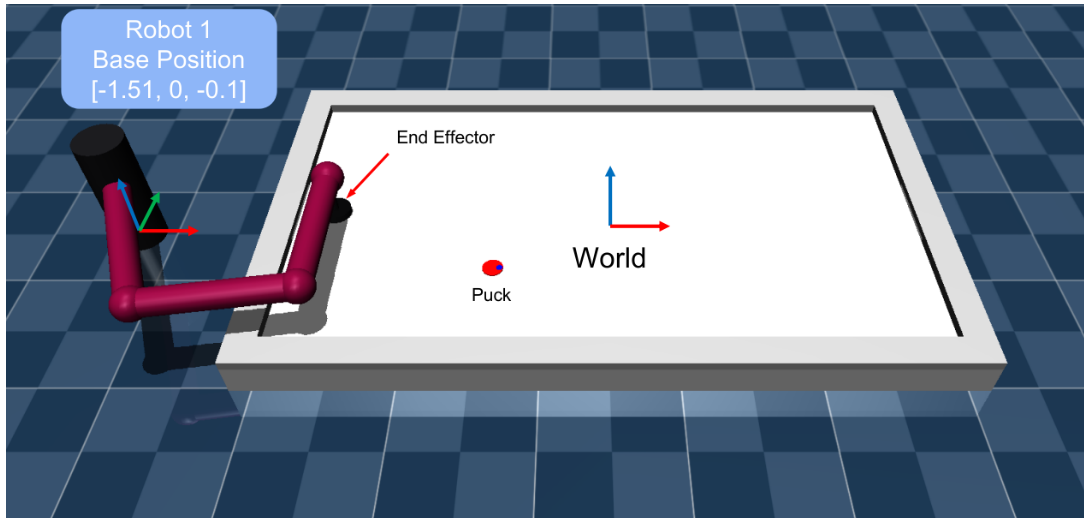


Figure 3.7: 3DoF planar environment robot (image taken from the website of the challenge).

The planar robot was composed by three joints, providing three degrees of freedom. At the end of the last rod, an end effector is present, touching the game table. The base position of the robot was considered as the center of a three-dimensional system of reference, from now on called *Robot coordinates*. Despite the presence of the third dimension, this robot could not move the end effector along the z-axis. The positions of the puck and the mallet could be computed in both robot and world coordinates.

3.5.2. KUKA iiwa14 LBR Robot

The KUKA robot was equipped with 7 joints, in particular, a universal joint on the end-effector was added, in order to increase the robot's flexibility. The universal joint was a passive joint that adapted the joint position based on contacts. In the simulation, a plugin to compute the joint's angle to keep the mallet parallel to the table was used. The position of the universal joint was not observed. The joints were enumerated, from 1 to 7, starting from the base of the robot. Particular attention was given to joints 4 and 7, called *elbow* and *wrist*, since they were subject to higher efforts than others during movements.

Like in the 3DoF case, a three-dimensional system of reference was inserted at bottom of the base of the robot. This time the robot was also able to move the end-effector along the z-axis, therefore also the world coordinates included a z value.

Moreover, the *End-Effector* was defined as the tip of the extension rod before the universal joint. The end-effector's position could be fully determined by the robot's joint position and forward kinematics. The base position of the robot is depicted in Figure 3.8, that

includes also an opponent.

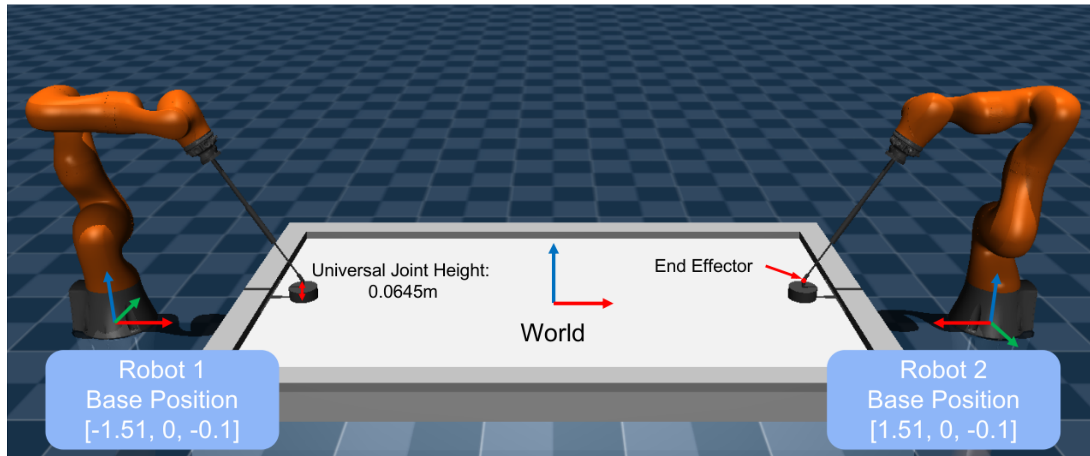


Figure 3.8: Kuka iiwa14 LBR Robot (image taken from the website of the challenge).

3.5.3. Evaluation metrics

As stated in the previous sections, each phase was evaluated according to some metrics. For what concerns warm-up and qualifying phases, each task was evaluated by 1000 episodes. Two metrics were computed in the evaluation:

- *Success rate*: a success criterion was defined for each task, it was checked at the end of every episode. Each episode could terminate because of two reasons: 1. Maximum number of steps reached; 2. No further interaction can be in the episodes;
- *Deployability*: this scored assessed the agent under multiple aspects. Each metric was assigned to one or more penalty points, depending on the level of risk. When a constraint was violated, the deployability penalty points were added up. Same violations were counted only once per episode.

3.6. Constraints

In this section, the constraints applied in the evaluation are going to be further expanded; the number next to each constraint name represents the penalty points assigned to the constraint violation, l and u represent lower and upper bounds, respectively.

There were five different types of constraints violations.

End-Effector's Position Constraints (3) The x-y-position of the end-effector should remain within the boundaries of the table. The z-position of the end-effector should remain

within a range. The end-effector's position could be computed as

$$p = ForwardKinematics(q)_x.$$

The constraint can be represented as:

$$l_x < x_{ee},$$
$$l_y < y_{ee} < u_y,$$
$$table_height - tolerance < z_{ee} < table_height + tolerance.$$

These constraints were very strict, with very small tolerance w.r.t. physical limits of the table and the robot, since an unfeasible command could potentially damage the table, the end-effector or robot actuators. For example, if the provided command required the robot to move the end-effector to a negative z value, in world coordinate, the robot would try to push the end effector inside the table, damaging the rod and causing emergency brakes in the robot joints, due to eccessive torque applied to win the vincolar reaction coming from the table.

Join Position Limit Constraints (2) The joint position should have not exceed the position limits. In the real-robot, violations of the position limit would trigger the emergency break.

$$q_l < q_{cmd} < q_u.$$

Joint Velocity Limit Constraints (1) The velocity should have not exceed the velocity limits. The controllers in the real-robot were set not to exceed the velocity limits.

$$\dot{q}_l < \dot{q}_{cmd} < \dot{q}_u.$$

Computation Time (0.5 - 2) The computation time at each step should have been smaller than $0.02s$. There were different possible violations for this constraint:

- Penalty points 0.5:

$$0.1s \geq \text{maximum computation time} > 0.02s.$$

- Penalty points 1:

$$0.2s \geq \text{maximum computation time} > 0.1s.$$

- Penalty points 2:

$$\text{maximum computation time} > 0.2s,$$
$$\text{average computation time} > 0.02s.$$

Link Constraint In the 7DoF environment, additional constraints were added that ensured positions of the elbow and wrists (joints 4 and 7 of the robot, starting the counting from the base) were above a threshold.

$$z_{elbow} > 0.25,$$
$$z_{wrist} > 0.25.$$

4 | Methodology

This chapter contains a description of the methodologies employed. Firstly, the challenge will be framed inside the MDP framework, then the developed rule-based agent will be described, together with its four tasks: *hit*, *defend*, *prepare*, *Default Position*, from now on called *default*, and the adopted noise filtering method. The chapter will conclude with the analysis of the final Hierarchical Agent and its main components, a switcher and a Finite State Machine.

4.1. Air Hockey as an MDP

As formalized in Section 2.1, a Markov Decision Process is a framework typically employed while working with sequential decision making problems. For what concerns the AirHockey challenge, the mapping of the environment to an MDP is described in the following sections.

4.1.1. State Space

For the general purpose 7DoF robot, a configuration results fully described by the joint angles: $q = \{q_i | i \in 1 \dots 7, q_i^{min} \leq q_i \leq q_i^{max}\}$. The space of all the possible joint positions is known as the *Joint Space*. The end-effector position ee, in a given configuration q, can be computed by means of *Forward kinematics*, $ee = FK(q)$. The *Task Space* contains all the allowed end-effector poses. From any position in the task space it is possible to compute one, or more, joint configurations exploiting the *Inverse kinematics*, $q = IK(ee)$. Indeed, *Forward Kinematics* describes the process of determining the position and the orientation of the end-effector in the cartesian space, based on the joint angles. Conversely, the *Inverse Kinematics* deals with the opposite problem: determining the joint angles necessary to achieve a specific orientation and position of the end-effector.

To build the continuous *State Space*, the observation provided by the environment, described in Section 3.4.1 was slightly modified. The rotational axis element θ, therefore also $\dot{\theta}$, was discarded.

In order to reduce the noise in the observation, while maintaining the form of the state unchanged, an additional preprocessing was performed, as further analyzed in Section 4.2.5.

4.1.2. Action space

A high-level control setting was developed to transpose the control from the *task space* into the *joint space*. The agent directly controls the end effector position, in x, y and z coordinates. A lower lever controller is then used to translate the cartesian coordinates into the desired *action*, composed by joint positions and velocities, since the employed interpolation (described, like the controller, in Section 3.4.1) is the cubic one. Given the cartesian coordinates, the joint velocities for the next step, necessary to reach the point, are computed by means of Inverse Kinematics and *Quadratic Programming* (QP), an optimization technique that deals with quadratic objective functions, subject to linear equality and inequality constraints. After that, the necessary joint positions are retrieved using the previous positions and the computed velocity:

$$joint_pos_{t+1} = joint_pos_t + \frac{(joint_vel_t + joint_vel_{t+1})}{2}0.02, \tag{4.1}$$

where 0.02 [seconds] represents the time duration of a single step; the sum of joint velocities is divided by two in order to provide smaller steps, to reduce the risk of constraint violations.

