UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Computer Engineering

"Model-Based Reinforcement Learning for industrial robotics applications"

RELATORE:
Prof. **Ruggero Carli**

LAUREANDO:
Niccolo' Turcato
Matr: 2021446

ANNO ACCADEMICO 2021/2022
DATA DI LAUREA: 05/09/2022

Contents

Abstract

In industrial robotics settings, *pick-and-place* tasks are amongst the most common activities that are delegated to robotic manipulators, because of the advantages they bring in terms of costs when performing highly repetitive tasks. Indeed, it is preferred to assign these tasks to a robot than to a human, for many reasons like for example reduced costs and because it allows to free man-hours for more intellectual tasks. An ideal goal would be to have a robotic system capable of handling arbitrary objects.

Some of the great challenges in this context are: (i) to guarantee the time efficiency of the task, crucial for handling costs; (ii) programming of the robot, indeed optimal pick and place of arbitrary objects is not a trivial task.

When allowed, throwing objects, instead of placing them, has the potential to improve the time-efficiency, as well as to increase the physical reachability of a robotic arm, by exploiting extrinsic dexterity. This can be called *pick-and-throw.* Unfortunately, when dealing with arbitrary objects, this approach opens a new set of problems on top of those already existing, mainly (i) how to pick the object in an optimal way for throwing; (ii) how to toss an object grasped in a certain configuration.

A recent work, Tossingbot [1], has implemented a whole *pick-and-throw* task in unstructured settings, using a trial and error approach exploiting unsupervised learning.

This approach has excellent performances in terms of time-efficiency and accuracy, but it does come at the cost of having to perform hundreds to thousands of training throws in order to learn the task even with the simplest object, a ball.

The objective of this thesis is to explore how to apply Reinforcement Learning techniques to the considered pick-and-throw task with a simple ball, in order to train the robot to reach the same performance as Tossingbot, using minimal training trials.

Mainly, we applied a recent Model-Based RL algorithm, MC-PILCO, to train a tossing policy in a simulated environment created from scratch and we validated part of the algorithm on data collected in laboratory sessions.

We show some positive and encouraging results obtained with the policy training in simulations, as well as with the real data, moreover, we exploited this particular task to highlight some interesting and crucial aspects of the algorithms itself.

Abstract

Nel contesto della robotica industriale, le mansioni del tipo *pick-and-place* sono tra le più comuni ad essere delegate a bracci manipolatori, poiché questi sono molto efficienti e adatti ai lavori ripetitivi.

In questi casi, delegare mansioni ai robot presenta diversi vantaggi, tra i quali il risparmio di costo e di *tempo uomo*. Inoltre, liberare le persone dai compiti più pesanti e ripetitivi permette a quest'ultimi di concentrarsi su incarichi intellettualmenete più rilevanti.

Un ostacolo ancora non del tutto superato, in questo ambito, è sicuramente la gestione di oggetti arbitrari, dove le sfide più importanti sono: (i) garantire l'efficienza in termini di tempo della mansione; (ii) la programmazione del robot.

Quando possibile, un modo per aumentare l'efficienza temporale dell'attività di *pick-and-place* è lanciare gli oggetti invece di riporli, in questo caso si può parlare di *pick-and-throw*. Ciò consente anche di allargare il confine dello spazio raggiungibile del robot, sfruttando la sua cosiddetta manualità estrinseca.

Purtroppo, quando gli oggetti manipolati sono arbitrari, questo approccio aggiunge nuovi problemi a quelli già esistenti, in particolare: (i) come raccogliere un oggetto con l'orientamento corretto per poi lanciarlo; (ii) come lanciare un oggetto afferrato in una certa configurazione.

In un articolo di recente pubblicazione, viene presentata l'implementazione di una mansione di *pick-and-throw* completa, in ambiente non strutturato, utilizzando un approccio *trial and error* che fa uso di unsupervised learning, questo sistema è stato chiamato Tossingbot [1].

Tossingbot ha prestazioni eccellenti sia in termini di efficienza temporale che di accuratezza. Tali risultati, tuttavia, si ottengono al costo di dover effettuare fino a migliaia di prove per imparare l'attività, questo anche considerando l'oggetto manipolato più semplice, ovvero una pallina.

L'obiettivo di questa tesi è di indagare come si può applicare il Reinforcement Learning, per insegnare una mansione di *pick-and-throw* a un braccio manipolatore in maniera efficiente.

In particolare, si vuole verificare se è possible raggiungere le stesse performance di Tossingbot nel lancio di una pallina, utilizzando un numero minimo di lanci di allenamento.

Nello specifico, è stato utilizzato un recente algoritmo di Model-Based RL, MC-PILCO per allenare una policy di lancio in un ambiente simulato, in più, parte dell'algoritmo è stato validato con dati raccolti in sessioni di laboratorio.

Sono stati ottenuti risultati positivi ed incoraggianti sia nell'ambiente simulato che con dati reali, oltre a ciò è stato possibile sfruttare la particolare geometria della mansione per evidenziare alcuni aspetti peculiari dell'algoritmo stesso.

Introduction

Pick-and-place tasks are amongst the most common kind of operations assigned to robotics manipulators in industrial settings.

They are activities of crucial importance and repeatable nature that in many sites and cannot be eliminated.

For economic reasons, time reduction of such task is an important aspect to consider. Moreover, another crucial aspect to reduce is the involvement of human operators in the manipulator task, which is in general undesirable.

Industrial robots, due to their great speed, precision and cost-effectiveness in repetitive tasks, tend to be used instead of manual workers in automated production lines, but these powerful machines require much more preliminary work, like calibration [2, 3], task and trajectory planning [4, 5, 6] in order to reach the desired objectives.

The robotics research community dedicated a lot of effort into making such tasks time and resource efficient, as well as more and more autonomous, some research addresses the time-efficient trajectory planning problems [7, 8], while other propose solutions to make autonomous the entire task [9].

When allowed, *throwing* objects has the potential to improve the time-efficiency, as well as to increase the physical reachability of a robotic arm, by exploiting object dynamics, a form of dynamic extrinsic dexterity [10].

In the case of *pick-and-place*, the ability of throwing objects would enable a robot arm to put objects rapidly into bins located outside its maximum kinematic range, which not only increases the physical domain where target boxes can be located, but also maximizes its operational efficiency.

However, precisely tossing arbitrary objects in unstructured environments is challenging because it depends on many factors related to the object properties (e.g. shape, inertia, mass, etc . . .) and its interaction with the rest of the environment (e.g. frictions, collisions, etc . . .). For example, grasping a hammer by its handle rather than its head may lead to different trajectories even with the same tossing movement of the robot. Instead, an ideal ball would perform the same trajectories when tossed in the same way, regardless of the orientation of grasping, it can however be significantly decelerated due to the effect of air friction.

A recent work, Tossingbot [1], confronted with the implementation of a whole *pick-and-throw* task in unstructured settings, namely the authors dealt with the full process of deciding how to pick an arbitrary object and how fast to throw it in order to reach an arbitrary target box.

The authors of [1] implemented an *end-to-end* unsupervised learning architecture, composed of two Deep Convolutional Neural Networks (DCNNs) and two robot primitives, in order to learn the whole task by trial and error, in an unsupervised way.

This architecture is composed of a DCNN to predict the optimal orientation to pick each observed object, a grasping primitive, a second DCNN to predict the release velocity of the picked object, in order to reach an arbitrary target location, and finally a throwing primitive.

The interested reader can refer to [1] for more details.

The proposed approach is successful and is relatively simple to implement, but presents some limitations, the most relevant of which is constituted by the number of trials required to learn the task. Indeed, the system demands to perform hundreds or thousands of trials even for the simplest object, a ball.

This task presents a number of practical problems, first of all being the design problem of defining the robot movement appropriate to enforce a Cartesian velocity to an object, then, second not in terms of importance, is the problem of releasing the grasped object at the right time and place synchronized with the tossing movement.

Relying to learning approaches is in this case a desirable approach for the resolution of such task, the reason being that many aspects of this task are quite difficult to approach with ad hoc solutions and even more difficult to describe with analytical models. In particular, there is the problem of throwing arbitrary objects with the correct velocity, applying the same velocity to different objects can result in different trajectories.

At this point, a whole set of problems related to learning opens up, what should the system learn? What should be handcrafted? What learning approach should be followed?

The authors of [1] proposed to teach the robot the release velocity to impose to the thrown objects, while the robot's tossing movement is implemented with a robot primitive (handcrafted) and chose to rely to am unsupervised learning approach that works by trial and error.

This approach is able to solve a lot of practical issues. The system just requires to see the unstructured bin of objects and to verify if the objects tossed by the robot actually fall in the correct bin or not. It is shown that by collecting loads of experience the system is able to learn a correct map from the visual information to the Cartesian velocity.

The simplicity of this kind of unsupervised learning approaches is also its curse, the key factor being the loads of experience needed to reach convergence, by just evaluating the final outcome of throwing an object the Tossingbot system is ignoring tons of information that lies in between the release and the landing: the object's trajectory.

In this thesis, we propose to teach the *pick-and-throw* task to a robotic manipulator using a Reinforcement Learning (RL) approach, in order to reach convergence to optimal performance with few training trials.

RL is an area of Machine Learning that is tasked with designing intelligent agents that take actions in an environment, in order to reach a certain goal. The agent is indeed equipped with the ability to evaluate by itself its own performance and should adapt its behavior accordingly as it progressively interacts with the environment.

The two main approaches in RL literature are Model-Free and Model-Based approaches (denoted as MB hereafter), the first kind relies on various strategies to learn the actions to perform a certain task directly from the interaction with the environment, while the second kind uses the interaction with the environment to build a dynamics Model that is able to predict the system's evolution given current state and the action performed by the agent.

MB approaches, with the right premises can be particularly effective in yielding optimal acting agents with limited actual experience with the real system, which is very desirable in contexts of sophisticated mechanical systems, of which industrial robotics is a prime example.

The *pick-and-throw* task can be thought of as a standalone activity that is very interesting by the point of view of learning, because efficient approaches require to learn the dependency of the landing position from the initial state, i.e. release position and velocity, but it can also be considered as a benchmark for industrial robotics applications.

Therefore, to show how MB RL methods could be valuable in the field of industrial robotics, we investigated how this approach could be applied to a *pick-and-throw* task, we built a minimal simulated environment with the Gazebo software [11] and applied a very recent algorithm for learning an acting control policy in a very efficient way: MC-PILCO [12].

MC-PILCO is a Model-Based policy gradient algorithm. In partivular, which exploits Gaussian Processes (GPs) to learn a dynamics model of the system and a Monte Carlo approach to estimate the policy gradient, the GPs can make use of specialized kernel functions if some knowledge of the system is available, which can lead to improved data-efficiency of the model. Moreover, any differentiable function can be used as policy, although simple, interpretable functions are recommended, the authors of [12] provided a simple, interpretable general-purpose policy function.

The robotic manipulator chosen for the task is Franka Emika's Panda robot, a collaborative manipulator for which all libraries and software for simulation is available, moreover an actual real Panda Robot was available to some experiments in the laboratory of the Department of Information Engineering of the University of Padova.

We developed a simple and efficient trajectory generation that allowed to implement a low accuracy, high precision tossing system, which, with opportune noise modeling, is more than acceptable for the desired task.

We performed a series of simulated trials in order to validate the performance of the algorithms in terms of system evolution prediction and system control, as well as to show the exploratory features of said algorithm, which can be interpreted due to the characteristics of the considered task.

The results show that the dynamics model presented in [12], with the needed adaptations, is capable of learning to predict the trajectories and landing locations of the tossed ball, provided that the nominal initial positions and velocities are sufficiently accurate, moreover the general-purpose policy from [12] was successful in learning a map from target location to input release velocity.

In the experiments we were able to show the ability of [12]'s policy gradient algorithm in exploring the policy's parameters space, thus increasing the probability of the training to yield an optimal policy.

Moreover, it was possible to gather some trajectory data of test ball in laboratory using 3D cameras and thus validate the Model of [12] on real data. In addition, a second trajectory generation was developed and tested for use on the real robot.

The experiments presented in this thesis should be considered preliminary to further research that explores learning how to throw different objects.

The policy learning test performed with the simulation data show that, when the model is precise enough, MC-PILCO is able to learn the long-term dependencies of the system evolution from the manipulator action.

Moreover, Model Learning experiments performed with the data collected in the laboratory sessions show promising results that encourage to follow this line of research.

This document is organized as follows:

- Chapter 1 gives a brief introduction on Model-Based methods for Reinforcement learning with particular attention to MC-PILCO;
- Chapter 2 describes in a formal way the task and present the proposed solution based on MC-PILCO, as well as all the experiments that have been performed.
- Chapter 3 will show and discuss all results obtained with the various experiments in simulations, demonstrating that with MC-PILCO it is indeed possible to learn industrial robotics tasks. Moreover, some encouraging experiments performed with real data of ball trajectories collected in laboratory will be discussed.

Chapter 1

Model-Based Reinforcement Learning

This chapter will give a brief introduction on *Model-Based* (MB) methods for Reinforcement learning (RL), also called *planning* or *indirect* methods. All content described in this chapter is preparatory to comprehend the work presented in chapter 2 and chapter 3, for deeper explanations refer to [13].

RL is an area of Machine Learning that is concerned with building intelligent agents that are required to take actions in an environment, in order to minimize the notion of cumulative cost. The key ingredient of the RL paradigm is that the behavior of the agent is defined and evolves as it progressively interacts with the environment, until it reaches *convergence* when an optimal behavior is found. The most important trait that differs RL from other machine learning paradigms (e.g. supervised learning) is the autonomy of the learning process. All that is needed in RL to train an agent is the environment interaction and a cost function to evaluate states.

In the RL literature, the environments are typically described by the *Markon Decision Process* (MDP), which is defined by the tuple $\langle \mathbf{S}, \mathbf{A}, \mathbf{P}, \mathbf{C}, \boldsymbol{\gamma} \rangle$:

- $\mathbf{S}$: a finite set of states;
- $\mathbf{A}$: a finite set of actions;
- $\mathbf{P}$: a state transition probability matrix/function with entries:

$$\mathcal{P}^a_{ss'} = \mathcal{P}[S_{t+1} = s' | S_t = s, A_t = a] \tag{1.1}$$

- $\mathbf{C}$: cost function, $C^a_s = \mathbb{E}[C_{t+1} | S_t = s, A_t = a]$, which returns the immediate cost in that specific state;
- $\boldsymbol{\gamma}$: discount factor, $\gamma \in [0, 1]$.

The MDP formulation of an environment is useful to use most of the tools in the literature [13], when the system is known and, in particular, when the system is finite. Most of the real world scenarios, like the ones considered in this work, are not finite and only partially observable, nonetheless the MDP formalism is usually retained even when dealing with complex systems.

The behavior of the agent is described by a policy function:

$$\begin{gathered}\pi : A \times S \rightarrow [0,1] \\ \pi(a,s) = \mathcal{P}(a_t = a, s_t = s)\end{gathered} \tag{1.2}$$

The policy defines the probability of the agent taking action a when in state s.

The goal of the RL paradigm is to yield a policy π such that its expected long term return G_t is minimized at each state:

$$G_t = C_{t+1} + \gamma C_{t+2} + \gamma^2 C_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k C_{t+k+1} \tag{1.3}$$

The long term value of a state, given policy π, is defined by the *value function*:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \sum_{a \in \mathbf{A}} \pi(a|s)(C_s^a + \gamma \sum_{s' \in \mathbf{S}} \mathbf{P}_{ss'}^a v_\pi(s')) \tag{1.4}$$

where the rightmost member of the equation is called *Bellman Expectation Equation.* If the policy is *deterministic*, at each state only one action is selected with unit probability, the formalism of π can be changed so that $\pi(s) = a$ defines the action selected by π. The *Bellman Expectation Equation* can then be rewritten:

$$v_\pi(s) = C_s^{\pi(s)} + \gamma \sum_{s' \in \mathbf{S}} \mathbf{P}_{ss'}^{\pi(s)} v_\pi(s') \tag{1.5}$$

MB RL methods follow the same philosophy of all RL methods, but exploit models of the environment built using the collected experience to yield an acting agent. By a model of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state. If the model is stochastic, then there are several possible next states, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities, i.e. a distribution of states.

The main reason of having a model of the environment is that the agent is able to *explore* the environment without actually needing a physical interaction, or reducing the interaction time needed to reach convergence. This can be really useful in contexts where wear and tear of the involved components are important aspects to consider or when system interaction is costly. This is especially true when dealing with sophisticated mechanical systems.

Indeed, RL techniques that do not build models of the system, called *Model-Free* (MF) or *direct* methods, are very easy to deploy and have very good performances on limited and constrained environments, but can require large interaction time on more complex systems.

The possible relationships between experience, model, values, and policy are summarized in fig. 1.1. Each arrow shows a relationship of influence and presumed improvement between different phases of the RL process.

Model-Free reinforcement learning paradigms directly build value functions and/or policies with the collected experience, they can be represented by the smaller loop between **experience** and **value/policy**. Instead **planning** methods follow the external loop in fig. 1.1: a **model** of the environment is built with the **experience** collected interacting with the system (*model learning*), the learned model is then exploited to build a **policy** directly or through a **value** function (*planning*), finally the resulting policy is used to interact with the system (*acting*), generating new **experience**.