4.1.3. Reward function

The designed reward associated with the MDP is task-dependent, the agent is positively rewarded as it accomplishes a task, or part of it. Despite the task differences, there is a common penalization in the reward: when the constraints are violated, when the opponent scores a goal or when the agent does not complete the task. In particular, for what concerns the constraints violations, their penalization is computed as follows:

$$normalized_violated_constraints = \frac{\sum_i (\phi_i \cdot \frac{1}{N} \sum_{j=1}^{N} \sigma_j)}{\sum_i \phi_i}, \tag{4.2}$$

$$\sigma_j = current_value_j - limit_j, \tag{4.3}$$

where N is the total number of violated constraints, ϕ_i is the penalty point associated to the violation of constraint i (described in Section 3.6), while σ, also called *slack*, represents the difference between the current value of the constraint variable and the

limit, for example:

$$limit_{z_{elbow}} = 0.25,$$
$$current_{z_{elbow}} = 0.3,$$
$$\sigma = 0.3 - 0.25 = 0.05,$$

if this difference is positive, then the constraint is violated since the value is over the limit. In 4.2 only positive values of σ are considered. Each reward function will be further explained in the specific task description in Section 4.2.

4.2. Rule-Based Agent

The developed rule-based agent, aims at exploiting a deterministic policy optimized via Policy Gradient With Parameter-based Exploration [PGPE, 20], following the approach in [10]. The actions from the *task space* are translated into the *joint space*, combining inverse kinematics and Anchored Quadratic Programming [AQP, 12]: a modified version of QP that, unlike Quadratic Programming, focuses on the optimization of the difference between the current state and a reference value, called *anchor*. A further explanation will be provided in Chapter 6.

The policy set is divided into 4 main branches, resembling the challenge tasks and described in the following sections: *Hit*, *Defend*, *Prepare*, *Default*: in the *Hit* the agent has to hit the puck in order to score a goal, in the *Defend* the agent has to prevent an incoming puck from scoring, stopping it without bouncing it back, in the *Prepare* the agent has to reposition the puck in a place suitable for hitting, and finally, in the *Default* the agent has to go back to a default configuration from which it will start another task. The policies are further expanded into phases, each of them associated to a specific subtask.

Task framework In all the tasks, except in the *Defend*, the agent acts in the following framework. To select the next position of the end-effector, a system of polar coordinates centered in the puck is used. The main components of this system are:

- β: the angle between the line puck-goal and line the puck-end-effector;
- γ: the angle between the horizontal and the line puck-goal;
- r: also called *radius*, is the segment connecting the puck and the end-effector;
- $d\beta$: the variation of the β angle in the desired next position of the end-effector;
- ds: the variation of length of the radius in the desired next position of the end-effector.

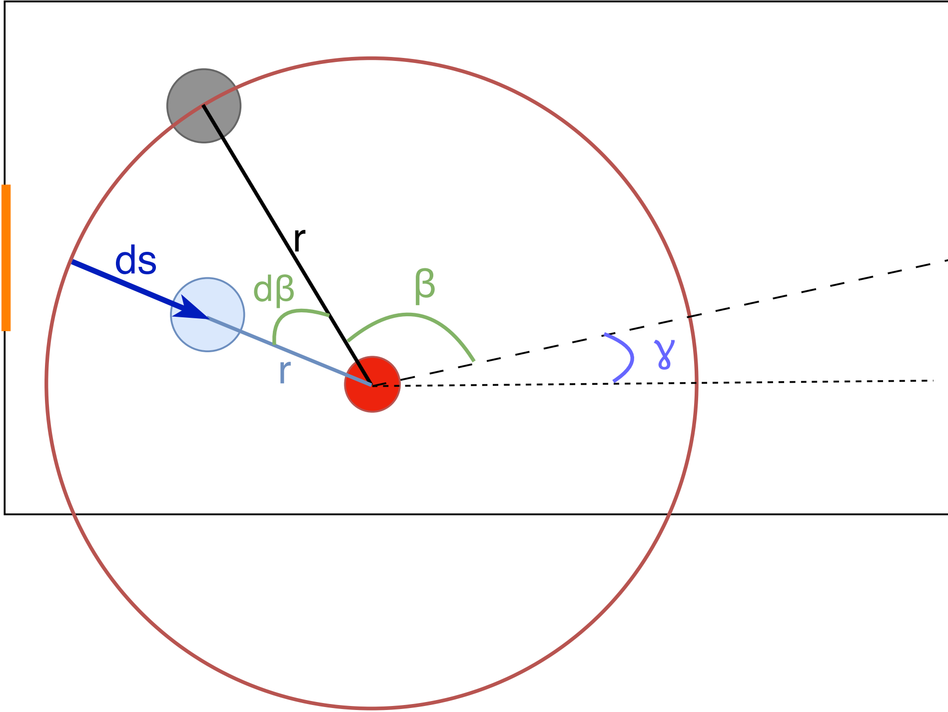


Figure 4.1: Coordinates used to find the next desired position of the end-effector. The goal positions are highlighted in orange. The current position of the end-effector is the gray one, while the blue one is the desired position in the next step. To reach that position, a variation of the radius, ds, and the angle, $d\beta$, is necessary. This variations, happening simultaneously, will result in a curved movement.

The goal considered is always the one of the opponent. In the *Prepare* task the γ angle is not considered and the angle β will span from the horizontal line and the radius. A visual representation of the framework can be observed in Figure 4.1.

The developed agent will compute the next desired position of the end-effector in world coordinates, converting it in robot coordinates before providing it to the lower level controller. The description of this two systems of reference is provided in Chapter 3.

Each step of the agent will last 0.02 seconds, since the update frequency of the environment, as explained in Section 3.4.1 is 50Hz ($1/50[Hz] = 0.02[s]$). This amount of time will be called either *step* or *dt* from now on. At each step, the agent will apply a variation of $d\beta$ and ds to select a new position of the end effector. This variations are task dependent and will be further explained in the following sections.

Each deterministic rule-based policy is parametric and its parameters, θ_i, will be optimized by means of PGPE (Section 2.2.3).

Despite a natural evolution in its complexity, the developed agent is common for both the

3DoF and the 7DoF robot. Thanks to the underlying mapping from the task space to the joint space, the high level controller does not make any assumption on the underlying model, except that the desired position it computed is reachable. By doing so, the agent can work on a robot with a custom number of joint, as long as it can span the whole task space.

4.2.1. Hit task

In the hit task the agent has to hit the puck to score a goal, as fast as possible and as many times as possible during a match.

This task is divided into 4 *subtasks*, or *phases*:

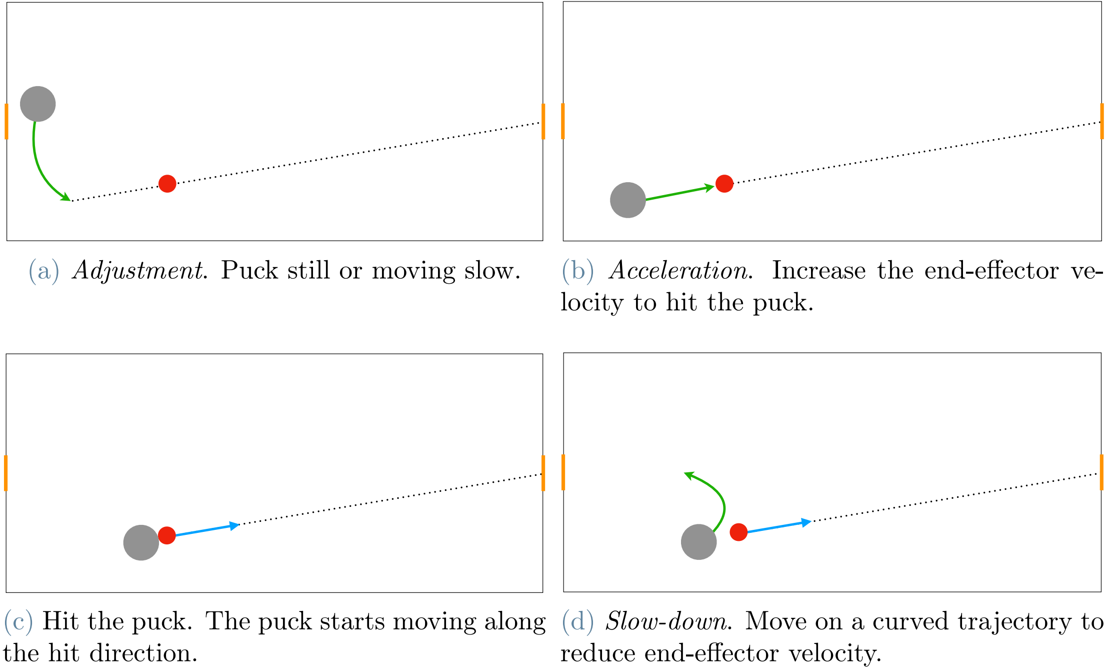
- *Wait*: wait for the puck to move slowly, in this phase the agent stands still;
- *Adjustment*: adjust the end-effector position, in order to place it behind the puck on a hitting trajectory with the goal;
- *Acceleration*: start accelerating the end-effector and stop only after and hitting the puck;
- *Slow-down*: after an hit, move the end effector on a curved trajectory to decrease it's velocity and reduce the probability of constraints violations.

Each subtask is characterized by a time variable, t_{phase}, which represents how many steps the agent spent in the specific subtask. This timer is reset every time that the agent changes task.

Rule based policy The policy that controls the hit task is structured as follow:

$$d\beta = \begin{cases} (\theta_0 + \theta_1 \cdot t_{phase} \cdot dt) \cdot correction & \textit{adjustment} \text{ phase} \\ \frac{correction}{2} & \textit{acceleration} \text{ phase} \\ (\theta_0 + \theta_1 \cdot dt) \cdot correction & \textit{slow-down} \text{ phase} \end{cases}, \tag{4.4}$$

$$ds = \begin{cases} \theta_2 & \textit{adjustment} \text{ phase} \\ \frac{ds_{t-1} + \theta_3 \cdot t_{phase} \cdot dt}{radius + r_{mallet}} & \textit{acceleration} \text{ phase} \\ constant & \textit{slow-down} \text{ phase} \end{cases}, \tag{4.5}$$



(a) *Adjustment.* Puck still or moving slow.

(b) *Acceleration.* Increase the end-effector velocity to hit the puck.

(c) Hit the puck. The puck starts moving along the hit direction.

(d) *Slow-down.* Move on a curved trajectory to reduce end-effector velocity.

Figure 4.2: Hit phases. Goal areas are highlighted in orange. The dotted line represents the ideal direction that the puck should follow in order to score a goal.

where *radius* is the distance between the center of the puck and the end-effector, while *correction* is always defined as:

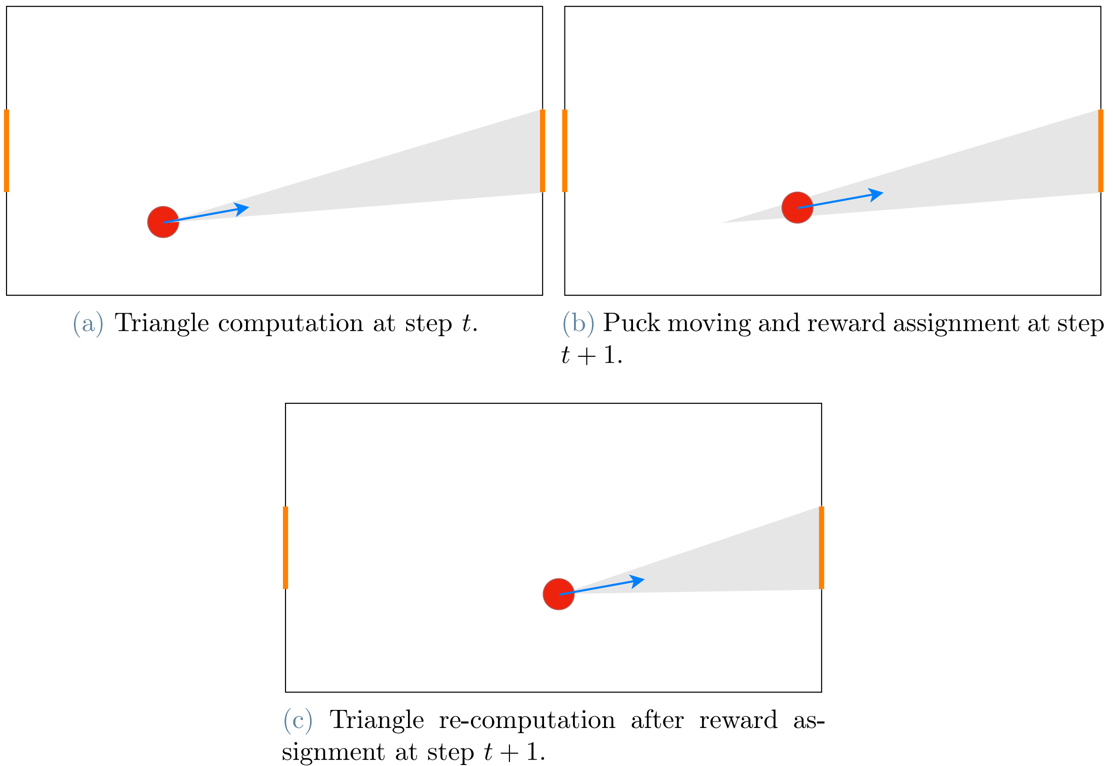
$$correction = \begin{cases} 180 - \beta & y_{puck} \leq \frac{table_width}{2} \\ \beta - 180 & y_{puck} > \frac{table_width}{2} \end{cases}.$$

An example of hit can be seen in Figure 4.2, where the caption describes the subtask.

Reward function To train the agent, a specific reward function for the hit was developed. A hit is considered to be successful if the puck gets inside the opponent's goal, therefore the whole goal area should be considered. Moreover, if the puck moves fast it will be harder for the opponent to stop it, fast hits should be rewarded as well.

The designed reward function is based on the building of a triangle with the opponent's goal area ends and the puck, as vertices (Figure 4.3).

If the puck is still inside the triangle computed at the previous step, a positive reward for the puck's position will be assigned, a second one, proportional to the puck's velocity will be computed, as well. At the end of each time step, after the reward assignment, a new triangle is computed, since the puck's position will change. On the other hand, if the



(a) Triangle computation at step t.

(b) Puck moving and reward assignment at step $t+1$.

(c) Triangle re-computation after reward assignment at step $t+1$.

Figure 4.3: Triangle construction for assigning reward after the mallet hit the puck. Goal areas are highlighted in orange, the blue arrow represents the puck velocity. The gray triangle is the ideal area in which the puck should be to score a goal. If at the next step the puck will be outside the triangle, a negative reward will be applied. After the computation of the reward, the triangle will be recomputed with the new puck position in $t+1$. The coordinates of the puck are the ones of its center.

puck will be outside the triangle, a negative reward will be assigned, the more the puck is outside the polygon, the bigger the penalty.

$$reward_hit = \begin{cases} -\frac{||ee_{pos} - puck_{pos}||}{0.5 \cdot table_diag} & \text{no hit} \\ A + B \cdot \frac{(1-(2\cdot\frac{\alpha}{\pi})^2)\cdot||v_{ee}||}{max_vel} & \text{hit the puck} \\ \frac{1}{1-\gamma} & \text{goal} \end{cases}, \tag{4.6}$$

where A and B are constant values, respectively equal to 100 and 10, $table_diag$ is the diagonal of the table, computed as:

$$table_diag = \sqrt{table_length^2 + table_width^2},$$

$\alpha = \arctan 2(ee_{y_vel}, ee_{x_vel})$ therefore the angle, in radians, between the positive x-axes of a cartesian plan and a point of coordinates (ee_{y_vel}, ee_{x_vel}) laying on it. Finally, max_vel is a constant value equal to the maximum velocity observable in the environment. If the agents successfully hits the puck, it receives an instantaneous reward for the hit, after that, the described approach based on the triangles is applied:

$$reward_hit = \begin{cases} B + ||v_{puck}|| & \text{puck inside the triangle} \\ -diff_angle & \text{puck outside the triangle} \end{cases} \tag{4.7}$$

where B is a constant positive value equal to 10, rewarding the agent for being inside the triangle, $||v_{puck}||$ on the other hand rewards the agent if the puck is moving fast. Moreover:

$$diff_angle = \arctan 2(v_{y_{puck}}, v_{x_{puck}}) - angle_border,$$

where $angle_border$ is the angle between the puck velocity vector and the closest triangle border. The more the puck will go outside the triangle, the more negative the reward will be. As last step, the final reward is computed as:

$$reward = reward_hit - \alpha \cdot \frac{\sum_i (\phi_i \cdot \frac{1}{N} \sum_{j=1}^{N} \sigma_j)}{\sum_i \phi_i}, \tag{4.8}$$

with α constant value, in this case equal to 1, is used to assign more or less weights to the constraint violations (coming from Equation 4.2) w.r.t. to the task completion.

4.2.2. Defend task

In the defend task, the agent has to stop an incoming puck and avoid it from being scored. In the rule-based policy this task is divided into two subtasks depending on the position of the puck w.r.t. the end-effector: *bottom* if end-effector below the puck, *top* if end-effector above the puck.

Rule-based policy In this task the agent aims at stopping the puck on a specific line, called *defend_line*, the x coordinate of the line are known, -0.8 in world coordinates, therefore the agent has to compute only the y coordinate to identify the best point on the line where to intercept the puck.

The desired y is computed as follow:

$$y_{target} = \begin{cases} y_{puck} + \theta_0 \cdot (r_{puck} + r_{mallet}) \cdot \frac{(||v_{puck}||)}{2} & \text{hit from } top \\ y_{puck} - \theta_0 \cdot (r_{puck} + r_{mallet}) \cdot \frac{(||v_{puck}||)}{2} & \text{hit from } bottom \end{cases},$$

where r_{puck} and r_{mallet} are the radii of the puck and the mallet, respectively, while v_{puck} stands for velocity of the puck. After retrieving the coordinates of the intercept point $[x_{target}, y_{target}]$, the agent will move the end effector adding a variation in x and y to the current end-effector position:

$$\delta_x = (\theta_1 + \theta_2 \cdot ||v_{puck}|| \cdot (x_{target} - x_{ee})),$$
$$\delta_y = (\theta_3 + \theta_4 \cdot ||v_{puck}|| \cdot (y_{target} - y_{ee})).$$

The final action is therefore computed as:

$$action = [x_{ee} + \delta_x, y_{ee} + \delta_y].$$

Reward function Since the computation of the interception point is quite straightforward, the defend reward function is simple as well, focused on not violating constraints. A large reward is returned in case of success while no reward will be assigned in case of failure. Moreover, for each violated constraints, a normalized penalty point, according to the constraints penalty points described in Section 3.6, is subtracted by the reward. Summarizing the final reward can be either positive or negative since it comes from the difference between the reward, in case of success, and the penalties of the constraints

violations:

$$reward = \begin{cases} B - \frac{\sum_i (\phi_i \cdot \frac{1}{N} \sum_{j=1}^{N} \sigma_j)}{\sum_i \phi_i} & \text{task successful} \\ -(P + \frac{\sum_i (\phi_i \cdot \frac{1}{N} \sum_{j=1}^{N} \sigma_j)}{\sum_i \phi_i}) & \text{task not successful} \end{cases}. \tag{4.9}$$

B and P are constant values, used to provide the agent a high reward or a high penalty if the task is successful or or not, respectively. The values used in the experiments were B equal to 1000 and P equal to 100.

4.2.3. Prepare task

In the prepare task, the agent has to reposition the puck that is in a position unsuitable for hitting. First of all, it is necessary to define the suitability for a hit.