Depending on the peculiarities of the system, like its dynamics and the complexity of the action to perform, MB approaches can lead to increased data efficiency with respect to MF methods, requiring less exploration to reach convergence. The biggest downside of planning methods, is the complexity and the difficulty of deploying, as the overall architecture is more complex and requires additional components, w.r.t. direct approaches. Moreover, indirect methods have a not null risk of introducing biases in the design of the model, which is not a problem in direct methods.

For this reason, the use of a model to learn an acting policy is a double-edged sword, it can lead to an optimal policy very efficiently, possibly needing limited experience with the system, therefore resulting in low wear of the environment, but a model may never reach a good enough approximation of the system's dynamics, so a policy that converges to optimality following a faulty model might very well be sub-optimal w.r.t. the real world.

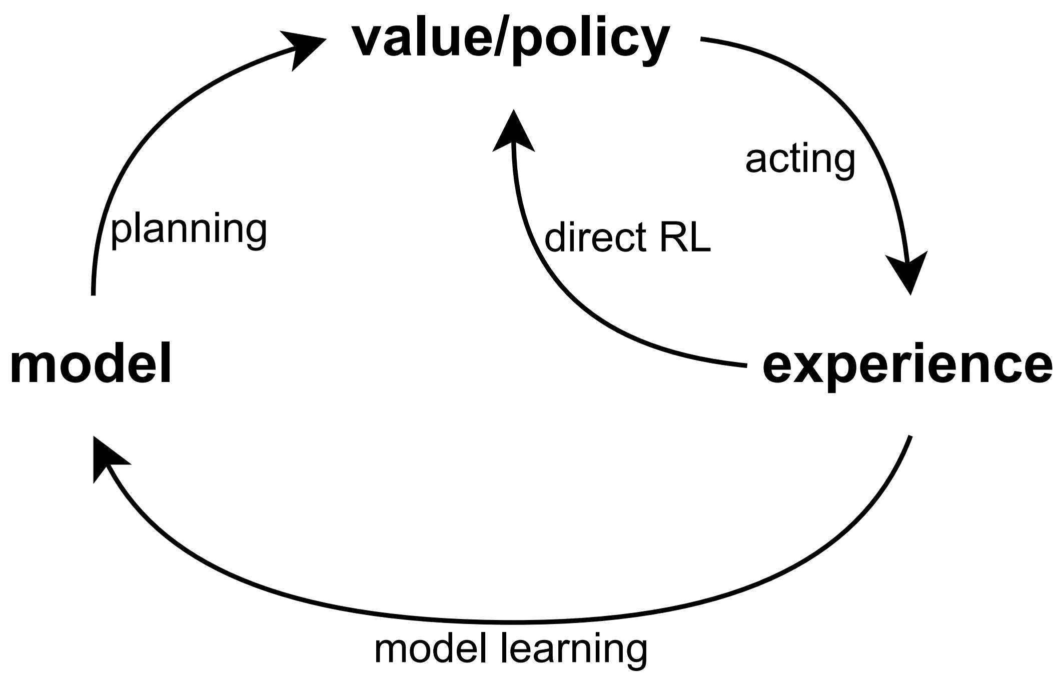

Figure 1.1: Relationships among learning, planning, and acting (from [13]).

1.1 Gradient-Based Policy Search with GPs

One popular Reinforcement Learning approach to learn an acting policy is *Policy gradient* or *Gradient-based policy search*, as the name implies it consists in learning a policy defined by some parameters θ by gradient descent over a cost function. This approach requires to compute a cost estimation that depends on the policy parameters, which is a non trivial problem.

The use of Model-Based approaches for learning the control of mechanical systems is very appealing in industrial robotics applications, where it is desirable to learn how to control a mechanical system, rather than designing ad hoc control strategies with complex mathematical tools that might present strong biases and/or limiting approximations. Moreover, typical direct learning approaches are not favored in these contexts, given the aforementioned poor data efficiency issues.

PILCO (Probabilistic Inference for Learning COntrol) [14] is a successful MB RL algorithm that uses GP models (details in appendix A) and gradient-based policy search to achieve substantial data efficiency in solving different control problems.

In PILCO, long-term predictions are computed analytically, approximating the distribution of the next state at each time instant with a Gaussian distribution by means of moment matching. In this way, the policy gradient is computed in closed form.

However, the use of moment matching introduces also two relevant limitations:

- Moment matching allows modeling of only unimodal distributions. This introduces relevant limitations related to the initial condition, as well as being potentially wrong.
- The computation of the moments is shown to be tractable only when considering Squared Exponential (SE) kernels and differentiable cost functions. Limiting the kernel choice might be very stringent depending on the system, it has been shown repeatedly that specialized kernel functions can lead to substantive improvements in data efficiency and generalization properties [15, 16, 17, 18, 19].

Several other MB RL algorithms have extended PILCO to overcome these limitations, Deep-PILCO [20] has explored how to overcome the use of SE kernels, which uses Bayesian Neural Networks [21, 22] to model system evolution. Results show that, compared to PILCO, Deep-PILCO requires larger number of iterations with the system to learn the task. This can be explained by the fact that NNs typically require large sets of data for convergence to good performance, therefore might not be the best choice for these applications. Another approach was presented in PETS [23]. which uses an ensemble of NNs to model uncertainty in the estimation of system dynamics. Again results show worse data efficiency when compared to PILCO for control of low-dimensional systems (e.g. cartpole [14, 24]), while presenting positive results in simulated high-dimension systems.

One method to bootstrap Model Learning of real mechanical systems in PILCO consists in the use of simulated data to learn a prior for the GP model before starting the RL procedure on the actual system [25]. This approach shows important improvements in the performance in unexplored configurations, however, accurate simulators are required in these settings, which might not be always available.

Other approaches addresses the limitations of gradient-based optimization, adopting gradient-free policy optimization [26], while others focused on overcoming approximations due to moment matching relying on particle-based methods [27, 28].

Promising results with particle-based methods were obtained exploiting the *reparameterization trick* [29] which has shown successful results in stochastic variational inference SVI [28, 30].

In this thesis, a recent MB RL algorithm was exploited for learning control of a mechanical system in the context of industrial robotics, namely: *Monte Carlo Probabilistic Inference for Learning COntrol* (MC-PILCO) [12].

MC-PILCO, like PILCO is a (MB) policy gradient algorithm, in which GPs are used to estimate one-step-ahead system dynamics and long-term state distribution are approximated with a particle-based method, instead of moment matching.

The reparameterization trick is exploited in the computation of the gradient of the expected cumulative cost with backpropagation [31]. The optimization problem is interpreted as a *stochastic gradient descend* problem (SGD) [32], and *dropout* [33] of the policy parameters is used in order to raise the chances to escape from local minima, obtaining performing policies. Results in [12] show that MC-PILCO outperforms its predecessor PICLO and other similar extensions of the same.

1.1.1 Model-based policy gradient

Consider the discrete-time system described by the unknown transition function $f(\cdot,\cdot)$:

$$x_{\tau+1} = f(x_\tau, u_\tau) + w_\tau \tag{1.6}$$

where $x_\tau \in \mathbb{R}^{d_x}$ and $u_\tau \in \mathbb{R}^{d_u}$ are respectively the state and control inputs of the system at each step τ, while $w_\tau \sim \mathcal{N}(0, \Sigma_w)$ is an independent Gaussian random variable modelling additive system noise. In this context, the *continuous* state x_τ corresponds to the *finite* state $s \in \mathbf{S}$ in the MDP formalism, while the *continuous* inputs u_τ correspond to the *finite* actions $a \in \mathbf{A}$.

The cost function in this formulation $c(x_\tau)$ defines the immediate penalty for lying in state x_τ, it does not depend on the control inputs forwarded at current time. The inputs forwarded to the system are chosen following a policy $\pi_\theta : x \to u$ that depends on the parameters θ. The objective is to yield a policy that minimizes the expected cumulative cost over finite number of steps T, namely:

$$J(\theta) = \sum_{\tau=0}^{T} \mathbb{E}[c(x_\tau)] \tag{1.7}$$

where the initial state x_0 is selected according to a given probability $p(x_0)$.

A planning approach for learning a policy consists, in general, of the succession of several trials; i.e. attempts to solve the desired task. As preliminary step, an initial policy, possibly random, is used to collect exploration data used to bootstrap the model of the system dynamics. Each trial consists of three main phases:

- *Model Learning*: data collected from all previous interactions are used to build a model of the system dynamics;
- *Policy Update*: the policy is optimized in order to minimize the cost $J(\theta)$ according to the current model;
- *Policy Execution*: the current optimized policy is applied to the system and the data are stored for model improvement.

The model of the system dynamics is used to predict the system evolution under the execution of the current policy π_θ, the predicted evolution is used to compute an estimation of $J(\theta)$ and its gradient $\nabla_\theta J(\theta)$, in order to update θ following gradient-descent approach.

1.1.2 GPR and one-step-ahead predictions

A common strategy with GPR-based approaches for one-step-ahead predictions consists in estimating the evolution of each state dimension with separate GPs, hereafter denoted as *full-state* model. Consider:

$$\Delta_\tau^{(i)} = x_{\tau+1}^{(i)} - x_\tau^{(i)}, \;\; i \in \{1, \ldots, d_x\} \tag{1.8}$$

being the increment between the value of the i-th component of the states a time $t+1$ and t. Define:

$$y_\tau^{(i)} = \Delta_\tau^{(i)} + e^{(i)} \tag{1.9}$$

as the noisy observation of $\Delta_t^{(i)}$, with $e^{(i)}$ modelling additive noise. Moreover, let:

$$\tilde{x}_\tau = \begin{bmatrix} x_\tau \\ u_\tau \end{bmatrix} \tag{1.10}$$

be the vector that summarize both the state and the input applied at time t, which is the general GP input. Then, one can define a dataset:

$$\begin{aligned} \mathcal{D} &= (\tilde{\boldsymbol{X}}, \boldsymbol{y}^{(i)}) \\ \boldsymbol{y}^{(i)} &= [y_{\tau_1}^{(i)}, \ldots, y_{\tau_n}^{(i)}]^T \\ \tilde{\boldsymbol{X}} &= \{\tilde{x}_{\tau_1}, \ldots, \tilde{x}_{\tau_n}\} \end{aligned} \tag{1.11}$$

where $\boldsymbol{y}^{(i)}$ is the vector of measurements, and $\tilde{\boldsymbol{X}}$ is the set of GP inputs. For each state dimension, the GPR assumes the following probabilistic model:

$$\boldsymbol{y}^{(i)} = \begin{bmatrix} h^{(i)}(\tilde{x}_{\tau_1}) \\ \vdots \\ h^{(i)}(\tilde{x}_{\tau_n}) \end{bmatrix} + \begin{bmatrix} e^{(i)}_{\tau_1} \\ \vdots \\ e^{(i)}_{\tau_n} \end{bmatrix} = \mathbf{h}^{(i)}(\tilde{\boldsymbol{X}}) + \mathbf{e}^{(i)} \tag{1.12}$$

where the additive noise is usually modeled as zero-mean Gaussian: $\mathbf{e}^{(i)} \sim \mathcal{N}(0, \sigma_i I)$, and $h^{(i)}(\cdot)$ is the unknown function:

$$h^{(i)} : \tilde{x} \to \Delta^{(i)}, \;\; i \in \{1, \ldots, d_x\} \tag{1.13}$$

that describes the system dynamics, modeled a prori as zero-mean GP. Following the formalism defined in appendix A, the GP model:

$$\begin{aligned} h^{(i)} &\sim \mathcal{N}(0, K_i(\tilde{\boldsymbol{X}}, \tilde{\boldsymbol{X}})) \\ K_i(\tilde{\boldsymbol{X}}, \tilde{\boldsymbol{X}}) &\in \mathbb{R}^{n \times n} \end{aligned} \tag{1.14}$$

with the a priori covariance matrix $K_i(\tilde{\boldsymbol{X}}, \tilde{\boldsymbol{X}})$ defined element-wise through a certain kernel function.

One standard choice of kernel function is the *Squared Exponential* or *RBF* kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \lambda^2 \exp(-||\boldsymbol{x} - \boldsymbol{x}'||^2_{\Lambda^{-1}}) \tag{1.15}$$

which is a common choice for regression of continuous functions, more details on the RBF kernel and kernels in general in the context of GPR in appendix A.2.

The scaling factor λ and the matrix Λ are kernel hyperparameters, which can be optimized by maximization of the marginal likelihood (some insight in appendix A.2.3).

Matrix Λ is typically assumed as diagonal: $\Lambda = diag(\lambda_1, \ldots, \lambda_n)$ and the diagonal terms $\lambda_1, \ldots, \lambda_n$ are called length-scales.

As known from GPR literature, given the general GP input $\tilde{x}_\tau$ at time τ, possibly $\tilde{x}_\tau \notin \tilde{\boldsymbol{X}}$, the posterior distribution of $h^{(i)}(\tilde{x}_\tau)$ can be computed in closed form:

$$\begin{aligned} \mathbb{E}[\hat{\Delta}^{(i)}_\tau] &= \mathbb{E}[\hat{h}^{(i)}(\tilde{x}_\tau)] = k_i(\tilde{x}_\tau, \tilde{\boldsymbol{X}})\alpha^{(i)}_\tau \\ \mathbb{V}[\hat{\Delta}^{(i)}_\tau] &= \mathbb{V}[\hat{h}^{(i)}(\tilde{x}_\tau)] = k_i(\tilde{x}_\tau, \tilde{x}_\tau) - k_i(\tilde{x}_\tau, \tilde{\boldsymbol{X}})\Gamma_i^{-1} k_i^T(\tilde{x}_\tau, \tilde{\boldsymbol{X}}) \end{aligned} \tag{1.16}$$

with Γ_i, $\alpha_\tau^{(i)}$ and $k_i(\tilde{x}_\tau, \tilde{\boldsymbol{X}})$:

$$\begin{aligned}\Gamma_i &= (K_i(\tilde{\boldsymbol{X}}, \tilde{\boldsymbol{X}}) + \sigma_i^2 I) \\ \alpha_\tau^{(i)} &= \Gamma_i^{-1} \boldsymbol{y}^{(i)} \\ k_i(\tilde{x}_\tau, \tilde{\boldsymbol{X}}) &= [k_i(\tilde{x}_\tau, \tilde{x}_{t_1}), \dots, k_i(\tilde{x}_\tau, \tilde{x}_{t_n})]\end{aligned} \tag{1.17}$$

More details on the derivation of the posterior distribution in appendix A.2.2.

The evolution of each state dimension is modeled by a distinct GP, assuming that the d_x GPs are conditionally independent from each other given current input $\tilde{x}_\tau$, the posterior distribution for estimate of the state at time $\tau + 1$ is then modeled as:

$$p(\hat{x}_{\tau+1} | \tilde{x}_\tau, \mathcal{D}) \sim \mathcal{N}(\mu_{\tau+1}, \Sigma_{\tau+1}) \tag{1.18}$$

where the mean vector and covariance marix can be expressed as:

$$\begin{aligned}\mu_{\tau+1} &= x_\tau + \left[\mathbb{E}[\hat{\Delta}_\tau^{(1)}], \dots, \mathbb{E}[\hat{\Delta}_\tau^{(d_x)}]\right]^T \\ \Sigma_{\tau+1} &= diag(\mathbb{V}[\hat{\Delta}_\tau^{(1)}], \dots, \mathbb{V}[\hat{\Delta}_\tau^{(d_x)}])\end{aligned} \tag{1.19}$$

1.1.3 Long-term predictions with GP dynamical models

The parameterized policy π_θ is evaluated and improved in the policy update step of MB policy gradient (section 1.1.1) following long-term predictions of the evolution of states: $p(\hat{x}_1), \dots, p(\hat{x}_T)$.

To rigorously compute these probabilities, it is demanded to apply the one-step-ahead GPs in cascade, propagating model uncertainty.

Namely. this requires to start from an intial distribution $p(x_0)$ and at each step τ, the next state distribution is obtained by marginalization of eq. (1.18) over $p(\hat{x}_\tau)$:

$$p(\hat{x}_{\tau+1}) = \int p(\hat{x}_{\tau+1} | \hat{x}_\tau, \pi_\theta(\hat{x}_\tau), \mathcal{D}) p(\hat{x}_\tau) d\hat{x}_\tau \tag{1.20}$$

The exact computation of eq. (1.20) is not tractable in most of the interesting cases, therefore in the following we will consider two different approximation approaches: moment matching (exploited in PILCO) and the particle-based method developed for MC-PILCO by the authors of [12].

1.1.3.1 Moment matching

This requires the assumption that the GPs use only SE kernels as covariance functions, then consider a Gaussian initial state distribution:

$$x_0 \sim \mathcal{N}(\mu_0, \Sigma_0) \tag{1.21}$$

Then $p(\hat{x}_1)$ can be computed in closed form (details can be retrieved in [34]) and next it is approximated to be a Normal distribution, whose mean and variance have been analytically computed previously.