Check enough space In order to perform a hit there should be enough space for the mallet behind the puck, therefore there should be at least a tolerance of:

$$radius_{puck} + 2 \cdot radius_{mallet}. \tag{4.10}$$

In the agent a flag called *enough_ space* was added, if this flag is False than the agent has to perform a prepare to reposition the puck. To be more conservative and reposition the puck more often, the tolerance is computed as follow:

$$\begin{aligned} x_{tol} &= \left(\left(\frac{table_length}{2} \right) - side_tolerance \right) - |x_{puck}|, \\ y_{tol} &= \left(\left(\frac{table_width}{2} \right) - side_tolerance \right) - |y_{puck}|, \\ tolerance &= radius_{puck} + 2 \cdot radius_{mallet}, \end{aligned}$$

where x_{tol} and y_{tol} are the offset of the puck w.r.t. the table borders and $side_tolerance$ is a value used to add an additional margin to the border, to make more frequent *Prepare*. Finally, the flag *enough_ space* is set to False if either $x_{tol} < tolerance$ or $y_{tol} < tolerance$. A visual representation can be seen in Figure 4.4:

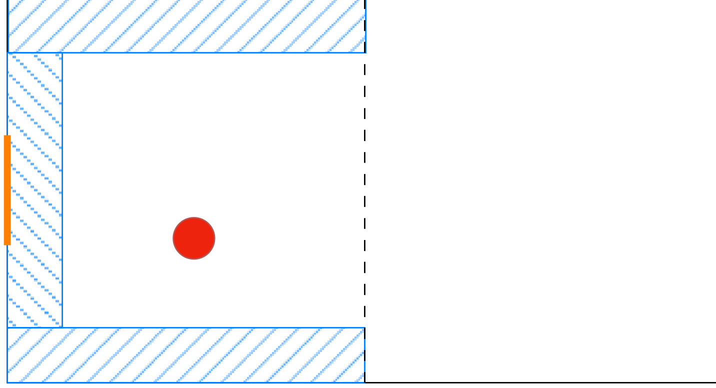


Figure 4.4: *Prepare* areas. If the center of the puck will be inside any of the hatched areas, a *Prepare* will be performed. A *Prepare* can be performed only in the agent's side of the game field.

Phases Similarly to the hit, the prepare is divided into 3 subtasks: *Wait*, *Adjustment*, *Acceleration*; they have the same aim of the ones in the hit. Also in this case, each subtask has a time variable associated t_{phase}, representing how many steps the agent spent in the specific subtask.

Rule Based Policy The policy controlling the prepare is the following:

$$d\beta = \begin{cases} \theta_0 \cdot t_{phase} + \theta_1 \cdot correction & adjustment \text{ phase} \\ correction & acceleration \text{ phase} \end{cases}, \tag{4.11}$$

$$ds = \begin{cases} 5 \cdot 10^{-3} & adjustment \text{ phase} \\ \theta_2 & acceleration \text{ phase} \end{cases}, \tag{4.12}$$

where *correction* is defined as:

$$correction = \begin{cases} \beta - 90 & y_{puck} \leq 0 \\ 270 - \beta & y_{puck} > 0 \end{cases}. \tag{4.13}$$

Reward function The reward function used is the same described in Section 4.2.2 for the *Defend* task.

4.2.4. Default Position task

After the completion of each task, the agent has to return to its default position, called *home* from now on. The home was computed by means of Forward Kinematics with the default joints position and, in world coordinates, it is located at $[-0.85995711, 0.0, 0.0645572]$.

Rule-based policy The policy ruling the return to home is the following:

$$correction = \begin{cases} 360 - \beta & y_{target_point} \geq 0 \\ beta - 0.01 & y_{target_point} < 0 \end{cases} \tag{4.14}$$

$$d\beta = (\theta_0 + \theta_1 \cdot dt) \cdot correction \tag{4.15}$$

$$ds = step_size \tag{4.16}$$

Where $step_size$ is a constant and the small value 0.01 was used in order to avoid the use of β alone. The θ_i used are the same of the *Hit* (Section 4.2.1). Here the $target_point$ represents the home, but, in principle, this function can be used to make the end effector reach any point following a curved trajectory.

When the end-effector reaches the *home* position it will stand still, waiting for a another task to start.

Reward function In this case, like in the prepare, the reward function used in the same of the *Defend* task.

4.2.5. Noise filtering

As introduced in the description of the state space (Section 4.1.1), each observation is preprocessed to reduce the effect of the noise.

The three main sources of noise are represented by:

- *Noisy joint observation*: the observed joints position and velocity can be slightly different from the real ones;
- *Noisy puck observation*: the position and the velocity of the puck provided by the environment can be slightly different from the real one;
- *Loss of Tracking*: the agent might lose the puck, seeing it in its previous state, At time t the agent would see the same position and velocity of the puck in $t-1$.

To smooth out the noise in the puck position and velocity a simplified Kalman Filter is used, in particular the one provided by the organizers and described in [12].

To delete the first type of noise, the developed agent uses its output at the previous time step instead of the observation for the joint position and velocities. The combination of Anchored Quadratic Programming [AQP, 12] solver and Inverse kinematics, used when translating the end-effector coordinates into the ones in the joint space, provides desired robot poses that are reachable, making this approach feasible. As one can imagine, the first observation is a noisy one and cannot be overwritten. However, the sensibility to such noise is negligible.

The magnitude of the noise in the server environment was estimated exploiting the dataset provided at the end of each evaluation. In particular, in the qualifying stage, while evaluating the *prepare*, the puck was initialized still, therefore by looking at the puck's position before a hit it was possible to estimate the noise. To develop a more robust agent, in the local tests, a gaussian noise with zero mean and a higher variance than the estimated one was used.

4.3. Hierarchical Agent

After developing each single task, it is necessary to combine them in order to build a complete agent, capable of playing a full game. To do so, a high-level agent has been developed.

The agent is called *Hierarchical* since it is at the top level in the decision process. This agent decides what task is the best one to execute, based on the observation of the environment and on two main components: the *Switcher* and the *Finite State Machine*.

4.3.1. Switcher

The *Switcher* is the component used to select a new task when the current one is completed. Each task, after its last action, sends a signal to the high-level agent, notifying its completion. The switcher will then select another task, which can potentially be also the same as before. This can happen, for example, if the agent performed a prepare but the puck is still in an unsuitable position for hitting, a second prepare might be necessary.

The structure of the switcher can be seen in figure 4.5.

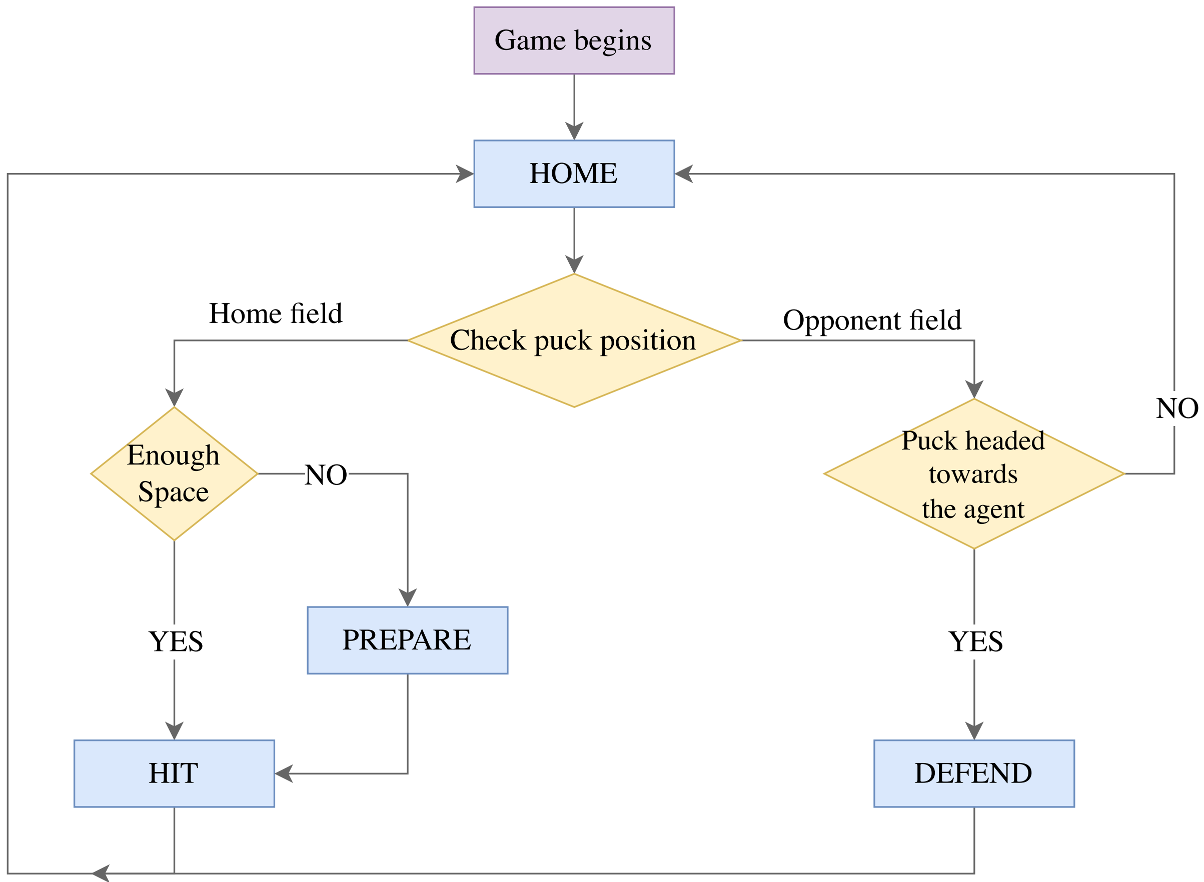


Figure 4.5: Switcher structure. The blue rectangles represent the tasks while the yellow rhombuses the checks that are performed.

The switcher relies on some parameters to pick the right action, like:

$$\text{if } ||v_{puck}|| > \theta \rightarrow \text{defend}$$

therefore it can be trained just like all the other tasks.