This procedure is iterated for each step of the prediction horizon, yeilding the following probability distributions: $p(\hat{x}_2), \ldots, p(\hat{x}_T)$.

Hence, it is possible to compute the policy gradient in closed form, which is a favorable property, but it comes at the costs already mentioned at the beginning of this section (1.1):

- SE kernels may lead to poor generalization properties;
- Only unimodal distributions can be modeled, which is not representative of many real systems.

1.1.3.2 Particle-based method

With this method, eq. (1.20) is approximated by means of a particle-based approximation:

- M particles x_0^m are sampled by the initial state distribution $p(x_0)$.
- Each of the M particles is propagated using the one-step-ahead GP models from eq. (1.18), i.e. $x_{\tau+1}^m$, $m \in \{1, \ldots, M\}$ is sampled from:

  $$p(x_{\tau+1}^m | x_\tau^m, \pi_\theta(x_\tau^m), \mathcal{D}) \tag{1.22}$$

  The propagation is iterated until a trajectory of T samples has been generated for each of the M particles.

The long-term prediction of eq. (1.20) is approximated at each time step $\tau + 1$ by the distribution of the particles $\{x_{\tau+1}^m,\ m = 1, \ldots, M\}$. Most importantly, this process does not have particular requirements on the choice of kernel function and the distribution of the initial state.

This approach does not suffer the most critical issues of moment matching, but the price to pay is the additional computational complexity. In particular, the computation of eq. (1.18) requires to compute eq. (1.19), which requires to compute, for each element of the state, mean and variance from the related GP model following eq. (1.16).

The computation of Γ_i and $\alpha_\tau^{(i)}$ from eq. (1.17) can be done offline, once for each GP model. While the computation of the mean of the GP target $\mathbb{E}[\hat{\Delta}_\tau^{(i)}]$ is linear in the number of training samples, the computation of the variance $\mathbb{V}[\hat{h}^{(i)}(\tilde{x}_\tau)]$ is the bottleneck of the operations, having quadratic complexity in the numer of training samples.

The cost of a single state prediction is then $O(d_x n^2)$, whereas the total computational cost of computing the evolution of all M particles if $O(d_x MTn^2)$.

Many aspects of the system dynamics and of the type of kernel function might influence the number of particles and the number of samples required to obtain a good enough approximation of the long-term system evolution, determining a remarkable computational burden as well as memory requirement.

Howsoever, GPUs with large memories are nowadays more and more common and can certainly alleviate the computational burden by running in parallel each particle's evolution.

1.2 Monte Carlo gradient estimation for MB Policy Search with GPs

This section will describe the core concepts of the MC-PILCO algorithm: GPR for model learning and Monte Carlo sampling to estimate the expected cumulative cost using the learned system model.

The algorithm follows the steps described in section 1.1.1, the following will describe in depth *model learning* and *policy update* steps.

1.2.1 Model Learning

MC-PILCO can use the one-step-ahead model described in section 1.1.2 but the authors of [12] proposed a novel prediction model, named *speed-integration*, that promises to bring advantages w.r.t. the previously described one. This subsection will describe the aforementioned model, discuss the possible kernel choices and finally describe the model's hyperparameters optimization as well as the possible reduction techniques for the learning problem.

1.2.1.1 Speed-Integration Model

This approach requires to characterize the state as:

$$x_\tau = \begin{bmatrix} q_\tau \\ \dot{q}_\tau \end{bmatrix} \tag{1.23}$$

where $q_\tau \in \mathbb{R}^{\frac{d_x}{2}}$ is the vector of generalized coordinates of the system at time τ, whose interpretation changes depending on the type of system and what it is desired to describe. Instead, $\dot{q}_\tau$ is the derivative, w.r.t. time, of q_τ.

The model exploits the natural relationship between the generalized coordinates and its derivative, indeed, when considering a small sampling time T_s, it is possible to approximate to constant accelerations between consecutive time-steps, using standard kinematic rules, it is possible to yield the following:

$$\begin{aligned} q_{\tau+1} &= q_\tau + T_s\dot{q}_\tau + \frac{T_s}{2}(\dot{q}_{\tau+1} - \dot{q}_\tau) = q_\tau + T_s\dot{q}_\tau + \frac{T_s}{2}\Delta\dot{q}_\tau \\ \dot{q}_{\tau+1} &= \dot{q}_\tau + \Delta\dot{q}_\tau \end{aligned} \tag{1.24}$$

Of course, the sampling time T_s must be well calibrated with respect to the application.

With these assumptions, the only unknown quantity in eq. (1.24) is $\Delta\dot{q}_\tau$, which are set to be the targets of the GPR, which can be learned with $\frac{d_x}{2}$ GP models.

Let denote with $\mathcal{I}_q$ and $\mathcal{I}_{\dot{q}}$ the ordered set of the dimension of q and $\dot{q}$ respectively, then it is possible to denote with:

$$\Delta\dot{q}_\tau^{(i_k)} = (\dot{q}_{\tau+1}^{(i_k)} - \dot{q}_\tau^{(i_k)}), \quad i_k \in \mathcal{I}_{\dot{q}} \tag{1.25}$$

the target of the i_k GP model, which is used to predict the evolution of the i_k velocity component, then $x_{\tau+1} = [q_{\tau+1}^T, \dot{q}_{\tau+1}^T]^T$ can be computed following eq. (1.24).

Compared to the *full-state* model described in section 1.1.2, *speed-integration* trains half as much GP models and correctly condition the predicted evolution of generalized coordinates with respect to their derivatives.

However, when a small enough sampling time is not available, *speed-integrating* might diverge from the real dynamics with high probability, but, as already stated, MC-PILCO can be used with the *full-state* modeling, which might be more effective in that case.

1.2.1.2 Kernel choice

Particle-based methods have the possibility to characterize the GP models, of any dynamical model adopted, with the appropriate kernel without restrictions, the authors of [12] presented a couple alternatives to the standard SE kernel of eq. (1.15), which will be presented as examples.

SE + Polynomial (SE + P$^{(d)}$):

$$\begin{aligned} k_{SE+P^{(d)}}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) &:= k_{SE}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) + k_p^{(d)}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) \\ k_p^{(d)}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) &:= \prod_{r=1}^{d} (\sigma_{P_r}^2 + \tilde{x}_{\tau_j}^T \Sigma_{P_r} \tilde{x}_{\tau_k}) \end{aligned} \tag{1.26}$$

The function $k_p^{(d)}$ is called Mutiplicative Polynomial (MP) kernel of degree d and is useful to capture the modes of the system that are polynomial in $\tilde{x}$, which is typically true for mechanical systems.

Semi-Parametrical (SP):

$$\begin{aligned} k_{SE+P^{(d)}}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) &:= k_{SE}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) + k_{PI}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) \\ k_{PI}(\tilde{x}_{\tau_j}, \tilde{x}_{\tau_k}) &:= \boldsymbol{\phi}^T(\tilde{x}_{\tau_j}) \Sigma_{PI} \boldsymbol{\phi}(\tilde{x}_{\tau_k}) \end{aligned} \tag{1.27}$$

The function k_{PI} allows to exploit prior knowledge in the system by defining a suitable basis function $\boldsymbol{\phi}(\tilde{x})$. k_{PI} is typically derived from basic physic principles and for this reason it is called *Physically Inspired* kernel [16].

The correctness of these formulations comes from the fact that the sum of kernels is still a kernel [35], refer to [12] for deeper explanations.

1.2.1.3 Model Optimization

Once the dynamics model is chosen (from section 1.1.2 or section 1.2.1.1 or any alternative most suitable to the considered system) the GP hyperparameters are optimized by maximizing the marginal likelihood (ML) of the training samples (see appendix A.2.3).

Various reduction techniques to lower the computational complexity and memory consumptions of the GP models are discussed in [12].

1.2.2 Policy Update

This subsection will present a general-purpose policy and describe the optimization performed in MC-PILCO as well as the application of dropout of the policy parameters.

1.2.2.1 General-Purpose Policy

MC-PILCO's policy optimization can be applied to any differentiable policy function, like a deep Neural Network, but for most applications, simpler and more interpretable functions are preferred, therefore [12] presented the following *squashed-RBF-network* policy:

$$\pi_\theta(\boldsymbol{x}) = u_{max} \tanh\left(\frac{1}{u_{max}} \sum_{i=1}^{N_b} w_i e^{-\|a_i - \boldsymbol{x}\|^2_{\Sigma_\pi}}\right), \; w_i \in R^{d_u} \tag{1.28}$$

which is a squashed, weighted sum of Gaussian basis functions.

The parameters are: $\boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{A}, \Sigma_\pi\}$, with $\boldsymbol{A} = \{a_1, \dots, a_{N_b}\}$ being the centers of the basis functions, weighted by $\boldsymbol{w} = [w_1, \dots, w_{N_b}]$. The matrix Σ_π is a covariance matrix used to shape the gaussian distribution, it can be assumed as diagonal with the diagonal elements called length-scales.

This policy's output values are constrained to lie in the interval $[-u_{max}, u_{max}]$, where u_{max} is assumed as the *constant* maximum control action, chosen depending on the particular system.

1.2.2.2 Gradient computation

The *reparameterization trick* is applied to compute the estimate of the expected cumulative cost (eq. (2.23)) with Monte Carlo sampling [36], as described in section 1.1.3.2.

Let $\boldsymbol{x}_\tau^{(m)}$, $m = 1, \dots, M$ and $\tau = 1, \dots, T$ be the state of the m-th particle at time τ, then the MC estimates of the expected cost and cumulative cost are:

$$\begin{aligned} \mathbb{E}[c(x_\tau)] &\sim \frac{1}{M} \sum_{m=1}^{M} c\left(\boldsymbol{x}_\tau^{(m)}\right) \\ \hat{J}(\boldsymbol{\theta}) &= \sum_{\tau=0}^{T} \left(\frac{1}{M} \sum_{m=1}^{M} c\left(\boldsymbol{x}_\tau^{(m)}\right)\right) \end{aligned} \tag{1.29}$$

At each time-step τ the expected state cost is approximated by the mean of the cost computed on the sampled particles.

The *reparameterization trick* is applied to define differentiable stochastic operations, the evolution of a particle $\boldsymbol{x}_\tau^{(m)}$ at following time step is obtained by sampling from:

$$p(x_{\tau+1}^m | x_\tau^m, \pi_\theta(x_\tau^m), \mathcal{D}) \sim \mathcal{N}(\mu_{\tau+1}, \Sigma_{\tau+1}) \tag{1.30}$$

defined in eq. (1.19). Instead of sampling directly from $\mathcal{N}(\mu_{\tau+1}, \Sigma_{\tau+1})$ it is convenient to sample from a properly dimensioned zero-mean and unitary variance gaussian distribution, then the particle is mapped to:

$$\begin{aligned} \boldsymbol{x}^{(m)}_{\tau+1} &= \mu_{\tau+1} + L_{\tau+1}\varepsilon, \\ L_{\tau+1}L^T_{\tau+1} &= \Sigma_{\tau+1}, \\ \varepsilon &\sim \mathcal{N}(0, I_{d_x}) \end{aligned} \tag{1.31}$$

where $L_{\tau+1}$ is the Cholesky decomposition of $\Sigma_{\tau+1}$. This *reparametrization* allows for standard differentiation, thus the gradient of $\hat{J}(\boldsymbol{\theta})$ from eq. (1.29) is computed using *backpropagation* with an optimizer like Adam [37].

The following computations show the dependencies of the Gradient of eq. (1.29) w.r.t. θ from the policy π_θ.

$$\begin{aligned} \nabla_\theta \hat{J}(\theta) = \frac{\partial \hat{J}(\theta)}{\partial \theta} &= \sum_{\tau=0}^{T} \left(\frac{1}{M} \sum_{m=1}^{M} \frac{\partial c(\boldsymbol{x}^{(m)}_\tau)}{\partial \theta} \right) \\ &= \sum_{\tau=1}^{T} \left(\frac{1}{M} \sum_{m=1}^{M} \frac{\partial c(\boldsymbol{x}^{(m)}_\tau)}{\partial \boldsymbol{x}^{(m)}_\tau} \frac{\partial \boldsymbol{x}^{(m)}_\tau}{\partial \theta} \right) \end{aligned} \tag{1.32}$$

$\frac{\partial \boldsymbol{x}^{(m)}_0}{\partial \theta} = 0$ since $p(x_0)$ does not depend on the policy. From eq. (1.31) the partial derivative of the particle's state w.r.t. policy parameters can be computed with eq. (1.19):

$$\begin{aligned} \frac{\partial \boldsymbol{x}^{(m)}_{\tau+1}}{\partial \theta} &= \frac{\partial \mu_{\tau+1} + L_{\tau+1}\varepsilon}{\partial \theta} = \frac{\partial \mu_{\tau+1}}{\partial \theta} + \frac{\partial L_{\tau+1}\varepsilon}{\partial \theta} \\ \frac{\partial \mu_{\tau+1}}{\partial \theta} &= \frac{\partial \boldsymbol{x}^{(m)}_\tau}{\partial \theta} + \left[\frac{\partial \mathbb{E}[\hat{\Delta}^{(1)}_\tau]}{\partial \theta}, \dots, \frac{\partial \mathbb{E}[\hat{\Delta}^{(d_x)}_\tau]}{\partial \theta} \right]^T \end{aligned} \tag{1.33}$$

Since $L_{\tau+1}$ is diagonal like $\Sigma_{\tau+1}$, it is easier to compute the gradient of $L_{\tau+1}\varepsilon$ rather then computing the gradient of $L_{\tau+1}$ and then multiply by ε. Considering:

$$\begin{aligned} L_{\tau+1} &= diag(\Big[\sqrt{\mathbb{V}[\hat{\Delta}^{(1)}_\tau]}, \dots, \sqrt{\mathbb{V}[\hat{\Delta}^{(d_x)}_\tau]}\Big]) \\ (L_{\tau+1}\varepsilon)^{(i)} &= \sqrt{\mathbb{V}[\hat{\Delta}^{(i)}_\tau]}\varepsilon^{(i)}, \;\; i = 1, \dots, d_x \end{aligned} \tag{1.34}$$

then we get:

$$\begin{aligned}\frac{\partial\left(L_{\tau+1}\varepsilon\right)^{(i)}}{\partial\theta} &= \frac{\partial\sqrt{\mathbb{V}[\hat{\Delta}_{\tau}^{(i)}]}}{\partial\theta}\varepsilon^{(i)} \in \mathbb{R}^{d_\theta} \\ &= \frac{1}{2\sqrt{\mathbb{V}[\hat{\Delta}_{\tau}^{(i)}]}}\frac{\partial\mathbb{V}[\hat{\Delta}_{\tau}^{(i)}]}{\partial\theta}\varepsilon^{(i)}\end{aligned} \tag{1.35}$$

The gradient computation of the i-th state element of the m-th particle w.r.t. θ is:

$$\begin{aligned}\frac{\partial\boldsymbol{x}_{\tau+1}^{(m,i)}}{\partial\theta} &= \frac{\partial\boldsymbol{x}_{\tau}^{(m,i)}}{\partial\theta} + \frac{\partial\mathbb{E}[\hat{\Delta}_{\tau}^{(i)}]}{\partial\theta} + \frac{\partial\sqrt{\mathbb{V}[\hat{\Delta}_{\tau}^{(i)}]}}{\partial\theta}\varepsilon^{(i)} \\ &= \sum_{k=0}^{\tau}\left(\frac{\partial\mathbb{E}[\hat{\Delta}_{k}^{(i)}]}{\partial\theta} + \frac{\partial\sqrt{\mathbb{V}[\hat{\Delta}_{k}^{(i)}]}}{\partial\theta}\varepsilon^{(i)}\right) \\ &= \sum_{k=0}^{\tau}\left(\frac{\partial\mathbb{E}[\hat{\Delta}_{k}^{(i)}]}{\partial\tilde{x}_k} + \frac{\partial\sqrt{\mathbb{V}[\hat{\Delta}_{k}^{(i)}]}}{\partial\tilde{x}_k}\varepsilon^{(i)}\right)\frac{\partial\tilde{x}_k}{\partial\theta}\end{aligned} \tag{1.36}$$

which shows recursive properties, with:

$$\frac{\partial\boldsymbol{x}_{0}^{(m,i)}}{\partial\theta} = 0 \quad \forall i = 1,\ldots,d_x; \quad m = 1,\ldots,M \tag{1.37}$$

From *full-state* model described in section 1.1.2, it is possible to yield the analytical expressions of gradients in eq. (1.36).

Since $\mathbb{E}[\hat{\Delta}\dot{q}_{\tau}^{(i)}]$ is the output value of the i-th GP with input $\tilde{x}_\tau$, the gradient is straightforward to compute for eq. (1.16):

$$\frac{\partial\mathbb{E}[\hat{\Delta}_{\tau}^{(i)}]}{\partial\tilde{x}_\tau} = \frac{\partial k_i(\tilde{x}_\tau,\tilde{\boldsymbol{X}})}{\partial\tilde{x}_\tau}\alpha_{\tau}^{(i)} \tag{1.38}$$

and for the variance term $\mathbb{V}[\hat{\Delta}_{\tau}^{(i)}]$:

$$\frac{\partial\mathbb{V}[\hat{\Delta}_{\tau}^{(i)}]}{\partial\tilde{x}_\tau} = \frac{\partial k_i(\tilde{x}_\tau,\tilde{x}_\tau)}{\partial\tilde{x}_\tau} - k_i(\tilde{x}_\tau,\tilde{\boldsymbol{X}})\left(\Gamma_i^{-1}+\Gamma_i^{-T}\right)\frac{\partial k_i^T(\tilde{x}_\tau,\tilde{\boldsymbol{X}})}{\partial\tilde{x}_\tau} \tag{1.39}$$

both of which depend only on the chosen kernel function. While the gradient of the extended state $\tilde{x}$ w.r.t. parameters θ is:

$$\frac{\partial\tilde{x}_\tau}{\partial\theta} = \begin{bmatrix}\frac{\partial x_\tau}{\partial\theta}\\ \frac{\partial u_\tau}{\partial\theta}\end{bmatrix} = \begin{bmatrix}\frac{\partial x_\tau}{\partial\theta}\\ \frac{\partial\pi_\theta(\tilde{x}_\tau)}{\partial\theta}\end{bmatrix} \in \mathbb{R}^{(d_x+d_u)\times d_\theta} \tag{1.40}$$

where the policy gradient peeps out.

The policy gradient is composed of the gradients of the various parameters, in case of the policy described in section 1.2.2.1:

$$\frac{\partial \pi_\theta(\tilde{x}_\tau)}{\partial \theta} = \left(\frac{\partial \pi_\theta(\tilde{x}_\tau)}{\partial \boldsymbol{w}}, \frac{\partial \pi_\theta(\tilde{x}_\tau)}{\partial \boldsymbol{A}}, \frac{\partial \pi_\theta(\tilde{x}_\tau)}{\partial \Sigma_\pi} \right) \tag{1.41}$$

and the term $\frac{\partial x_\tau}{\partial \theta}$ can be computed recursively

Putting all togheter:

$$\begin{aligned} \nabla_\theta \hat{J}(\theta) &= \sum_{\tau=1}^{T} \left(\frac{1}{M} \sum_{m=1}^{M} \sum_{i=1}^{d_x} \frac{\partial c(\boldsymbol{x}_\tau^{(m)})}{\partial \boldsymbol{x}_\tau^{(m,i)}} \frac{\partial \boldsymbol{x}_\tau^{(m,i)}}{\partial \theta} \right) \\ &= \sum_{\tau=1}^{T} \left(\frac{1}{M} \sum_{m=1}^{M} \sum_{i=1}^{d_x} \frac{\partial c(\boldsymbol{x}_\tau^{(m)})}{\partial \boldsymbol{x}_\tau^{(m,i)}} \sum_{k=0}^{\tau-1} \rho_k^{(m,i)} \begin{bmatrix} \frac{\partial x_k^{(m)}}{\partial \theta} \\ \frac{\partial \pi_\theta(x_k^{(m)})}{\partial \theta} \end{bmatrix} \right) \end{aligned} \tag{1.42}$$

with:

$$\rho_k^{(m,i)} = \frac{\partial \mathbb{E}[\hat{\Delta}_k^{(m,i)}]}{\partial \tilde{x}_k^{(m)}} + \frac{\partial \sqrt{\mathbb{V}[\hat{\Delta}_k^{(m,i)}]}}{\partial \tilde{x}_k^{(m)}} \varepsilon^{(i)} \tag{1.43}$$

The contribution of the gradient of the policy w.r.t. θ to this forumulation is clear.

The computation of each particle's component is independent from the other particles, therefore each term of the sum with index m in eq. (1.42) can be done in parallel and the recursive properties of the state gradient can be exploited.

1.2.2.3 Dropout

Gradient-based optimizations is known to suffer from the presence of local minima that often times trap the optimization in a sub-optimal configuration of the parameters. Various optimizers like Adam [37] incorporate techniques to reduce noise in the gradient computation, in order to best estimate the true gradient, but all SGD methods and derivatives have no guarantee of reaching global minima.

In order to discriminate a global minima, it is necessary to have a broad knowledge of the loss function $J(\theta)$, in this case, only a noisy estimate $\hat{J}(\theta)$ is available, therefore it is necessary to numerically explore the function in the parameters space ($\theta \in \Theta$). The authors of [12] proposed the use of dropout [33] to improve exploration in the parameters θ and increase the ability of escaping from local minima during policy optimization of MC-PILCO.

Considering the policy described in eq. (1.28), the dropout is applied by randomly dropping the weights $\boldsymbol{w}$ with probability p_d, this is implemented by scaling each center's weight w_i with a random variable $r_i \sim Ber(1-p_d)$ where $Ber(p_d)$ is a Bernoulli distribution assuming value $\frac{1}{1-p_d}$ with probability $1-p_d$ and 0 with probability p_d.

This is equivalent to formulating a distribution over the weights, thus yielding a parameterized stochastic policy π_θ.

As shown in [38], the distribution of each weight w_i can be approximated by means of a bimodal distribution, defined as the sum of two scaled Gaussians with infinitely small variance ξ^2:

$$w_i \sim p_d \mathcal{N}(0, \xi^2) + (1-p_d)\mathcal{N}\left(\frac{w_i}{1-p_d}, \xi^2\right) \tag{1.44}$$

The use of a stochastic policy during policy optimization phase has the effect of increasing the entropy of the particles' distribution, thus incrementing the probability of visiting low-cost regions of the parameters' space, escaping local minima.

The authors of [12] validated the additional property of dropout of mitigating issues related to exploding gradients, probably due to the fact that the gradient is computed with several different values of $\boldsymbol{w}$. Since different values of $\boldsymbol{w}$ yield different policies, the gradient estimates are computed by averaging with slightly different policies, obtaining a regularization effect.

The downside of dropout is that the additional entropy of a stochastic policy might affect the precision of the obtained solution, therefore it is necessary that the policy optimization step returns a deterministic policy.

For these reasons, the authors of [12] defined a heuristic scaling procedure to gradually decrease the dropout rate, p_d, until reaching 0, during the policy update iterations.

A monitoring signal s is defined as function of the statistics of $\hat{J}$ in the preceding iterations, the expression is:

$$\begin{aligned}
&s_j = \alpha_s s_{j-1} + (1-\alpha_s)\frac{\mathcal{E}[\Delta \hat{J}_j]}{\sqrt{\mathcal{V}[\Delta \hat{J}_j]}} \\
&\mathcal{E}[\Delta \hat{J}_j] = \alpha_s \mathcal{E}[\Delta \hat{J}_{j-1}] + (1-\alpha_s)\Delta \hat{J}_j \\
&\mathcal{V}[\Delta \hat{J}_j] = \alpha_s(\mathcal{V}[\Delta \hat{J}_{j-1}] + (1-\alpha_s)(\Delta \hat{J}_j - \mathcal{E}[\Delta \hat{J}_{j-1}])^2) \\
&\Delta \hat{J}_j = \hat{J}(\theta_j) - \hat{J}(\theta_{j-1})
\end{aligned} \tag{1.45}$$

where $\Delta \hat{j}_j$ is the cost change between the two consecutive policies

$\pi_{\theta_{j-1}}$ and π_{θ_j} respectively at the $j-1$ and j-th optimization steps.

The quantities $\mathcal{E}[\Delta\hat{J}_j]$ and $\sqrt{\mathcal{V}[\Delta\hat{J}_j]}$ are respectively the mean and standard deviation of $\Delta\hat{J}_j$, computed with Exponential Moving Average (EMA) filtering with coefficient α_s, also referred as the filter's memory.

The signal s_j is an EMA filtered version of the ratio $\frac{\mathcal{E}[\Delta\hat{J}_j]}{\sqrt{\mathcal{V}[\Delta\hat{J}_j]}}$.

The optimization algorithm checks at each iteration, if the absolute value of s was below a certain threshold σ_s in the last n_s iterations:

$$[|s_{j-n_s}|, \dots, |s_j|] < \sigma_s \tag{1.46}$$

where the minor sign is applied element-wise. If this condition is true, the dropout rate p_d is decreased by δp_d while both the optimizer's learning rate α_{lr} and threshold σ_s are scaled by an arbitrary factor λ_s, until reaching minimum allowed value:

$$\begin{aligned} p_d &= \max(p_d - \Delta p_d,\ 0) \\ \alpha_{lr} &= \max(\lambda_s \alpha_{lr},\ \alpha_{lr_{\min}}) \\ \sigma_s &= \lambda_s \sigma_s \end{aligned} \tag{1.47}$$

The procedure is iterated as long as:

$$p_d \geqslant 0 \text{ and } \alpha_{lr} \geqslant \alpha_{lr_{\min}} \tag{1.48}$$

The idea behind the heuristic is that the signal s would reach small value when the optimization reaches convergence (low $\mathcal{E}[\Delta\hat{J}_j]$) or when the Model's predictions are uncertain (high $\mathcal{V}[\Delta\hat{J}_j]$), in both these cases it is necessary to close the optimization and collect more data either for performance validation or to strengthen the model's prediction.

MC-PILCO with dropout is summarized in algorithm 1.

It is worth mentioning that some adaptation is likely to be required to use MC-PILCO in particular setups, depending on the required task. In the next chapter we will describe all the modifications and additions that were required to solve the considered task. Moreover, we compared the performances of policy optimization and the effects on the resulting policy under different setups with and without the use of dropout.

The following table contains all optimization parameters defined in [12] and related values tuned by the authors. If not specified differently, the same values were used throughout all this work.

Algorithm 1 MC-PILCO with Dropout

```
Require: Policy $\pi_\theta$, cost $c$, kernel $k$,
  optimization parameters: $N_{opt}, M, \alpha_{lr}, \alpha_{lr_{\min}}, p_d, \Delta p_d$
  and monitoring signal parameters: $\sigma_s, \lambda_s, n_s$.

  Apply initial control policy to system and collect data.
  while task not successfully learned do
      (1) Model Learning:
      GP models are learned from sampled data - section 1.2.1
      (2) Policy Update:
      $s_0 = 0$
      for $j = 1, \ldots, N_{opt}$ do
          Simulate $M$ particles rollouts with current GP models and $\pi_\theta$
          Compute $\hat{J}(\theta_j)$ from eq. (1.29)
          Compute $\nabla \hat{J}(\theta_j)$
          $\pi_{\theta_{j+1}} \leftarrow$ gradient-based update
          Update $s_j$ with eq. (1.45)
          if (1.46) then
              Update $p_d$, $\alpha_{lr}$ and $\sigma_s$ with eq. (1.47)
          end if
          if not (1.48) then
              exit for loop
          end if
      end for
      (3) Policy Execution:
      apply updated policy to system and collect data.
  end while
  return final policy $\pi_{\theta*}$ and system model;
```

parameter	description	value
p_d	*dropout rate*	0.25
Δp_d	p_d *reduction* term	0.125
α_{lr}	*optimizer's learning rate*	0.01
$\alpha_{lr_{\min}}$	*minimum learning rate*	0.0025
α_s	EMA *filter memory*	0.99
σ_s	*threshold of monitoring signal*	0.08
n_s	*number of monitored iterations*	200
Λ_s	*reduction coefficient of* σ_s and α_{lr}	0.5

Table 1.1: Standard values for the policy optimization parameters [12].

The full algorithm was implemented by the authors of [12] in Python with the PyTorch [39] library, the code is publicly available [1] and was exploited in this work.

[1]MC-PILCO's code is available at: `https://www.merl.com/research/license/MC-PILCO`

Chapter 2

Industrial Robotics Application: Tossingbot

The application described in this chapter was inspired by Tossingbot [1], where the authors explored how to learn a full *pick-and-throw* task of arbitrary objects using *self supervised learning.* Namely, Tossingbot exploits Deep Convolutional Neural Networks (DCNNs) [22] both to learn how to grasp an arbitrary object and the linear velocity it needs to be tossed from a release position in order to reach a target bin, by *trial and error.* The authors of Tossingbot trained a DCNN to learn the grasping orientation for throwing and a second DCNN for the tossing task, these networks are used sequentially at each tossing trial[1], the predicted grasping orientations and tossing speed are then used with two robot *primitives* to perform the *pick-and-throw* operation. The tossing module is not used for full regression, i.e. its targets are not the velocities v_{toss} but it predicts a residual δ which is used to compute the final release velocity provided to the tossing primitive by:

$$\|v_{toss}\| = \|\hat{v}_{toss}\| + \delta \tag{2.1}$$

where $\hat{v}_{toss}$ is the velocity computed by standard *ballistic* equations that describe the accelerated motion of a bullet under the effects of gravity. This setup, named *Residual-physics*, leverages the generalization of the ballistic equations to adapt to new target locations, while the DCNN is used to account for unmodeled dynamics, e.g. momentums or frictions.

The use of this setup is justified by the results presented in [1], which show significant advantages of predicting a residual w.r.t. a full regression of the tossing velocity.

During the training phase, a certain number of tossing trials are performed and the outcome is registered, this data is used to jointly

[1]a *tossing trial*, in this context, is the execution of a single robot tossing trajectory that results in the bullet entering the target bin or missing it.

train the grasping and tossing modules. Lastly, a testing phase evaluates the performances of the trained modules in terms of success rate of reaching the target bin.

The task of Tossingbot is clear, pick an arbitrary object and toss it into a bin, in a self-supervised way. In this work we focused on the last part of the task, learning to toss objects with a robotic manipulator, and we evaluated the application of the Model-Based Reinforcement Learning framework MC-PILCO. We considered the simplest object to throw: a ball, which is one of the objects considered in [1], and the results show very good performances.

The reason that drove this work is showing that this kind of task can be learned in a much more efficient way, indeed [1] shows that the *self supervised learning* approach works pretty good in practice, but requiring thousands of training trials even in simulation to reach convergence. This is mostly due to the fact that Tossingbot's approach relies on DNNs which are known to be typically data-inefficient.

Instead, in this work we show that by adapting MC-PILCO's Model-Based framework, described in section 1.2, we are able to teach the task to the robot with a very limited amount of exploration trials. This is very interesting for industrial robotics, since this results in a lowered wear and occupation of the equipment during a preliminary phase.

The main idea is to take advantage of MC-PILCO's Model-Based Policy Seach algorithm to train a tossing policy, that is able to predict the cartesian velocity vector that the robot needs to impose to the bullet from a release position, in order to reach an arbitrary landing position. The Model Learning step is not required to account for the robot's dynamics, but only for the bullet dynamics from the moment it is released until the moment it reaches target altitude. This is possible until the robot control does not introduce too much noise during execution of the tossing trajectory.

The robotic manipulator chosen for the task is a Franka Emika (FE) Panda Robot, which is a 7 DOF collaborative manipulator with only revolute joints. The reasons of this choice are dual:

- a Panda robot is available for these experiments in the Department of Information Engineering (DEI) Robotics laboratory;
- Franka Emika provides uniform support both for control of the real robot and for simulations in Gazebo simulator [11], under the Robot Operating System (ROS) framework [40].

This chapter will first describe in a precise way the task, then

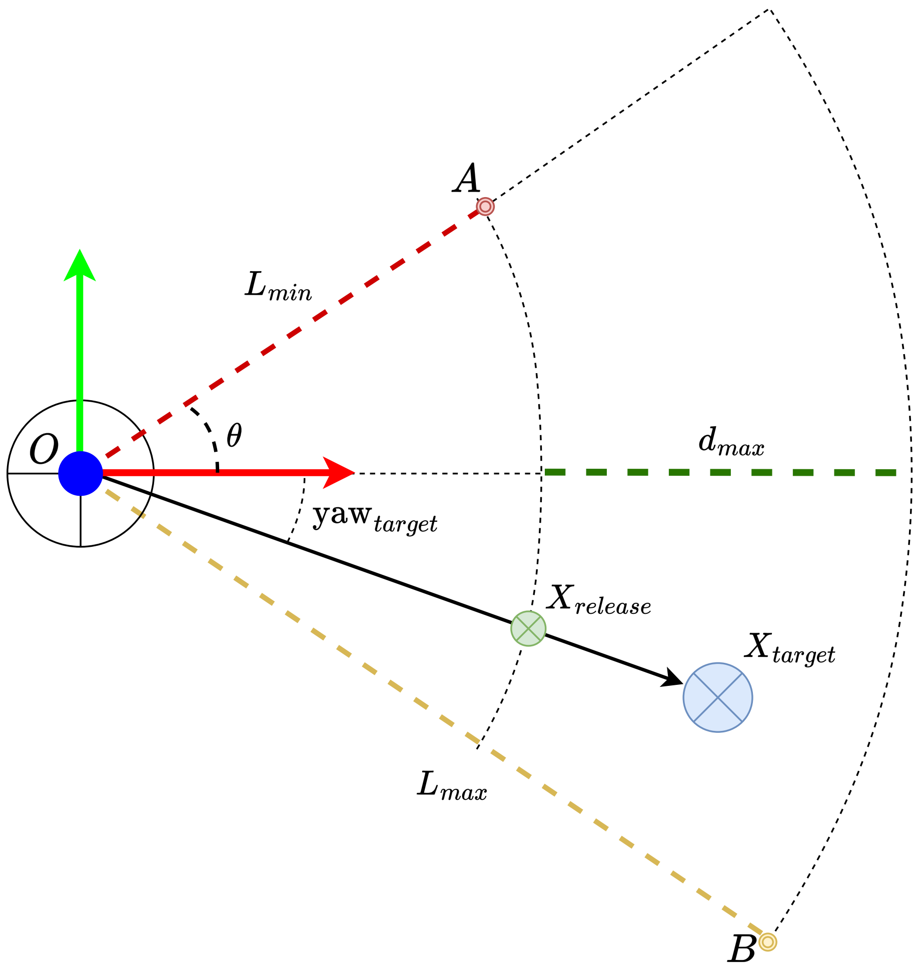

Figure 2.1: Scheme describing the task (top view)

section 2.1 will present the simulating environment that was prepared for validation of the learning framework, section 2.2 will describe how MC-PILCO was adapted to the considered task, finally section 2.3 will introduce the work that we were able to implement and test in the laboratory of DEI.