4.3.2. Finite State Machine (FSM)

While the switcher can decide what is the next task to perform, a second layer of safety was added, in the form of a *Finite State Machine (FSM)*. An FSM is a computational model used to design systems that can exist in a limited number of states.

Since each task is characterized by its own parameters, the switching process might result in offhanded movements which might violate the constraints. To assist the switcher and ensure a safe and smooth switch among tasks, the FSM forces the agent, at the end of each task, to go back to the home position and start selecting a new task from there.

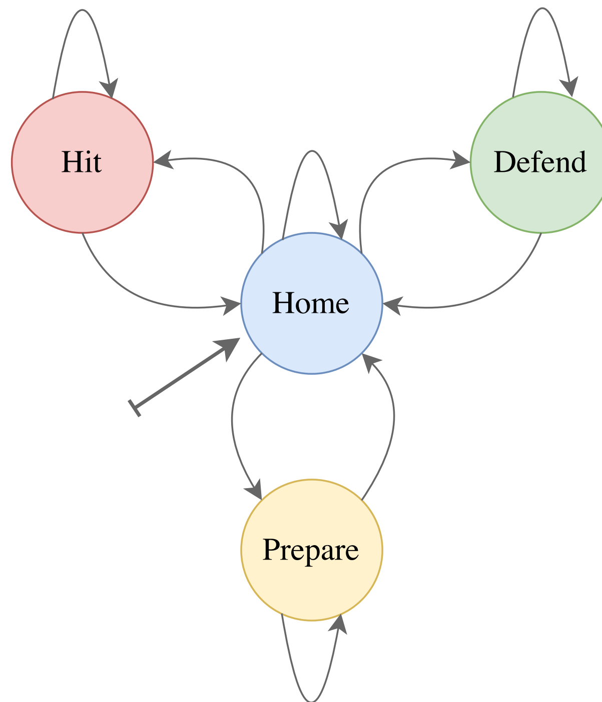


Figure 4.6: Finite State Machine for controlling the transitions among tasks. The agent will always start a game in the *Home* state. The machine allows only for transitions that make the agent perform the same task as before or go home. In order to change task the agent has to visit the *Home* state first.

Figure 4.6 describes the behaviour of the FSM. Each state represents a task while each edge is an allowed transition. The agent will always start a match in the *Home* state. It is possible to remain in the same state but not switching from a task to another without passing from the *Home* state, even if the switcher might suggest a transition of such kind.

5 | Experimental results

In this chapter, the experiments associated with the training runs of the developed agent are presented. The chapter will open with the initial experiments in section 5.1, also highlighting some of the problems encountered during the training. In particular, the successive section will address the aforementioned issues, showing the analysis performed to identify and solve the problem. In the end, section 5.3, will show the training results of the final agent.

5.1. Initial experiments

During the first phase of the challenge, the main objective consisted in becoming familiar with the environment and with the interfaces that were used to control the robot. In particular controlling the agent to reach a specified point or to follow a provided trajectory. When the agent was successfully able to move the end-effector in the desired position, a first version of the rule-based policies was introduced, starting from the *hit* task. This preliminary approach was not based on the framework that exploited the polar coordinates centered in the puck, as described in Section 4.2. This method provided only a variation of ds, considering $d\beta$ as a constant, moreover it considered also a time to arrival, i.e., the amount of time in which the agent had to reach the point. This policy resulted in sharp movements which led to numerous constraints violations. The policy was therefore updated with the one described in Section 4.2.1.

In this challenge, the constraints had a huge impact on the robot. Violating them not only reduced the quality of the agent in the evaluation but, in a real world environment, it might result in damaging the robot actuators, the table or the end-effector. Therefore, while developing the reward function, a lot of importance has been given to the constraints violations. The reward employed is the one described in the equation 4.9, reported here for convenience.

$$reward = \begin{cases} B - normalized_violated_constraints & \text{task successful} \\ -(P + normalized_violated_constraints) & \text{task not successful} \end{cases}, \tag{5.1}$$

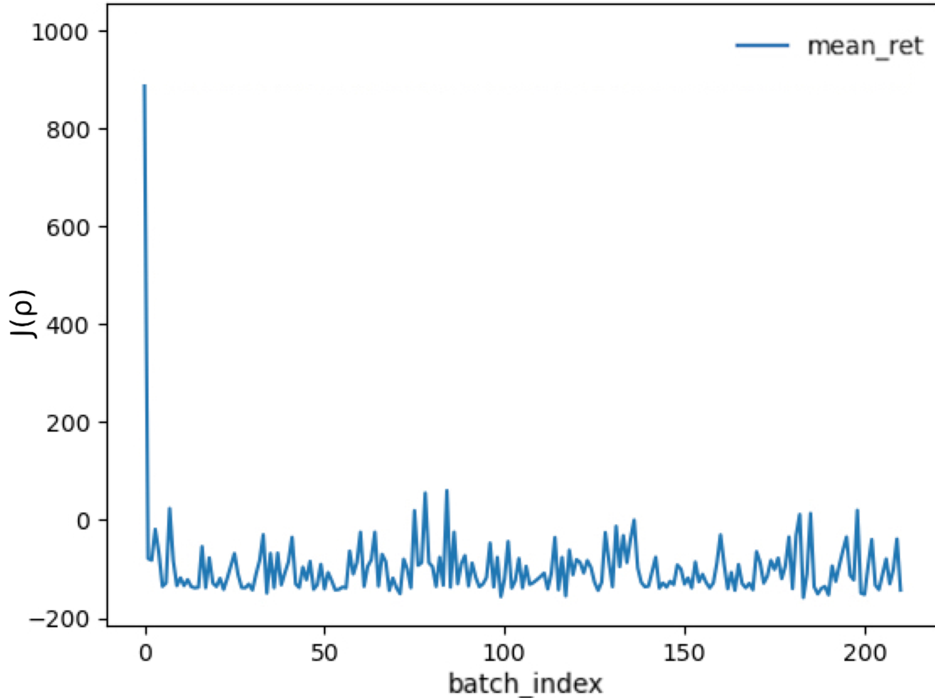


Figure 5.1: Mean return of a training run of the hit task with a simple reward function, aimed at not violating constraints. The agent keeps collecting large penalties since it is not completing the task; the reward function focused on the constraints is too sparse.

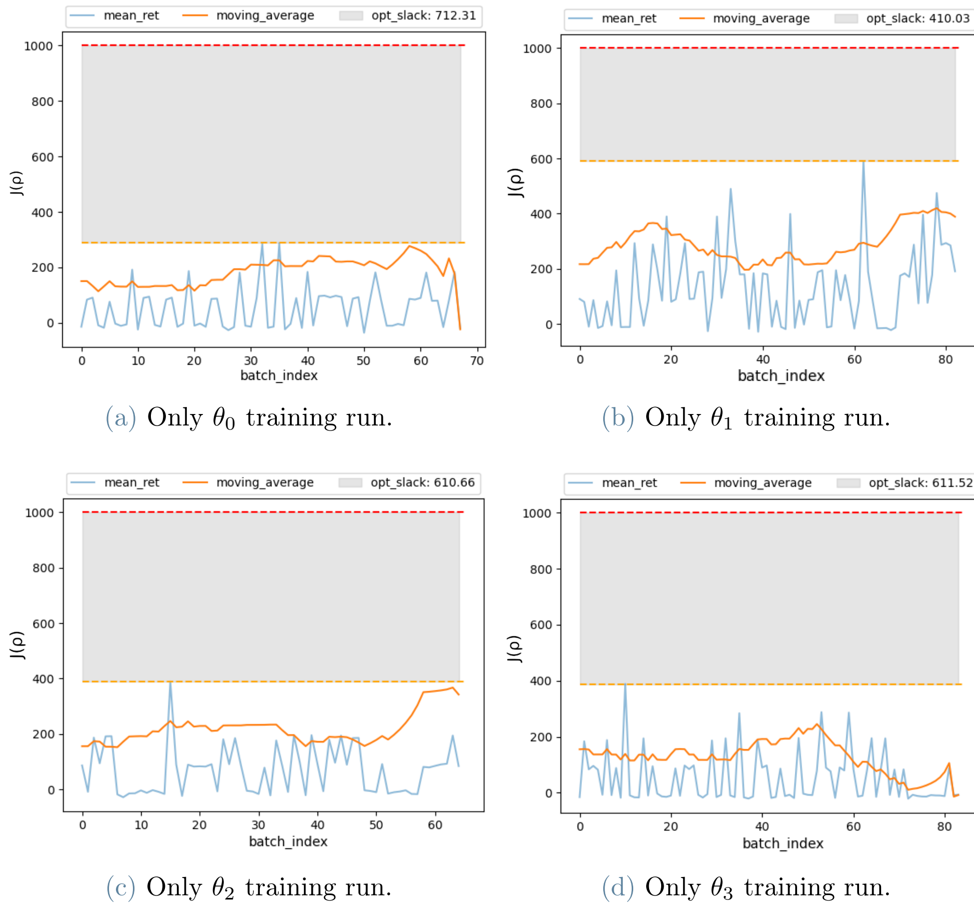
where B and P are constant values that are used, respectively, to provide the agent with a high reward if the task is successful or a high penalty in case of failure. In the performed experiments B was equal to 1000, while P was equal to 100.

Unfortunately this simple reward function led to unsuccesful trains, an example can be seen in figure 5.1, where on the horizontal axes there are indexes of each batch, which represents an iteration of the PGPE algorithm (2.1), while on the vertical axes $J(\rho)$ is the value defined in equation. 2.8. The plot shows how, while aiming at not violating the constraints, the agent does not improve the overall performances, keeping to collect high penalties for not completing the task.

The following results are referred to training runs of the *hit* task.

As explained in the previous sections, the developed rule-based policy is parametrized by multiple values that had to be learned, the θ_i. In a training run, all these values are learned at the same time. To further inspect the behaviour of the agent, different training runs on a single θ parameter were launched. In these training runs the algorithm had to learn only one parameter, while the others were fixed. The results of the training runs can be seen in figure 5.2, where *opt_slack* represents the difference between the

best value reached in the training run and the ideal one, with no penalties, while the *moving_average* is the average of the return values computed on a moving window of 20 items. Despite a general improvement in the moving average, the reward was still too noisy, the agent was not learning. Moreover a very high sensibility to the parameter arose, even a small variation of the the parameter's value could lead to a great variation of performances.



(a) Only θ_0 training run.

(b) Only θ_1 training run.

(c) Only θ_2 training run.

(d) Only θ_3 training run.

Figure 5.2: Train of the hit task on a single θ. While training a parameter the others are fixed. The gray area represents the slack between the maximum value reached during the train and the ideal result, without violations or fails.

5.2. Gradient analysis

The parameters θ of the rule-based policy are sampled, during the trains, from a gaussian distribution, with its mean and variance. To further inspect the reasons behind the ineffectiveness of the previously described trains, an analysis of the gradient updates was performed. A training run was launched, on a single parameter θ, saving all the updates of

the gradients relative to the mean and the standard deviation of the gaussian distribution from which θ was sampled. The results of this experiments are visibile in figure 5.3.

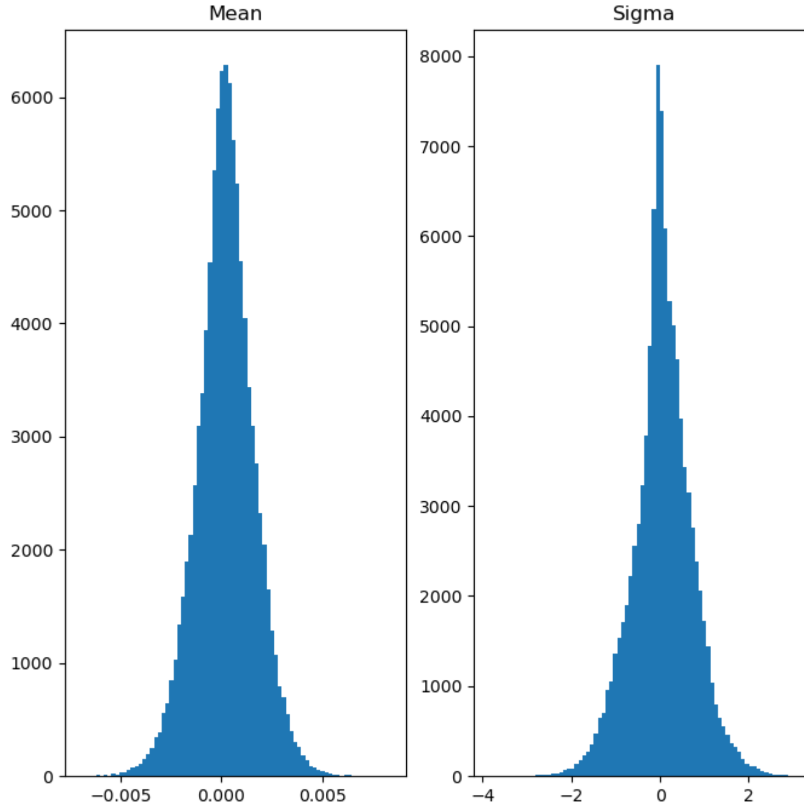


Figure 5.3: Distribution of the gradient updates. The updates are related to the mean and the standard deviation (sigma) of the gaussian distribution from which the trained parameter θ was sampled. On the horizontal axis there are the values of the updates while on the vertical one, the number of updates with that specific value.

As can be observed in the plots, the updates distribution tends to be symmetric, with slightly more positive updates. This behaviour could arise from two sources:

- *Local optima*: since an initialization of the parameters θ is necessary, it might happen that it was initialized close to a local optima, therefore the policy might get stuck in it, not understanding which gradient direction is the best one to follow;
- *Inefficient reward function*: the designed reward function might be too *sparse*, it means that the reward signal is unfrequent and the agent does not receive enough feedbacks in the decision-making process.

To check if the noise in the return arose from a suboptimal setting of θ parameters, an additional experiment was performed, deliberately initializing the parameters with

inaccurate values. Various versions of this experiment have been made, testing different learning rates. Since the task under analysis was the *hit* one, at each episode the puck was initialized in a random position and with a random, small, initial velocity. To allow a fair comparison among the experiments, the initial conditions were forced to be the same, testing each possibility: initial position fixed, initial velocity fixed, both initial position and velocity fixed.

None of these experiments showed a reduction in the oscillation of the return; therefore the necessity of developing a new, less sparse, reward function, arose naturally.

5.3. Training results

The reward function described in section 4.2.1 is the result of the ri-elaboration of the function described in equation 5.1. The focus of the reward was changed, from penalizing the constraint violation, to promoting the correct execution of the *hit* task. This change led to a new reward function way more dense than the previous one, the agent received frequent feedbacks, allowing it to find new optimal parameters during the train while increasing the average return. Moreover, a small learning rate was used, in order to mitigate the high sensibility of the policy to the parameters values. Figure 5.4 and figure 5.5, show the learning curve and the the variations of the mean and variance of the gaussian distributions of the hyperpolicy.

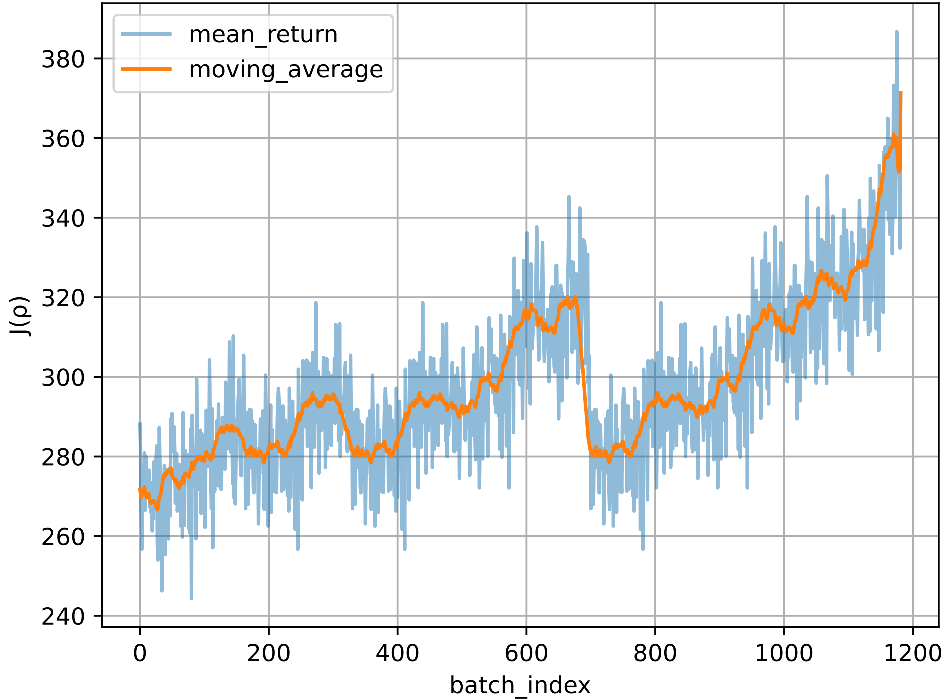


Figure 5.4: Learning curve of *hit* task with dense reward function. Agent trained with discount factor $\gamma = 0.997$ and learning_rate $= 10^{-4}$.

In particular, in figure 5.4, around batch 700 a drop can be observed, this can be related to the fact that the agent was in a local minima, after getting out of it the performances reduced but, over time, the overall return increased. This positive results is reinforced by figure 5.5 which shows a coherent direction in the updates of mean and variance of the four theta distributions. Despite the overall increase of the success rate, a reduction of the violated constraints was not observed.

5.4. Challenge outcome

In this section the outcomes of each phase will be reported, showing the results coming from the server evaluation. This outcomes are also consultable in the *leaderboard* section of the challenge official website.

The results of the hit training run described in section 5.3 were integrated in the final hierarchical agent used in the tournament phase. The final agent was not completely rule-based but incorporated also a different approach, trying to train the agent without violating constraints, however this approach won't be further analyzed in this thesis. This agent, orchestrated by the switcher and the finite state machine, was composed of four