The task can be summarized by the schemes in fig. 2.1 and fig. 2.2, the RGB triplet of axes in the left part of the figure is the reference frame of the robot's base, supposing that the target bins all lie at the same altitude, namely z_t, then the target domain can be described as:

$$X_t = \begin{bmatrix} \cos(\text{yaw}_t) \cdot \ell \\ \sin(\text{yaw}_t) \cdot \ell \\ z_t \end{bmatrix} \tag{2.2}$$

where $\ell \in [L_{min}, L_{max}]$ and $\text{yaw}_t \in [-\theta, \theta]$. This simplifies the modelling, but we will show in section 2.2 that the algorithm can easily be adapted to work on arbitrary target positions.

Similar to what the authors of [1] did, we constrained the release

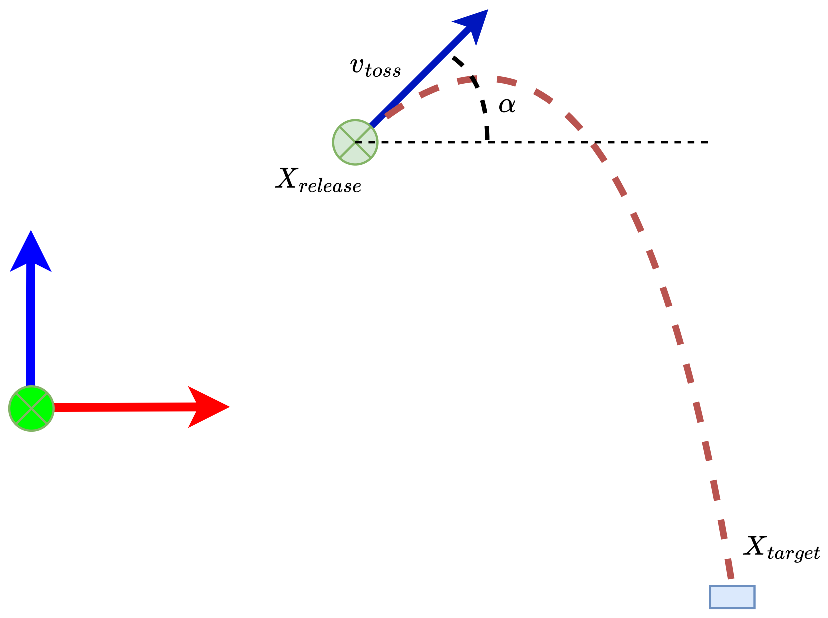

Figure 2.2: Scheme describing the task (side view)

position to lie on the semi-circumference of radius L_{min}, aligned with the target direction, namely:

$$X_{rel} = \begin{bmatrix} \cos\left(\text{yaw}_{target}\right) \cdot L_{min} \\ \sin\left(\text{yaw}_{target}\right) \cdot L_{min} \\ z_{release} \end{bmatrix} \tag{2.3}$$

Furthermore, we fixed the orientation of the tossing velocity vector v_{toss} to be angled at $\alpha = \frac{\pi}{4}$ radiants upwards.

The parameters L_{min} and $z_{release}$ are completely dependent on the kinematic of the robotic manipulator [41], while L_{max} must be sensibly calibrated such that positions X_t can be reached. Lastly v_{toss} cannot be any velocity, robots manufacturers provide information on the limits in both joints and cartesian space, in this case v_{toss} must be at least smaller than the maximum velocity in cartesian space indicated by the manufacturer, but in general this limit also depends on the trajectory that was designed [2].

2.1 Simulations in Gazebo

Franka Emika provides all the libraries for control of the manipulator, as well as the the code for simulating the robot in Gazebo, therefore it was not difficult to build from scratch a minimal simulating environment, which can be seen in fig. 2.3, it consists of:

[2]We referred to Franka Emika's parameters for the Panda robot: `https://frankaemika.github.io/docs/control_parameters.html`

- Gazebo simulator and Robot Operating System (ROS);
- FE Panda Robot;
- A red ball, modeled to replicate weight and dimensions of a golf ball[3]. The reference frame of the model is placed at the exact geometric center of the sphere;
- A hollow green cylinder, of 10cm of diameter, with 0.5cm border, of height 10cm that worked as the target bin for the tossing task. It was custom built for this application;
- A taller green cylinder which is used to place the ball for the robot to pick up;
- A table, where the Panda robot is *«placed»*, just for aesthetics.

The robot interface was provided by *MoveIt!* [42] and an *effort* controller[4] provided by the *ros_control* framework [43], was used to control the execution of the tossing trajectory.

During the exploration and test trials, the target cylinder is moved arbitrarily in the target space in front of the robot (just described in fig. 2.1), at ground level, since the goal of the task is to make the bullet fall into the cylinder, we set the x, y coordinates of the target position to be the geometrical center of the cylinder's base, while the z coordinate is set to be the altitude of the cylinder's top face, in the considered reference frame. In this simulation, we chose, for simplicity, to place the robot's base frame at a certain altitude w.r.t. the world frame, namely the pose of the base frame of the Panda is obtained by applying the following transformation matrix to the world frame:

$$T_B^w = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & z_b \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

as it can be clearly seen in fig. 2.3, it is just a translation over the z axis of the world frame.

Therefore, the position of the target X_t expressed in the world frame is:

$$X_t = \begin{bmatrix} \cos(\text{yaw}_t) \cdot \ell \\ \sin(\text{yaw}_t) \cdot \ell \\ h_{cyl} \end{bmatrix} \tag{2.5}$$

[3] `https://en.wikipedia.org/wiki/Golf_ball`

[4] *effort_joint_trajectory_controller* specifically

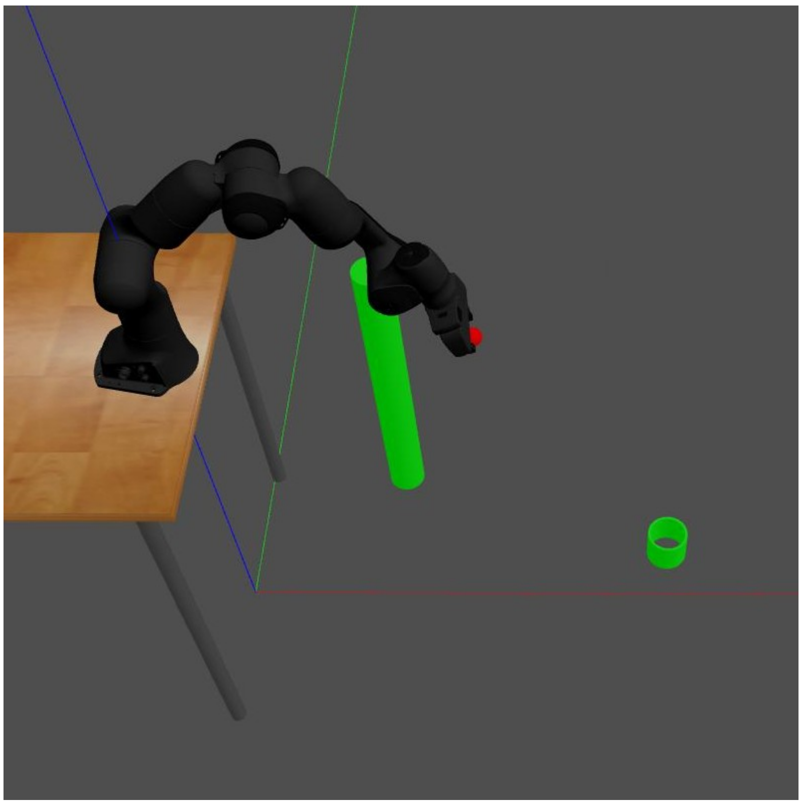
Figure 2.3: Gazebo simulation

with $h_{cyl} = 0.1$m being the height of the cylinder.

All the positions, if not specified, will be expressed in the world reference frame of the simulation, for convenience of the programmer. This can be done without fear of losing generality, for example, all positions could be re-expressed in the robot reference frame if needed, by just applying the needed transformation T_w^B.

In order to learn the bullet dynamics, the MB-RL framework requires the simulation to provide the data of the trajectories performed by the bullet during unconstrained motion from release until it reaches the altitude of the target. The trajectories must be free of collisions, therefore the hollow cylinder in fig. 2.3 cannot be used when collecting data for model learning, because it is likely that the bullet might collide with its edges. It will only be used when testing the tossing policy, instead, during the exploration trials we used a lower cylinder just as a placeholder.

The data of the bullet trajectories was collected using Gazebo's integrated odometry plugin[5], which provides both the positions and velocities in the desired reference frame.

The execution of a *tossing trial*[6] is composed of the following steps:

[5]specifically: *gazebo_ros_p3d* plugin `https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/include/gazebo_plugins/gazebo_ros_p3d.h`

[6]A short recording of the simulated experiment is available at: `https://drive.google.com/`

- The ball is spawned over the pickup green cylinder;
- The target bin is spawned in front of the robot, in one of the possible positions describe in fig. 2.1;
- The robot picks the ball from the pickup position such that the origin of the ball reference frame is coincident with the origin of the end-effector reference frame.
- The robot performs the tossing at the Cartesian velocity indicated by the policy or computed random, this is explained in section 2.1.1;
- The trajectory of the bullet from release until it reaches altitude z_t is recorded and forwarded to the model learning block.

It was chosen not to spawn the bullet already attached to the end-effector to simulate the grasping errors that are inevitable in a real scenario.

2.1.1 Control of Robot Tossing

The main goal of the simulation was to provide a benchmark environment to extensively evaluate the performance of the Monte Carlo gradient estimation (described in section 1.2). The first objective in this task was to design a robot trajectory that is able to bring the end-effector to the desired release position with desired cartesian velocity. The trajectory computation is required to be computationally efficient, in order to be able to perform many trials in few time, and to be precise enough that the release velocity is not too much uncertain.

The approach chosen for the trajectory generation is pretty straightforward, it is required to reach a release positioon X_{rel} with the end-effector of the manipulator, with velocity:

$$v_{toss} = \|v_{toss}\| \begin{bmatrix} \cos(\mathrm{yaw}_t) \cdot \cos(\alpha) \\ \sin(\mathrm{yaw}_t) \cdot \cos(\alpha) \\ \sin(\alpha) \end{bmatrix} \tag{2.6}$$

which is the velocity vector described at the beginning of the chapter.

This requirement can be easily translated from cartesian space into

file/d/1xNmkhDtpo_v9ZzN3xaZgijfw7tfrrhJe/view?usp=sharing

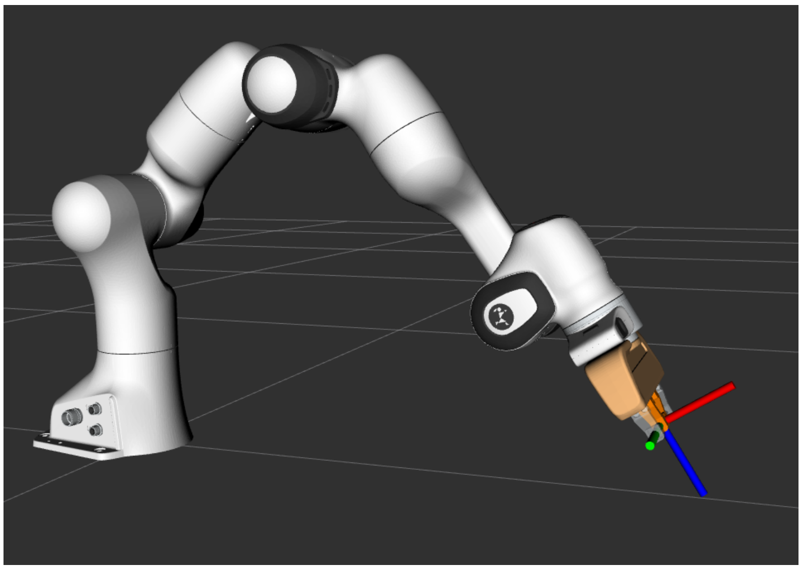
Figure 2.4: The end-effector reference frame at the robot tossing pose

joint space by considering:

$$\begin{aligned} f_k(q_{rel}) &= \begin{bmatrix} R_{rel} & X_{rel} \\ 0\ 0\ 0 & 1 \end{bmatrix} = T_{rel} \\ J(q_{rel})\dot{q}_{rel} &= \begin{bmatrix} v_{toss}^T & 0 & 0 & 0 \end{bmatrix}^T \end{aligned} \tag{2.7}$$

namely, T_{rel} is the output pose of the *forward kinematics* function [41] of the manipulator, targeting the end-effector reference frame, q_{rel} is the joint configuration that achieves said end-effector pose, while $J(q_{rel})$ is the *Jacobian* matrix [41] of the manipulator at joint configuration q_{rel}, reffered to the reference frame of the end-effector. Clearly, X_{rel} is a position in cartesian space, R_{rel} is, instead, a rotation matrix that expresses the orientation of the end-effector frame, it is very easy to compute, looking at fig. 2.4.

R_{rel} is the rotation matrix describing the rotation that brings the x axis of the end-effector frame to be aligned with the cartesian velocity vector v_{toss}, it can be seen clearly in fig. 2.5 that the x axis is perpendicular to the segment that connects the touching points of the gripper's pliers. Therefore, one can compute:

$$\begin{aligned} q_{rel} &= f_k^{-1}(T_{rel}) \\ \dot{q}_{rel} &= J^{\dagger}(q_{rel}) \begin{bmatrix} v_{toss}^T & 0 & 0 & 0 \end{bmatrix}^T \end{aligned} \tag{2.8}$$

where $J^{\dagger}(q_{rel})$ is the pseudoinverse of the *Jacobian* matrix of the manipulator at joint configuration q_{rel}, and f_k^{-1} is the *inverse kinematics* function of the robot.

If we are able to bring the robot, with the bullet grasped in the

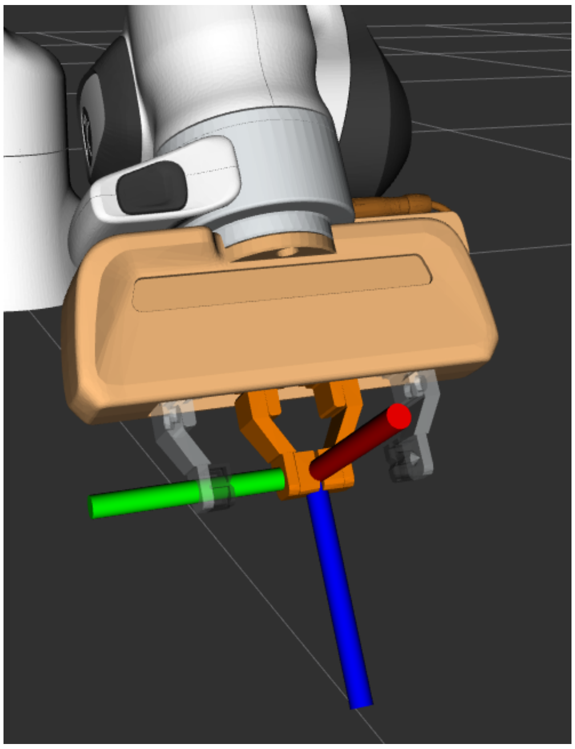
Figure 2.5: Closeup of the end-effector reference frame

end-effector, at joints configuration q_{rel} at joints velocities $\dot{q}_{rel}$ and we release the ball at the right moment, we can impose the cartesian velocity v_{toss} to it, therefore giving rise to an unconstrained motion of initial velocity v_{toss}.

In general, the joints configuration q_{rel} is not unique, therefore it was chosen specifically to reduce the complexity of the motion, i.e. to reduce the number of joints moving. Then, the position X_{rel} is a consequence of the following choices, but that is fine since the parameters L_{min} and z_{rel} of eq. (2.3) are arbitrary.

In practice, considering $\text{yaw}_t = 0$ without loss of generality, it was chosen q_{rel} such that:

- The axis of joints 2, 4 and 6 are parallel to each other and perpendicular to the axis of the joint 1, these are the only joints that contribute to the motion;
- The robot is in *elbow-up* configuration;
- the last joints are positioned such that the end-effector reference frame is at the desired orientation R_{rel};
- Joints 3 and 5 are at their zero position.