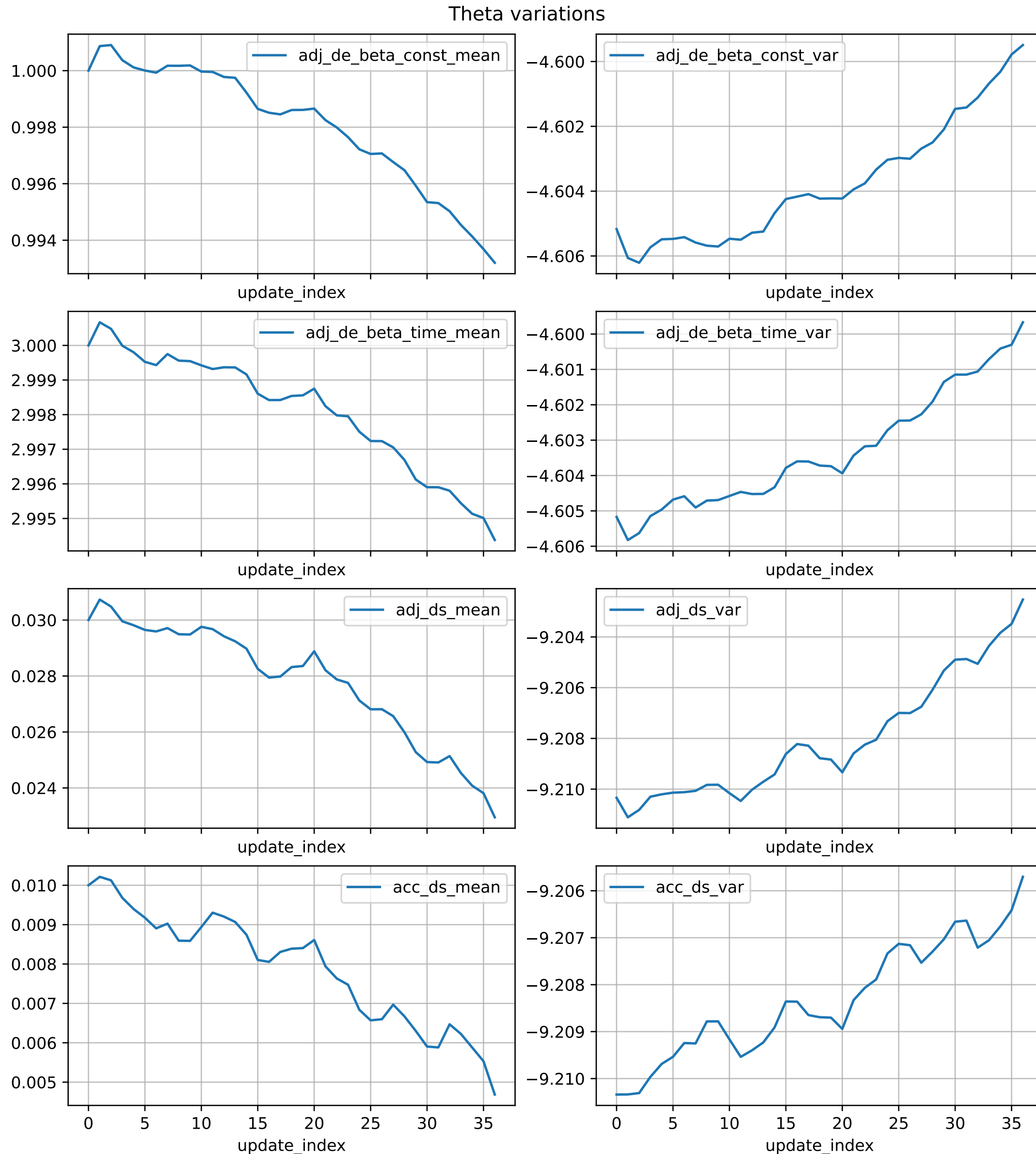


Figure 5.5: Updates of the mean and the variance of the gaussian distributions of the hyperpolicy, from which θ values are sampled. The labels of the plots represent the names of each parameter. Each row is associated to a θ, the common x-axis represents the index of each update of the parameters. On the y-axis of the variance plots, the values are expressed by means of the $\log(\sigma)$ to avoid excessively small numbers, therefore the final variance is e^{σ}.

diffefrent sub-agents, in particular:

- *Hit agent*: trained with the described rule-based policies;
- *Defend agent*: trained by means of SAC, described in section 2.2.4, combined with an algorithm that avoided constraints violations called ATACOM [13], further described in Chapter 6;
- *Prepare agent*: based on the rule-based policy described in 4.2.3;
- *Home agent*: like the defend agent, trained exploiting ATACOM.

Warm-up

In this phase only the hit and defend tasks were tested. The developed agent showed a high success rate in the hit task, and a lower one in the defend. However the agent resulted undeployable due to an excessive amount of violated constraints (> 1500):

Hit success	Defend sucess	Max penalty points
85.0 %	12.2 %	4402.0

Table 5.1: Warm-up 3DoF planar robot, server evaluation.

The max penalty points are computed as the maximum value among the ones associated to each task, in particular for this phase they were 4402.0 in the hit task and 3096.0 in the defend.

Qualifying

In the qualifying stage, the prepare task was added in the task pool. The evaluated agent exploited rule-based approach to perform hit and prepare while the defend relied on the ATACOM approach.

Hit success	Defend sucess	Prepare success	Max penalty points
22.7 %	61.6 %	36.2 %	920.0

Table 5.2: Qualifying 7DoF Iiwa robot, server evaluation.

As in the warm-up phase, the maximum penalty points are computed as the maximum penalty among the three tasks.

Hit Penalty	Defend penalty	Prepare penalty
920.0	1.5	329.0

Table 5.3: Qualifying penalty points in each of the three phases.

As can be seen, the hit task was the one generating the highest amount of penalties, this was also the reason why a refactoring of the return function of the hit policy was performed.

Tournament

In the final phase, the approaches described in previous chapters were gradually developed and integrated in the agent, showing an overall increase of the performances, which resulted in a decrement of the Penalty Points (PP).

In the beginning, the agent contained only the Switcher, without the Finite State Machine (FSM).

Constraint	Joint velocity	Joint Position	End-effector	Link	Total	PP
Violations	22	14	7	1	44	81

Table 5.4: Local evaluation of the agent, implementing only the Switcher.

Successively the FSM was integrated, significantly reducing the amount of violated constraints:

Constraint	Joint velocity	Joint position	End-effector	Link	Total	PP
Violations	13	3	9	1	26	49

Table 5.5: Local evaluation of the agent, implementing the Switcher and the FSM.

Up to this point, the agent was still implementing a simplified *Default* task, which just moved the end effector back on the default position, following a straight line. Therefore the policy described in Section 4.2.4 was developed, leading to to an additional decrement of constraints violation.

Constraint	Joint velocity	End-effector	Computation time	Total	PP
Violations	15	5	2	22	31

Table 5.6: Local evaluation of the agent, implementing the Switcher, the FSM and the rule-based *Default* task.

In the end, the final agent was developed training the *Default* task with ATACOM. This agent showed promising performances both in local and server evaluation:

Constraint	Joint velocity	End-effector	Computation time	Total	PP
Violations	12	4	1	17	24.5

Table 5.7: Server evaluation of the final agent, implementing the Switcher, the FSM and the *Default* task trained with ATACOM.

All the described results were collected by letting the agent play a complete game (45000 episodes), against a baseline agent, provided by the organizers.

The final agent showed good results despite the presence of noise in the modified environment on the server. Moreover, during the second round competitions the agent achieved a winning rate of 50% against the other participants enrolled.

6 | Related Works

This chapter will provide an overview of previous works in the field of both Air Hockey and parametric rule-based policies, combined with Reinforcement Learning. Moreover, a previous work from Liu et al., to train an agent without violating constraints will be described as well.

Parametric rule-based policy

Nowadays, designing high-level decision making systems is a task that requires a continuous balancing between the necessity of both transparency and robustness. To be transparent, a controller should always make possible to determine why a particular decision was selected; on the other hand, a robust controller should be able to generalize to unexpected situations. This two desiderata tend to be in contrast with each other. Indeed when RL is used to train a controller to be highly robust, then the interpretability of the final results is heavily influenced by the model employed.

Likmeta et al., developed an approach combining rule-based controllers and Reinforcement Learning, exploiting the strength points of both methods [10]. In particular, the authors aim at "Preserving the safety and transparency properties of the hand-crafted rule-based controllers while enhancing them with the generalization capabilities of RL". Indeed, a *parametric rule-based policy* was designed, i.e., a rule-based controller where the rules can be provided by domain experts and are defined by a set of *parameters*, whose values are learned by means of an RL algorithm. The developed framework is general, but it was applied in the autonomous driving (AD) scenario. *Policy gradient with parameter-based exploration* [PGPE, 20] was used to train the modeled policies. In the AD scenario, the controlled vehicle can face multiple challenges like: changing lane, facing a crossroads or facing a roundabout. In each situation the agent might select multiple actions, the idea behind the work was to select the most conservative one (e.g., a deceleration is more conservative w.r.t., an acceleration), according a carefully designed *total order relationship* among actions. Since the controlled vehicle, called *ego vehicle*, might interact with multiple vehicles at the same time, the policy π_θ was modeled as a

module taking as input information about the ego vehicle and any other vehicle around it, providing as output a set of actions, one for each vehicle involved. This work, empirically demonstrated that RL, in particular PGPE, can be used as an effective learning tool to improve the performances of hand-crafted policies. However, the general applicability of the controller is still heavily tied to the configuration of the parametric rules and there is no constraints optimization, therefore there is no coping with the filling of the reality gap. Moreover the work showed an agent acting in single tasks, not in a complete driving scenario.

Air Hockey and motion planning

For what concerns previous experiments related to the air hockey field, Liu et al. [12], showed that, a real general-purpose robotic manipulator arm, can achieve performances that are close to the task-specific robots, by employing advanced optimization techniques, using two Kuka Iiwa 14. In the agent developed in [12], a high-level policy selects the most appropriate task i.e., hitting, defending etc., while a low level one provides the required trajectory. This work provided a novel trajectory optimization technique, exploiting the robot redundancy to generate high-speed motion without violating the joint's constraints. In the air hockey task, hitting is the most challenging movement. The developed approach started by planning a collision-free cartesian trajectory, using hitting and stop points. Then, a linear constrained *Quadratic Programming* (QP) was used to compute the desired joint velocities on each point of the trajectory. When the agent reached the hitting point, to optimize the hitting configuration a *Nonlinear Programming* is performed, then the maximum end-effector velocity reachable in that configuration is computed, by means of *Linear programming*. The QP step was later substituted with the so called *Anchored Quadratic Programming* (AQP), an updated version of the quadratic programming, which significantly improved the hitting performance. Since the 7DoF robot is redundant for a two dimensional task, both QP and AQP optimize the redundant velocities at every point of the trajectory but AQP minimizes the difference of the current velocities with respect to a reference one, called *anchor*. Further information about AQP can be found in [12].

Further enhancements were also developed by Kicki et al. [8]. The work proposes a learning-based approach for constrained kinodynamic planning, called *Constrained Neural Motion Planning with B-splines* (CNP-B). This framework exploits the concepts of constraint manifold framing the problem as a planning over it. Dynamics and neural planning methods are also included, generating plans able to satisfy an arbitrary set of constraints and computing them in a short constant time, namely the inference time of a neural network. This approach allows the robot to plan and replan reactively, making it suit-

able for highly dynamic environments. All the kinematics dynamics, and the safety/task constraints, are defined as a single constraint manifold. Moreover, this approach relies on the representation power of neural networks to learn a planning function, inducing an indirect encode of the manifold structure. The constraint satisfaction is framed as a manifold learning problem, such that the trajectories generated are minimized not only w.r.t. an arbitrary task, but also from the constraint manifold. By recovering the metric of the constraint manifold it is also possible to attribute different priorities to each constraint. The described approach was tested on two simulated tasks, imitating real-world scenarios: moving a heavy vertically oriented object between two tables, and executing a high-speed hit in Air Hockey. The experiments showed that the new method reached state-of-the art performances, generating accurate and precise hitting motions, outperforming the results achieved by [12].