At such configuration, if we compute the projection on the ground plane, of the x axis of the end-effector frame, we find that it is always

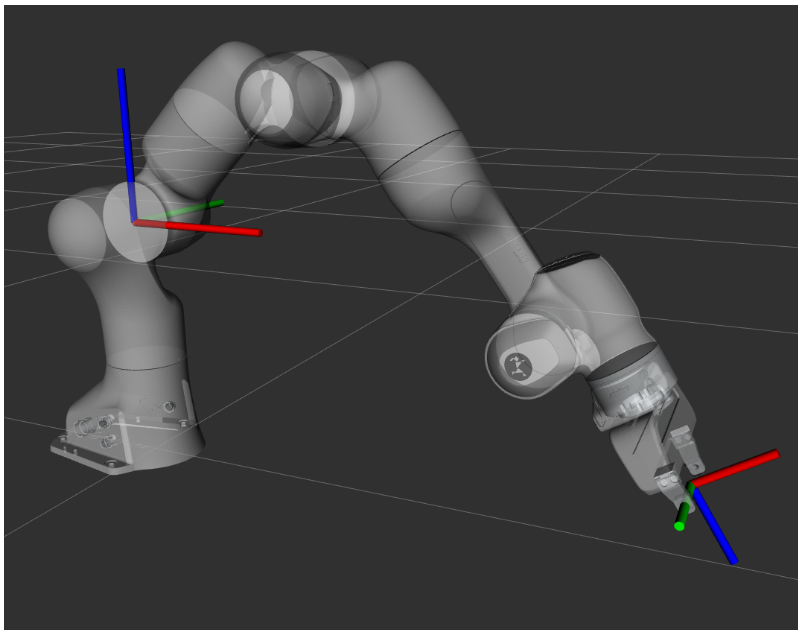
Figure 2.6: The end-effector and joint 2 reference frames at the robot tossing pose

parallel to the x axis of the joint 2 reference frame, it can be seen in fig. 2.6. Therefore by adjusting the first joint, we can move the release position X_{rel} on the semi-circumference described in fig. 2.1, and the tossing velocity vector is then:

$$v_{toss} = \|v_{toss}\| \begin{bmatrix} \cos(q_{rel,1}) \cdot \cos(\alpha) \\ \sin(q_{rel,1}) \cdot \sin(\alpha) \\ \sin(\alpha) \end{bmatrix} \tag{2.9}$$

where $q_{rel,1} = \text{yaw}_t$ is the value of the first joint at release pose, and it fully determines the parameter yaw_t.

In the simulation we performed the following approximation:

$$\begin{aligned} \dot{q}_{rel} = J^{\dagger}(q_{rel}) v_{toss} &= \begin{bmatrix} \dot{q}_{rel,1} & \dot{q}_{rel,2} & \dot{q}_{rel,3} & \dot{q}_{rel,4} & \dot{q}_{rel,5} & \dot{q}_{rel,6} & \dot{q}_{rel,7} \end{bmatrix}^T \\ \dot{\hat{q}}_{rel} &= \begin{bmatrix} 0 & 0 & 0 & \dot{q}_{rel,4} & 0 & \dot{q}_{rel,6} & 0 \end{bmatrix}^T \\ \dot{q}_{rel} &\sim \dot{\hat{q}}_{rel} \end{aligned} \tag{2.10}$$

while it is true that $\dot{q}_{rel,1}, \dot{q}_{rel,3}, \dot{q}_{rel,5}, \dot{q}_{rel,7} \sim 0$, $\dot{q}_{rel,2}$ from the computation is not close to zero, but it is a small enough value that it can be neglected with reasonable effects on the tossing error. The next chapter will show that this trajectory generation is precise enough to be used as described in section 2.2.

In the end, it is necessary to just move 2 joints, such that they reach their target values, with target velocity. This goal can be re-expressed in the following way: joint k must move from value q_{k_0} to $q_{rel,k}$ in time

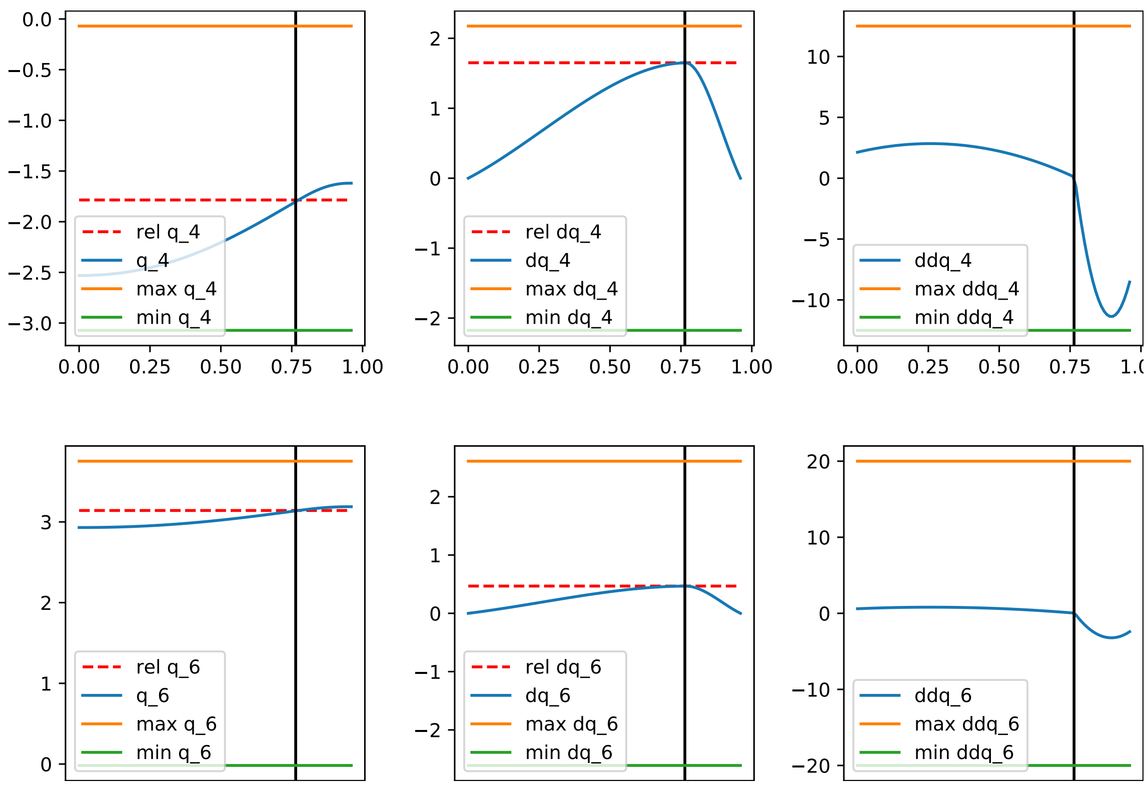

Figure 2.7: The reference trajectories of joints 4 and 6 during a toss with $\|v_{toss}\| = 1m/s$. [positions left, velocities middle, accelerations right]

τ_F such that the velocity of the same joint at time τ_F is the desired velocity $\dot{q}_{rel,k}$.

Let's define:

$$\Delta q_k = q_{rel,k} - q_{k_0} \tag{2.11}$$

and consider the function $s(\tau)$ defined as:

$$\begin{aligned} s(\tau) &= -\frac{a}{4\tau_F^3}\tau^4 + \frac{a}{3\tau_F^2}\tau^2 + \frac{a}{2\tau_F}\tau^2 \\ \dot{s}(\tau) &= -\frac{a}{\tau_F^3}\tau^3 + \frac{a}{\tau_F^2}\tau^2 + \frac{a}{\tau_F}\tau \end{aligned} \tag{2.12}$$

where $\tau_F = \frac{12\Delta q}{7\dot{q}}$ and $a = \dot{q}_{rel}$.

Then we have $s(\tau_F) + q_{k_0} = q_{rel,k}$ and $\dot{s}(\tau_F) = q_{rel,k}$ which is our goal. In fig. 2.7 we show the reference trajectories that are computed with eq. (2.12) and forwarded to the robot controller to execute a tossing trial of unitary cartesian velocity, the vertical lines indicate the instant τ_F, which is the time that the joints reach the target position and velocity, that is the moment when the bullet needs to be released from the end-effector. The final part of the trajectories is composed of a damping period which brings velocities to zero.

By no means this is the most precise design possible, but it is very efficient, requiring just to compute and invert a single *Jacobian* matrix and to compute 2 joint trajectories with eq. (2.12), which is a completely known polynomial function. Moreover this computation is completely immune to the ever-persisting problem of *Jacobian* singularities, since the configuration q_{rel} is for sure non-singular. Other approaches require to compute and invert the *Jacobian* matrices over all the trajectory (section 2.3).

The only problems that might be encountered with this approach are that the trajectories may exceed the joints limits in terms of acceleration, and that the acceleration at final time is not zero. In practice, these are not pressing concerns, because we managed to tune the the acceleration times and Δq such that these limits are not reached and what happens after τ_F is of no concern since it does not affect the bullet trajectory.

It must be noted that this approach typically causes heavy oscillations to the 4th joint when stopping and would likely cause errors or damage to a real manipulator. Indeed when working with the Panda robot in DEI's laboratory we relied to a safer approach.

Finally, we tested the precision of our tossing system, by performing a series of 190 tossing trials in the range of Cartesian velocities that are of interest for this application. In fig. 2.8 we compare the Cartesian velocities that were given in input to the system that was just described, with the actual velocities of the bullet measured with the odometry. The resulting curve is not too far from the identity function that is plotted for convenience, indeed it will be shown in the next chapter that we were able to account for this noise in the test performed in this simulation.

2.1.2 Bullet Release

Handling the release of the bullet was the most challenging aspect of the work related to the simulation and the same is true for the experiments performed in the laboratory, for the following reasons:

- The gripper provided by Franka Emika is already simulated in Gazebo and all code for its control is provided, but the control is not real time and it is not possible to synchronize the joints trajectory with the gripper aperture/closure. The same is true for the physical gripper.
- Modeling friction in Gazebo is not automatic, particularly for user-defined models, and all frictions must be modeled when defining

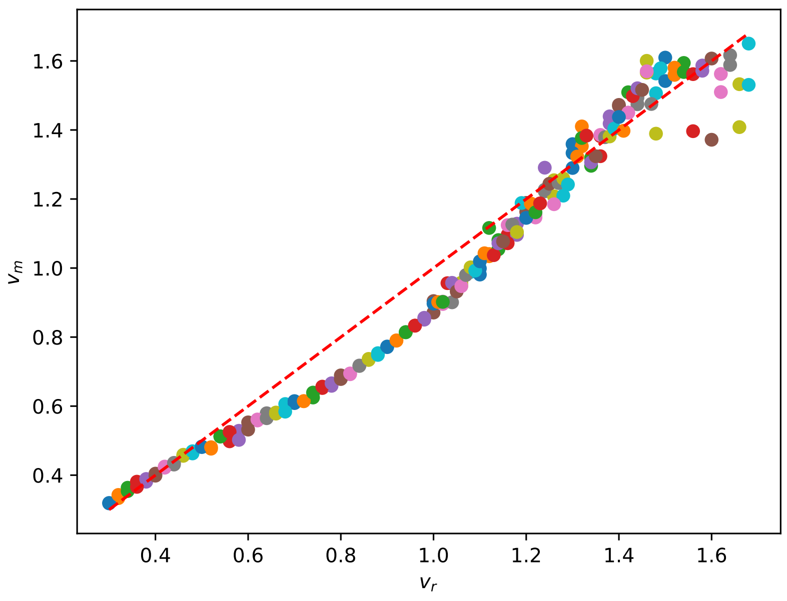

Figure 2.8: The comparison between the required tossing velocity (v_r) and the actually measured velocity (v_m) at release of the ball, captured by the odometry. Please note that the Panda robot's max nominal velocity in Cartesian space is 1.7m/s.

the geometry of the objects. If these parameters are not set in the ball's model, it is not possible to perform grasping in simulation.

Therefore to actually utilize Franka Emika's gripper to grasp the bullet one needs to model the friction effects in order not to lose the bullet before the actual release point. Even if these parameters are set correctly, and the robot is able to grasp the bullet with its gripper, one cannot program the release of the same at the correct instant of time, because of the lack of real time control and/or synchronization.

We decided to avoid this issue all together and postpone it to when it was time to perform the experiments with the physical manipulator, instead in the simulations we relied to a Gazebo plugin provided by Pal Robotics, named *gazebo_ros_link_attacher*[7]. When needed to pick the object, the plugin is used to attach the reference frame of the ball's Gazebo model to the last reference frame of the manipulator's model, and detach the frames to release it.

The problem of synchronization of bullet release with robot trajectory is not solved by the link attacher plugin, which requires to be called at the correct time. The effort controller we used to forward the trajectories to the robot allows the user to define a callback function which provides the updated state of the robot at a certain frequency.

[7]available at `https://github.com/pal-robotics/gazebo_ros_link_attacher`

Unfortunately, we discovered that said frequency is far too low for our aims, therefore we relied to a different approach.

Fortunately, the odometry plugin that was exploited is able to work at high sampling rates, and in the simulations it was set to the highest value available: $1KHz$. A ros node was specifically implemented to continuously read the bullet odometry and perform the detachment of the ball from the robot end effector when the measured position of the ball is close to the nominal release position. In practice we had to anticipate the release of a small distance to account for delays in the detachment, the final results are good and we measured the distance of the actual release point from the nominal point being in average well under $1cm$.

In fig. 2.9 and fig. 2.10 it is possible to observe some plots related to an example throw performed with the simulator. In fig. 2.9 the left image shows a trajectory collected in an example toss in the simulation, the red dashed curve is the raw data, while the blue curve is the same data resampled at constant frequency with linear interpolation, in the same plot it is possible to see the release point as the small green triangle. The right image of fig. 2.9 shows the velocity norm along the trajectory in blue, which has the typical shape of accelerated motion, while the target speed given in input to the tossing system is the orange horizontal line. This last plot shows that the system, in this particular example, was overshooting, this is a known problem and section 2.2.1.1 will explain how it was tackled.

The plots in fig. 2.10 are referred to the trajectory shown in fig. 2.9, the left one showing the plots of the velocities and the right one presenting the boxplots of the velocity deltas, box divided in the x, y, z components. The most relevant information that can be inferred from this plots is the statistics captured by the boxplot in fig. 2.10 which reveals that the data collected by the systems is very simple, because the deltas in the 3 components are clearly almost constant, with Δv_z being negative as expected.

2.2 Learning a Robot Tossing Policy with MC-PILCO

The considered task requires the robot to throw a bullet in arbitrary target location, the initial position of the system (i.e. release point of the bullet) and the direction of the initial velocity are constrained by the target. We can think of the tossing system described in this

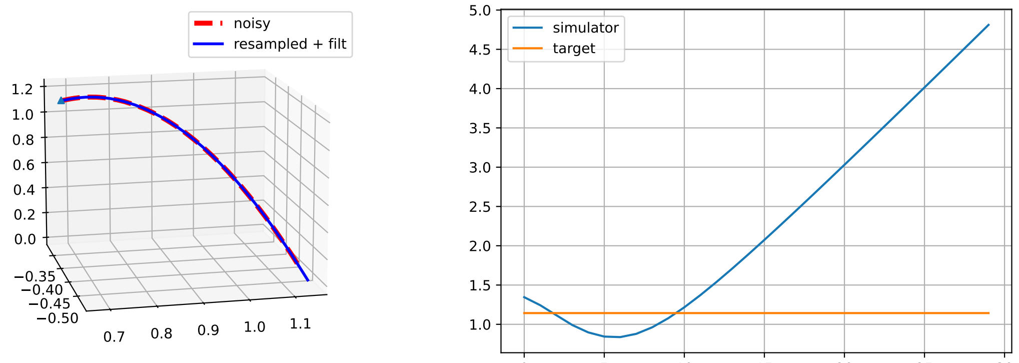

Figure 2.9: [a,b], (a): 3D plot of a trajectory performed in a simulated toss, (b): the respective plot of the velocity norm along the trajectory

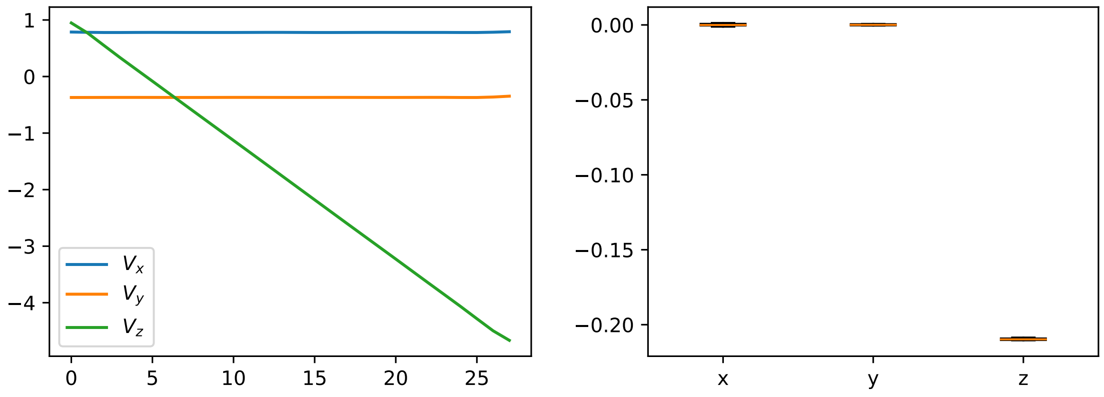

Figure 2.10: [a,b] referring to fig. 2.9, (a): velocities of the x, y, z components, (b): the respective boxplot of the velocity deltas of the trajectory

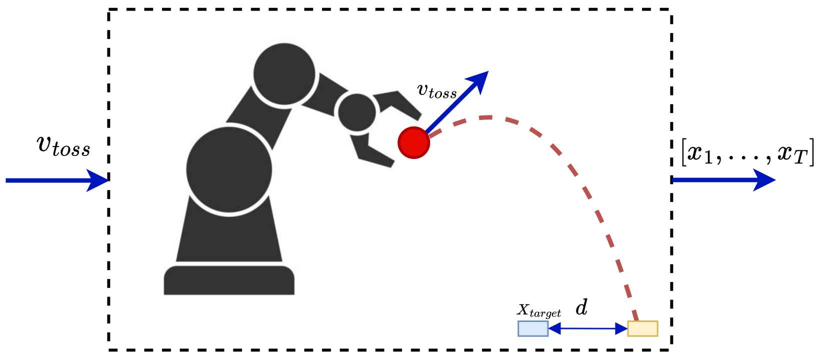

Figure 2.11: The tossing system control scheme

chapter as a system which takes a single discrete control action, i.e. a scalar representing the initial velocity norm of the bullet, and returns a single output which is the trajectory described by the bullet from the initial position until it reaches target altitude, check out fig. 2.11 to have a idea.

Therefore the policy does not need to learn a mapping from any state to the control action, it needs to learn a function that takes in input the target position and returns the control action needed to reach the goal state, i.e. the norm of the release velocity to forward to the tossing system.

The model used for deriving a tossing policy does not consider the robot's dynamics for simplicity, it only assumes that the robot tossing determines the initial bullet velocity at given release position, possibly with some noise.

In section 2.2.1 we describe a variation of the *speed-integration* model to predict the landing position of a bullet given the initial velocity and release point, in section 2.2.2 we present a cost function that depends on the distance of the landing position from the target, finally in section 2.2.3 we portray the designed tossing policy based on the general purpose policy described in section 1.2.2.1.

2.2.1 Model definition

The model considers only the evolution of the bullet from the release position, until reaching target level, since the cost function by definition takes as input the system's state and in this case the state must take into account the target position in some way. The simplest way to do so is by concatenating the bullet state at time τ with the target position it is required to reach in the current trial.

The state is then described by the vector:

$$x_\mathcal{T} = \begin{bmatrix} p_\mathcal{T} \\ \dot{p}_\mathcal{T} \\ X_t \end{bmatrix} \in \mathbb{R}^9$$
$$p_\mathcal{T}, \dot{p}_\mathcal{T}, X_t \in \mathbb{R}^3 \tag{2.13}$$

where $p_\mathcal{T}$ and $\dot{p}_\mathcal{T}$ are respectively the position and the velocity in the world reference frame of the ball at time τ, while X_t is the current target location, which lies in the domain described at the beginning of the chapter in eq. (2.2).

The model that was exploited is the *speed-integration* model described in section 1.2.1.1 with the addition of a *stop-integration* condition, which is used to model the ball reaching the target altitude, namely the condition can be expressed in the following way:

$$f_{si}(x_\mathcal{T}) = \begin{cases} 1 & \text{if } p_\mathcal{T} \text{ is higher than current } z_t \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

The eq. (1.24) that describes the states transition need to be adjusted:

$$\begin{aligned} p_{\mathcal{T}+1} &= p_\mathcal{T} + f_{si}(x_\mathcal{T}) \left(T_s \dot{p}_\mathcal{T} + \frac{T_s}{2} \Delta \dot{p}_\mathcal{T} \right) \\ \dot{p}_{\mathcal{T}+1} &= \dot{p}_\mathcal{T} + \Delta \dot{p}_\mathcal{T} \end{aligned} \tag{2.15}$$

The state vector has only 6 elements that describe the evolution of the bullet, the last 3 elements contain constant terms that remain the same for the whole trajectory, the model only requires to predict the velocity changes, therefore 3 GPs must be trained, one for each 3D dimension. As Kernel function we relied to the classical RBF (eq. (1.15)) during the simulations, this choice will be explained once the trajectories will be presented and analyzed. In this kind of application the user might also define a mean function for the GPs, and supposing that the gravity acceleration in parallel to the z axis and opposite in direction, the mean function might be:

$$m(x_\mathcal{T}) = \begin{bmatrix} 0 \\ 0 \\ -gT_s \end{bmatrix} \tag{2.16}$$

where g is the gravity acceleration and $\delta_\mathcal{T}$ is the sampling time. In the final implementation for the simulative trials we just used zero-mean

GPs with RBF kernel, due to the trajectory data being extremely simple.

In this particular case, the control action is only applied at the initial state. To reduce the problem's complexity, the manipulator's action is approximated as the application of the initial velocity of the bullet, with multiplicative noise ψ. The distribution of the initial state x_0 is:

$$x_0 = \begin{bmatrix} X_{rel} \\ \pi_\theta(X_t)(\psi+1) \\ X_t \end{bmatrix} \tag{2.17}$$

which is completely dependent on X_t, since the policy is supposed to learn to handle the task for arbitrary target positions. In particular, X_t can be sampled uniformly from its domain (eq. (2.2)):

$$X_t = \begin{bmatrix} \cos(\text{yaw}_t) \cdot \ell \\ \sin(\text{yaw}_t) \cdot \ell \\ z_t \end{bmatrix}, \quad (\text{yaw}_t) \sim U([-\theta,\theta]), \ell \sim U([L_{min}, L_{max}]) \tag{2.18}$$

where $U([a,b])$ is a uniform random variable defined on the interval $[a,b]$, with $a < b$.

Again, we should redefine the states distribution from eq. (1.20), since we incorporated the control action into the initial state distribution, we *«lose»* the dependency of the state x_{k+1} from the policy action $\pi_\theta(x_k)$

$$p(\hat{x}_{\tau+1}) = \int p(\hat{x}_{\tau+1}|\hat{x}_\tau, \mathcal{D}) p(\hat{x}_\tau) d\hat{x}_\tau \tag{2.19}$$

and what happens in the actual implementation is that the input of the GPs is just x_k and not the extended vector $\hat{x}_k$. Moreover, not all the elements of the state vector x_k are useful to predict the following state, obviously the target X_t does not influence the bullet trajectory, we made the further assumption that the ball's dynamics are not dependent on the current position, in this way the inputs for the GPs prediction can be further reduced to the vector $\dot{p}_k$, i.e. the velocity at time k of the bullet. Reducing the dimension of the input vectors for the GPs is useful to reduce the computational complexity in the prediction of mean and variance in eq. (1.17).

2.2.1.1 Modeling of Tossing noise

From the data collected in the aforementioned experiment (fig. 2.8), it was possible to decently estimate a probability density function of the system's noise, in the experiments we found that modeling the noised tossing as:

$$v_m = v_r\,(1+\psi) \tag{2.20}$$

works quite well, it expresses that the system is effected by an additive noise that is a percentage of the input. We estimated this noise by computing $\frac{v_m - v_t}{v_t}$ for each tossing trial in (fig. 2.8) and presented the resulting density in fig. 2.12. We ignored all the bins related to negative values in fig. 2.12, so ignored the *undershooting* of the system and only considered the *overshooting*, so we fit a Beta distribution over the bins corresponding to positive values. The Beta PDF is:

$$\begin{aligned} Beta(\alpha,\beta) &= \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \\ B(\alpha,\beta) &= \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \end{aligned} \tag{2.21}$$

where $\Gamma(\cdot)$ is the **Gamma** function: $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt$.

The noise is defined as the random variable: $\psi \sim Beta(\alpha,\beta)\sigma$, with $\alpha = 1$, $\beta = 3$ and $\sigma = 0.1$, in this way ψ is constrained in the interval $[0,\sigma]$ and its distribution is plotted in red in fig. 2.12. In fig. 2.13 we compare the samples from fig. 2.8 with the newly defined noise, the red area is plotted between the line of the identity function and the line of the function $v_m = v_t(1+\sigma)$ and represents where the noised samples can fall. For example, consider the vertical line plotted in fig. 2.13 at $v_t = x$, then $v_t(1+\psi) \in [v_t, v_t(1+\sigma)]$, so between the red and black points that are positioned at the intersections of the vertical lines with the edges of the red area. In the next chapter results will show this approximation working quite well. A more precise approach could be to estimate the noise in fig. 2.8 by fitting a curve on the samples.

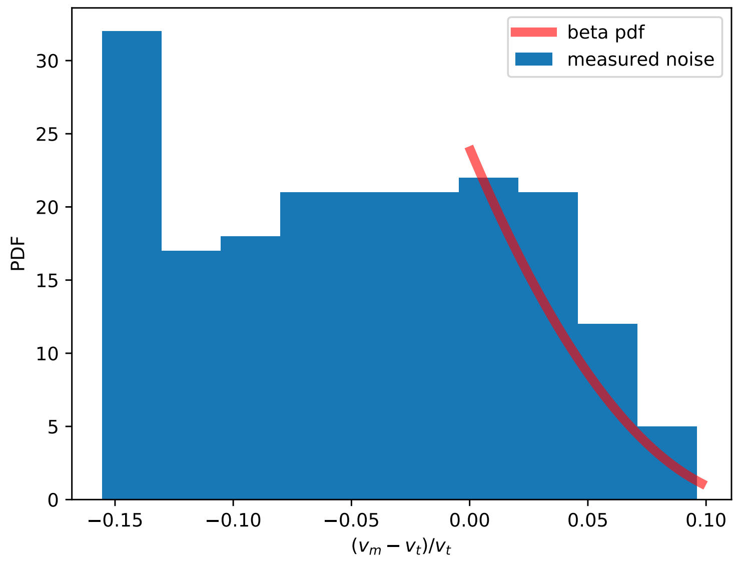

Figure 2.12: The estimated noise of the tossing velocity as percentage of the measured released velocity

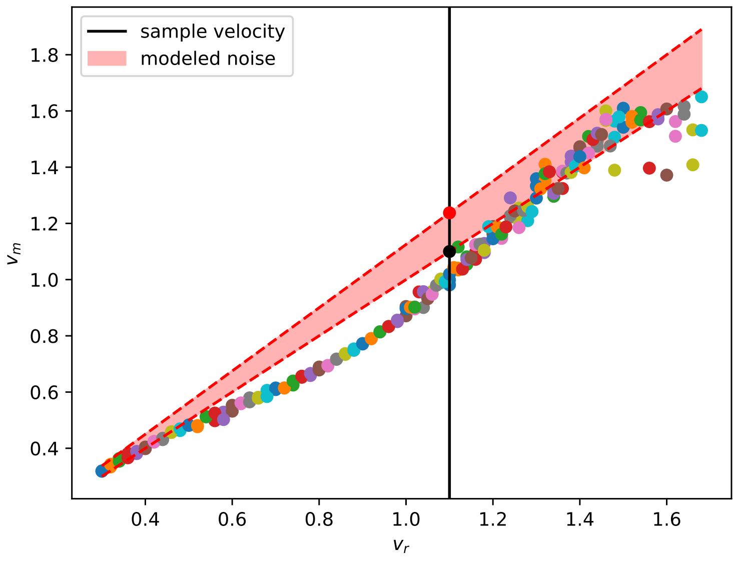

Figure 2.13: Comparing the samples of velocities with the modeled noise

We found that this overall model works well in the experiments that were performed, to a grade that depends from the sampling frequency of the trajectories. As a matter of fact, eq. (2.15) suffers from *drifting*, because the predicted position at time T is likely to have crossed the altitude z_t and might not represent correctly the outcome of the tossing. Higher sampling frequencies result in smaller velocity changes, resulting in smaller drifting.

2.2.2 Cost Function

The definition of the cost, or equivalently reward, function is maybe the most important part in any RL application, since this cost univocally defines the task's objectives in terms that the optimization can *«understand»*. In particular, the cost function must reflect how the policy performance in the task is evaluated, and in our case the most discriminant feature is where the actual landing position is w.r.t. the target bin.

In our case it is fundamental that the cost function must penalize policies that throw the ball far from the target position. Then let us introduce the saturated distance from target state:

$$\begin{aligned} c_{st}(x) &= 1 - e^{-\|p - X_T\|^2_{\Sigma_c}} \\ \Sigma_c &= \begin{bmatrix} \frac{1}{\ell_c} & 0 & 0 \\ 0 & \frac{1}{\ell_c} & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{2.22}$$

with ℓ_c appropriately set considering the possible distances.

The cost function to evaluate the policy is then:

$$J(\theta) = \mathbb{E}[c(x_T)] \tag{2.23}$$

so it is the expected saturated distance of the position at time T from target state, over the horizontal plane.

We can also provide a hint to the gradient of $\hat{J}(\theta)$, which is just the last term of the external summation in eq. (1.42):

$$\begin{aligned} \nabla_\theta \hat{J}(\theta) &= \frac{1}{M} \sum_{m=1}^{M} \sum_{i=1}^{d_x} \frac{\partial c(\boldsymbol{x}_T^{(m)})}{\partial \boldsymbol{x}_T^{(m,i)}} \frac{\partial \boldsymbol{x}_T^{(m,i)}}{\partial \theta} \\ &= \frac{1}{M} \sum_{m=1}^{M} \sum_{i=1}^{d_x} \frac{\partial c(\boldsymbol{x}_T^{(m)})}{\partial \boldsymbol{x}_T^{(m,i)}} \sum_{k=0}^{T-1} \frac{\partial \Delta \boldsymbol{x}_k^{(m,i)}}{\partial x_k} \frac{\partial x_k}{\partial \theta} \end{aligned} \tag{2.24}$$

It is clear that the gradient describes the dependencies of the cost $c(\boldsymbol{x}_T^{(m)})$ from the initial state $x_0^{(m)}$, the only concerning aspect is that with high sampling rates, the dependencies are very deep which results in the gradient being computed as a summation of many partial derivatives (standard derivation rules of Calculus).

This might create issues in the optimization if the result of this sum tends to be too small or too big, this typically leads to the problem of *vanishing/exploding gradient*, which is a known problem of many gradient based optimizations.

In practice we did not find this problem pressing, the reason might be that the model of section 2.2.1 is particularly expressive and the dependencies are clear, moreover the use of particles is certainly providing an intrinsic regularization.

2.2.3 Tossing Policy

It is clear from eq. (2.17) that it required that the policy must be defined in the following way:

$$\pi_\theta : X_t \to v_{toss} \tag{2.25}$$

namely, π_θ must take in input a target position and return the velocity vector that the robot should apply to the bullet. From eq. (2.6) we know that the direction of the velocity vector is constrained by X_t and the parameter α, then the only degree of freedom to decide is the intensity of the tossing, i.e. the norm $\|v_{toss}\|$. Therefore we designed a policy based on eq. (1.28) that does exactly that:

$$\pi_\theta(X_t, \alpha) = \frac{u_{max}}{2}\left(\tanh\left(\sum_{i=1}^{N_b}\frac{w_i}{u_{max}}e^{-\|a_i - X_t\|^2_{\Sigma\pi}}\right)+1\right)\begin{bmatrix}\cos(\text{yaw}_t)\cdot\cos(\alpha)\\ \sin(\text{yaw}_t)\cdot\cos(\alpha)\\ \sin(\alpha)\end{bmatrix} \tag{2.26}$$

where the general purpose policy from eq. (1.28) was adapted to predict only positive values, in this case is used to learn the norm of the vector, i.e. $\|v_{toss}\|$. Note that, w.r.t. eq. (1.28), $w_i \in R$ and $u_{max} \in \mathbb{R}$ is set according to the manipulator's Cartesian limits.

2.2.3.1 Centers initialization

It is known that the performance of SGD-based methods are heavily influenced by the initialization of the optimization parameters [22], meaningful initialization is important to have a good initial exploration during policy update step. The most influential parameters of the policy (eq. (2.26)) are the centers a_i, which in this particular application belong to the 3D real vector space. A good principle for centers initialization is to evenly spread them in the space where the policy inputs lie, but also other principles could be successful and could be investigated. Therefore 3 different centers initialization were considered, whose results will be discussed in the next chapter, the first kind, that will be addressed as *Sparse* initialization:

$$a_i = \begin{bmatrix} c_x \cdot L_{max} \\ 2(c_y - \frac{1}{2}) \cdot d_{max} \\ z_t \end{bmatrix} \qquad c_x, c_y \sim \mathcal{N}(0,1) \tag{2.27}$$

A second initialization, that will be called *Line* initialization:

$$a_i = \begin{bmatrix} d_{max} \cdot c + L_{min} \\ d_{max} \cdot (c - \frac{1}{2}) \\ z_t \end{bmatrix} \qquad c \sim \mathcal{N}(0,1) \tag{2.28}$$

Finally, the last initialization considered, referred as *Focus*:

$$a_i = \begin{bmatrix} L_{min} + \frac{d_{max}}{2} + c_x \\ c_y \\ z_t \end{bmatrix} \qquad c_x, c_y \sim \mathcal{N}(0,\sigma) \tag{2.29}$$

with sigma being a very low value, $\sigma = 0.05$ in the experiments.