Constrained Reinforcement Learning

While applying Reinforcement Learning techniques in the robotic field, many practical issues arise, including safety and mechanical constraints. For example, a robotic manipulator should not damage the environment around it, and should not take actions that are over its feasible range. While applying RL techniques in this context, one can talk about *Constrained RL*: a subfield of Reinforcement Learning where the learning process is subject to certain constraints, coming from various needs (safety measures, ethical concerns, specific performance criteria etc.). In particular, the so called *Safe Exploration*, requires that both physical and safety constraints are satisfied throughout the whole learning process, as shown in [6]. The work of Liu et al., [13], proposed a novel method to deal with constrained RL, by *Acting on the Tangent Space of the Constraint Manifold (ATACOM)*. This method converts the constrained RL problem to a standard unconstrained one, it can deal with both equality and inequality constraints, and does not require an initial safe policy, allowing the agent to learn from scratch. Furthermore, it does not require any shielding or backup policy, bringing the agent back to a safe region. ATACOM can be used on any model-free RL algorithm independently from the policy type (deterministic or stochastic). Finally, it can increase the learning performances while coping with equality constraints, since it is able to focus the exploration on the lower-level manifold. However, its application in real-world scenarios resulted unfeasible, due to its high sensibility to model errors and sensors noises, which can cause sudden constraint violations. To mitigate this downsides it requires differentiable constraint functions and an accurate model of the robot, which can be integrated by an efficient tracking controller. In [13], ATACOM was tested with three tasks *Circle Moving*, *Planar Air Hockey* and *Iiwa Air Hockey*

(whose results are described in [12]). Five state-of-the-art model-free algorithms were used in each environment (PPO [18], TRPO [17], DDPG [11], TD3 [5], SAC [7]) showing that they could maintain the constraints below a threshold while efficiently learning the policy.

7 | Conclusions

In this work it has been shown how closing the reality gap between the simulation and the real world, represents a challenging task while coping with highly dynamic environments. The Air Hockey challenge arose in the context of developing and testing solutions capable of planning and reacting to fast environmental changes. This thesis showed how a general purpose robot, in this particular case, a 7DoF manipulator, can be used to successfully execute a specific task, playing Air Hockey, if appropriately trained. To provide an understandable but also capable of generalizing policy, a combination of rule-based policy and reinforcement learning was employed. The developed policy was trained by means of PGPE, successfully increasing the overall performances of the agent, however, without reducing the total amount of violated constraints. Finally a hierarchical agent, was developed to enable the robot to play a complete game. This agent was composed by multiple low level agents, responsible of executing a single task of the Air Hockey game: hitting, defending, preparing and going to a default position. The agent showed the ability of playing and winning a full game against a baseline agent provided by the organizers, moreover it ranked fifth out of a total of forty-six participants enrolled in the challenge.

7.1. Future Works

One of the main challenges the agent had to deal with, was embodied by the constraint violations. Both the task-specific and the robot-specific constraints represented an important obstacle in the learning process. As described in Chapter 6, the use of an algorithm thought to allow the agent a *safe exploration*, like ATACOM for example, might significantly improve the performances, allowing learning the policy without taking actions that might violate the constraints. Moreover, in this thesis the switcher described in Section 4.3.1 was built upon a set of parameters used to decide the next action that the agent had to perform. This parameters were set manually, therefore, they could be trained like other tasks, to find the best configuration of parameters. Finally, in addition to the development of better policies for the specific tasks, a constrained version of PGPE

could be elaborated. This version might work in such a way that the agent could explore freely, violating constraints in the simulated environment, but will develop a policy that, with some theoretical guarantees, will not violate them in a real world-scenario, keeping an average cost below a threshold (e.g., joint position constraint violated every ten episodes).

Bibliography

[1] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[2] R. I. Brafman and M. Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3(Oct): 213–231, 2002.

[3] K. Ciosek, Q. Vuong, R. Loftin, and K. Hofmann. Better exploration with optimistic actor critic. *Advances in Neural Information Processing Systems*, 32, 2019.

[4] C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters. Mushroomrl: Simplifying reinforcement learning research. *The Journal of Machine Learning Research*, 22(1):5867–5871, 2021.

[5] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[6] J. Garcıa and F. Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018. URL `http://arxiv.org/abs/1801.01290`.

[8] P. Kicki, P. Liu, D. Tateo, H. Bou-Ammar, K. Walas, P. Skrzypczyński, and J. Peters. Fast kinodynamic planning on the constraint manifold with deep neural networks. *arXiv preprint arXiv:2301.04330*, 2023.

[9] Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[10] A. Likmeta, A. M. Metelli, A. Tirinzoni, R. Giol, M. Restelli, and D. Romano. Combining reinforcement learning with rule-based controllers for transparent and general decision-making in autonomous driving. *Robotics and Au-*

tonomous Systems, 131:103568, 2020. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2020.103568. URL `https://www.sciencedirect.com/science/article/pii/S0921889020304085`.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2019.

[12] P. Liu, D. Tateo, H. Bou-Ammar, and J. Peters. Efficient and reactive planning for high speed robot air hockey. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 586–593. IEEE, 2021.

[13] P. Liu, D. Tateo, H. B. Ammar, and J. Peters. Robot reinforcement learning on the constraint manifold. In *Conference on Robot Learning*, pages 1357–1366. PMLR, 2022.

[14] M. Papini. Safe policy optimization. 2021.

[15] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–697, 2008. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2008.02.003. URL `https://www.sciencedirect.com/science/article/pii/S0893608008000701`. Robotics and Neuroscience.

[16] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[17] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[19] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Policy gradients with parameter-based exploration for control. In V. Kůrková, R. Neruda, and J. Koutník, editors, *Artificial Neural Networks - ICANN 2008*, pages 387–396, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87536-9.

[20] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

[21] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[22] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

List of Figures

2.1 The ongoing interaction between the agent and the enviroment comprises the agent's selection of an action and the subsequent response of the environment, providing a new observation (state) and a reward. 7
2.2 Graphical representation of parameter-based methods. 14

3.1 A real-world example of puck, mallet and air hockey game field. 19
3.2 3dof-robot (image taken from the website of the challenge). 21
3.3 7dof-robot, KUKA iiwa 14 LBR (image taken from the website of the challenge). 22
3.4 Challenge framework (image taken from the website of the challenge). . . . 24
3.5 Control paradigm (image taken from the website of the challenge). 25
3.6 Challenge's air hockey table, a smaller version of a standard air hockey table in which also the puck and the mallet got shrunk (image taken from the website of the challenge). 27
3.7 3DoF planar environment robot (image taken from the website of the challenge). 28
3.8 Kuka iiwa14 LBR Robot (image taken from the website of the challenge). . 29

4.1 Coordinates used to find the next desired position of the end-effector. The goal positions are highlighted in orange. The current position of the end-effector is the gray one, while the blue one is the desired position in the next step. To reach that position, a variation of the radius, ds, and the angle, $d\beta$, is necessary. This variations, happening simultaneously, will result in a curved movement. 36
4.2 Hit phases. Goal areas are highlighted in orange. The dotted line represents the ideal direction that the puck should follow in order to score a goal. . . 38

4.3 Triangle construction for assigning reward after the mallet hit the puck. Goal areas are highlighted in orange, the blue arrow represents the puck velocity. The gray triangle is the ideal area in which the puck should be to score a goal. If at the next step the puck will be outside the triangle, a negative reward will be applied. After the computation of the reward, the triangle will be recomputed with the new puck position in $t + 1$. The coordinates of the puck are the ones of its center. 39
4.4 *Prepare* areas. If the center of the puck will be inside any of the hatched areas, a *Prepare* will be performed. A *Prepare* can be performed only in the agent's side of the game field. 43
4.5 Switcher structure. The blue rectangles represent the tasks while the yellow rhombuses the checks that are performed. 46
4.6 Finite State Machine for controlling the transitions among tasks. The agent will always start a game in the *Home* state. The machine allows only for transitions that make the agent perform the same task as before or go home. In order to change task the agent has to visit the *Home* state first. . 47

5.1 Mean return of a training run of the hit task with a simple reward function, aimed at not violating constraints. The agent keeps collecting large penalties since it is not completing the task; the reward function focused on the constraints is too sparse. 50
5.2 Train of the hit task on a single θ. While training a parameter the others are fixed. The gray area represents the slack between the maximum value reached during the train and the ideal result, without violations or fails. . . 51
5.3 Distribution of the gradient updates. The updates are related to the mean and the standard deviation (sigma) of the gaussian distribution from which the trained parameter θ was sampled. On the horizontal axis there are the values of the updates while on the vertical one, the number of updates with that specific value. 52
5.4 Learning curve of *hit* task with dense reward function. Agent trained with discount factor $\gamma = 0.997$ and learning_rate $= 10^{-4}$. 54
5.5 Updates of the mean and the variance of the gaussian distributions of the hyperpolicy, from which θ values are sampled. The labels of the plots represent the names of each parameter. Each row is associated to a θ, the common x-axis represents the index of each update of the parameters. On the y-axis of the variance plots, the values are expressed by means of the $\log(\sigma)$ to avoid excessively small numbers, therefore the final variance is e^{σ}. 55

List of Tables

5.1 Warm-up 3DoF planar robot, server evaluation. 56
5.2 Qualifying 7DoF Iiwa robot, server evaluation. 56
5.3 Qualifying penalty points in each of the three phases. 57
5.4 Local evaluation of the agent, implementing only the Switcher. 57
5.5 Local evaluation of the agent, implementing the Switcher and the FSM. . . 57
5.6 Local evaluation of the agent, implementing the Switcher, the FSM and the rule-based *Default* task. 58
5.7 Server evaluation of the final agent, implementing the Switcher, the FSM and the *Default* task trained with ATACOM. 58

List of Acronyms

Acronym	Description
MDP	Markov Decision Process
RL	Reinforcement Learning
PO	Policy Optimization
FSM	Finite State Machine
DoF	Degrees Of Freedom
OAC	Optimistic Actor Critic
SAC	Soft Actor Critic
PGPE	Policy Gradient with Parameter-based Exploration
API	Application Programming Interface
FK	Forward Kinematics
IK	Inverse Kinematics
QP	Quadratic Programming
AQP	Anchored Quadratic Programming
PP	Penalty Points