2.3 Laboratory Experiments

This section will present the experiments that were performed in DEI's laboratory, it was not possible to implement the whole task, since Franka Emika's gripper lacks the real time control, nor it is possible to synchronize the gripper aperture with the joints trajectory.

Nonetheless, it was possible to collect some data for preliminary model learning experiments and to test a robot tossing trajectory.

We used a red rubber ball of $6cm$ of diameter as bullet, intel RealSense stereo cameras to record its trajectories and, obviously, the Panda Robot.

2.3.1 Collecting data for Model Learning

The model presented in section 2.2 requires only to collect unconstrained motion of the bullet, therefore we prepared a simple setup to collect some random tossing of the bullet. The experiment is composed of:

- The bullet;

- At least one stereo camera;
- One apriltag [44];
- Two human operators;

and it is performed in the following way[8]:

- The camera is placed in front of a scene, possibly of uniform color, such that the ball can be distinguished from the background in the images;
- The apriltag is placed on the ground, face up, such that it can be seen clearly from the camera and the z axis of its reference frame is vertical.
- The two operators get into position to the edges of the field of view (FOV) of the camera;
- The operators start tossing the ball to each other while the camera is recording.

To reduce the motion blur of the moving bullet, it was necessary to set the exposure time to its minimum value, this allowed, in the collected images, to reduce deformations of the bullet shape to minimum.

This setup clearly does not require that the robot is used, and this allowed to test the performance of the Model learning without even needing to operate the manipulator. This serves also as an example of how MB techniques might be very appealing in industrial robotics applications, in this case the tossing policy can be trained using just data collected very efficiently by the operators, resulting in a minimal preliminary wear of the robotic manipulator.

To extract the 3D positions of the bullet, in the camera frame, during the trials performed by the operators the following steps were performed:

- The video is divided in smaller sequences of RGB images, each sequence showing a single throw, for each of these images the corresponding aligned depth map and time step are saved. The depth map is a matrix of the same dimensions of the image and aligned with the image, meaning that the RGB image and the depth map are expressed in the same reference frame.

[8]a recording of the experiment is available here `https://drive.google.com/file/d/1_MgQnvLw60ORCRMA2dpCD7dVVzd__2Cf/view?usp=sharing`

- The RGB images must be pre processed in order to extract the ball from the background, therefore we implemented a simple procedure, exploiting the OpenCV library [45], to build a binary mask of the bullet. This procedure is composed of:
  - Estimate the RGB color vector of the ball, this was done simply by sampling the color of a group of pixels that represent the ball's surface. The location of the ball surface on the image plane is picked by the user with a click on the right location and the software returns the average color of a neighborhood of pixels around the clicked pixel;
  - All the pixels in the image that are further that a certain distance in the color space from the estimated bullet color are set to zero, otherwise they are set to the maximum value;
  - Gaussian filtering is applied to the binary image obtained from the previous step to eliminate most of the outlier pixels, this works until the outliers are small groups of pixels not connected to the bullet pixels;
  - All the pixels in the gray-scale image obtained from the previous step are set to the maximum value if their value is different from zero.

  The resulting binary image, representing a binary mask, in the best performing scenario contains white pixels in the correspondence of the bullet location in the image frame, while all other pixels are black, but typically the binary mask also contains other objects. For example in presence of particular lighting, the operators' skin can also be selected by the mask, depending on how the parameters are tuned;

- Next, for each binary image from the previous step, the software estimates the position of the ball in the image plane using the blob tracking of the VISP library [46], it is just necessary to locate the blob corresponding to the ball in the first image of the sequence. This can be done manually by the user by clicking in the blob's location, or alternatively it could be automated if the image contains just the mask of the bullet. In practice, most of the binary images contain several other objects, but in most cases they are not connected to the ball's blob, therefore the blob of interest can be tracked until it remains in the FOV of the camera or it gets too close to other blobs. This step is the most susceptible to the sampling frequency of the camera, if the blobs in subsequent samples are too far apart, the tracking fails. The process was employed on images captured at 60 Hz and it worked successfully.

- The 3D positions of the ball in each image can be retrieved, ignoring lens distortions, by considering the center of mass (COM) of the blob in the image plane and the depth value in the corresponding depth map with this simple formula of the pinhole camera model [47]:

$$p_i = d_i \begin{bmatrix} \frac{u-u_c}{\alpha_u} \\ \frac{v-v_c}{\alpha_v} \\ 1 \end{bmatrix} \tag{2.30}$$

  where d_i is the depth value, (u,v) are the coordinates of the blob's COM in the image reference frame and $u_c, v_c, \alpha_u, \alpha_v$ are the camera's intrinsic parameters. To robustly estimate the 3D position, for each blob position, d_i is computed by averaging the values of depth in certain neighborhood of the estimated position in the 2D frame of the depth map.

This procedure works good up to a certain extend, provided that the ball is in the camera's depth range, but it typically outputs some outlier points, therefore in order not to loose too much data, it was necessary to implement an outlier rejection technique. The two most common types of outliers are:

- Points whose depth value is too little w.r.t. the real known scene, the depth map registers these points being very close to the sensor, in the order of tens of centimeters, this is impossible since in the experiments, both operators were at least further than 1 meter from the camera. To reject these points, the software discards points whose z coordinate in the camera frame is lower than a certain user-defined threshold;

- Points that lie clearly in a wrong spot due to some noise in the depth map, these are typically far from the other samples in an horizontal direction. To counter these effects we implemented a rejection method that fits a plane to the trajectory points with the RANSAC Algorithm [48] and rejects points that are further than a certain distance threshold from the plane. This approach works good in practice since the unconstrained motion results in a trajectory in the shape of a parabola, which lies on a plane in space.

The methods are, in gerenal, applied in the order that they were just described.

One example of outliers of the first kind is shown in fig. 2.14, while an example of outliers of the second type can be seen in fig. 2.17, where outliers in red have been removed from the trajectories.

Once the outliers are removed from the trajectory they leave *«holes»*, then the choices are to return in output the smaller trajectories that are created, or to *«fill the holes»* in some way. The sampling frequency of the camera is known, and it was verified to be very consistent, but most MB algorithm perform re-sampling in any case to obtain samples with constant sampling times, therefore also in this case it was decided to re-sample each trajectory using the the inliers and interpolating each dimension using time as the independent variable. This has the effect of recovering the points that were lost as outliers.

The last necessary step is to express the trajectories in a global reference frame, such that its z axis is perpendicular to the ground. This is necessary since it is expected to find dynamics effects in the z axis elements, such as the gravity pull and frictions, while the x and y elements should not present such effects. With these considerations in mind, a Model obtained by trajectory data expressed in this reference frame can be directly used in the robot's reference frame, which also has z axis perpendicular to the ground.

Therefore we used the apriltag placed on the ground to determine a global reference frame, using apriltag's ROS library[9] it is possible to estimate from an image the transformation from the reference frame of the camera to the apriltag's frame, which will be called *ground* reference frame. To robustly estimate this transformation it was necessary to estimate an average transformation from a sequence of images, we estimated translation and rotation in terms of a 3D vector and quaternion from the images of a portion of a video and computed an average of both. The final transformation is obtained as:

$$T_g^c = \begin{bmatrix} R\left(\frac{1}{N}\sum_{i=1}^{N} q_i\right) & \frac{1}{N}\sum_{i=1}^{N} v_i \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} \tag{2.31}$$

where q_i and v_i are respectively, the translation vector and quaternion of the transformation measured in the i-th image, while $R(q)$ expresses

[9] `http://wiki.ros.org/apriltag_ros`

the rotation matrix computed from the quaternion q.

Then, all trajectories expressed in the camera reference frame can be brought in the *ground* reference frame by applying T_c^g.

The velocities along the trajectory can then be computed by means of acausal numerical differentiation of the positions, specifically, we applied the central difference approximation, namely, the velocity along one direction at time τ is approximated with $\dot{p}_\tau = (p_{\tau+1} - p_{\tau-1})F_s/2$, where F_s is the camera sampling frequency.

In fig. 2.15 and fig. 2.18 it is possible to view the velocity plots corresponding respectively to the trajectories in fig. 2.14 and fig. 2.17 in the two frames. In each velocity plots it is possible to see that one dimension is clearly accelerated, for both examples it is possible to say that the dimension y is clearly the most accelerated when considering trajectories in the camera reference frame, this can be explained by the fact that the y axis of the camera frame is almost vertical.

Instead, for the trajectories expressed in the *ground* reference frame it clear that the z direction is the accelerated one, which is the expected outcome.

In fig. 2.16 and fig. 2.19 it is possible to observe the boxplots of the velocity deltas of, respectively, the trajectories in fig. 2.14 and fig. 2.17, expressed in both reference frames. These statistical plots show that trajectories expressed in the camera frame have deltas of x and z elements close to zero in average and with high variance and deltas of the y element higher than zero in average and with smaller variance.

By contrast those plots expressed in the *ground* frame show that x and y deltas are, in average, close to zero with high variance while the deltas of the z component have an average negative value with smaller variance.

These last observations are very important, since the velocity deltas are the targets of the GPs in the model presented in section 2.2.1, and this indicates that the data that will be forwarded to the model are coherent with what was seen in the simulations (figs. 2.9 and 2.10).

2.3.2 Control of Robot Tossing

The trajectory generation for robot tossing presented in section 2.1.1 is not practically applicable to a real Panda robot, because of the issues mentioned in the same section, therefore it was necessary to define a different approach.

The critical aspect of trajectory generation for a manipulator is

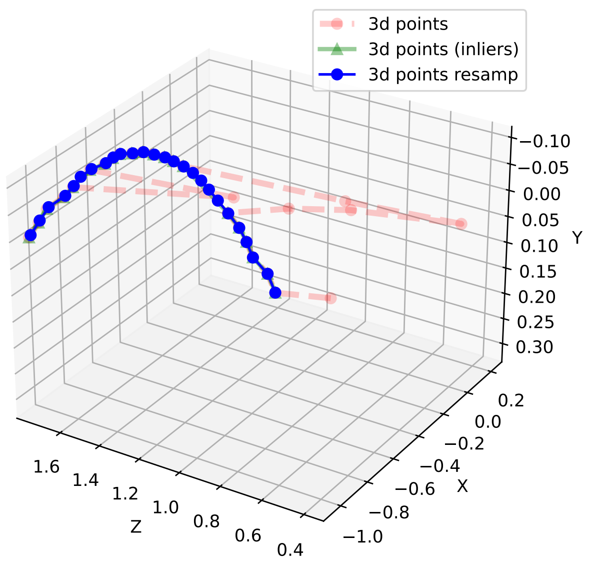

Figure 2.14: Example of a trajectory collected during the laboratory experiments, the red points are outliers rejected because of being too close to the camera plane. This trajectory is expressed in the camera frame.

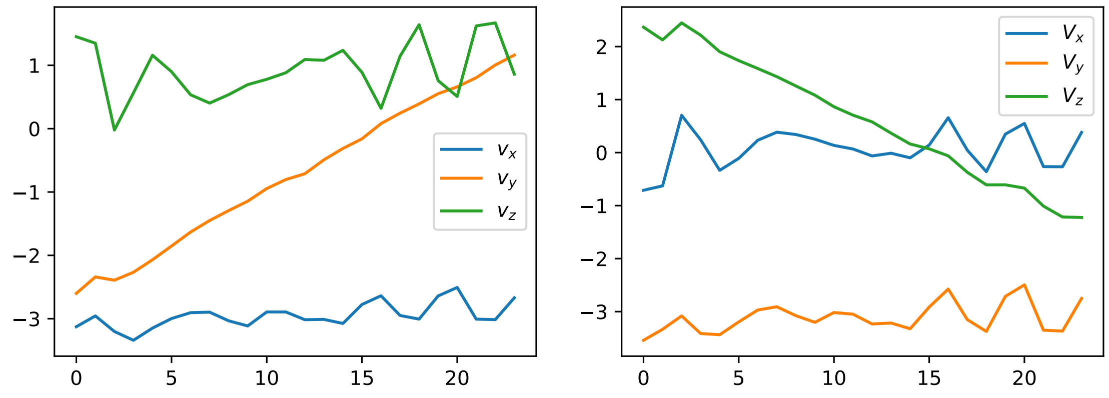

Figure 2.15: [a,b] Comparing velocities of the trajectory in fig. 2.14, (a): w.r.t. camera reference frame, (b): w.r.t. ground reference frame.

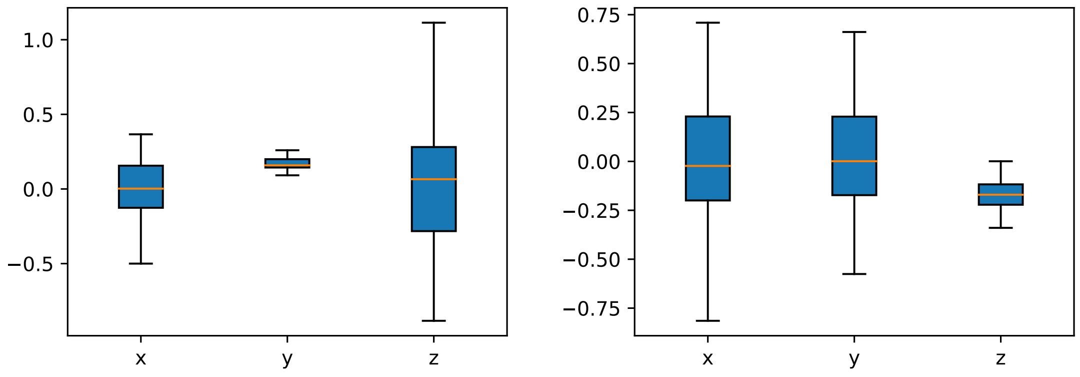

Figure 2.16: [a,b] Comparing velocity deltas of the trajectory in fig. 2.14, (a): w.r.t. camera reference frame, (b): w.r.t. ground reference frame.

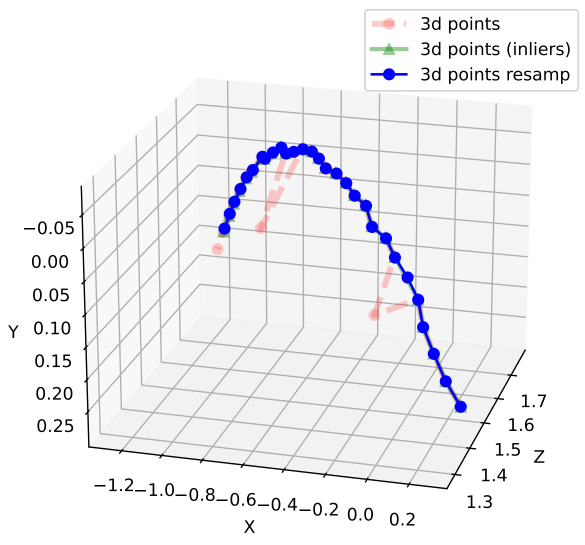

Figure 2.17: Example of a second trajectory collected during the laboratory experiments, the red points are outliers rejected because of being too far from the plane fitted with RANSAC. This trajectory is expressed in the camera frame.

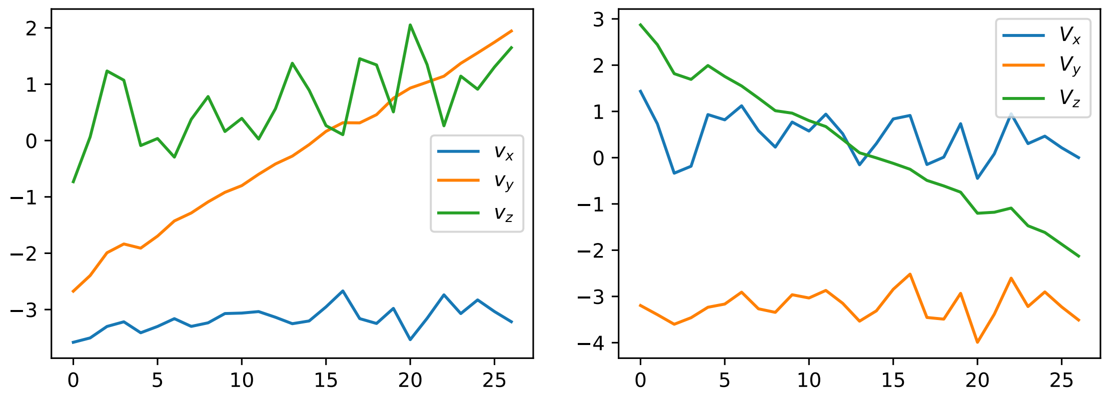

Figure 2.18: [a,b] Comparing velocities of the trajectory in fig. 2.17, (a): w.r.t. camera reference frame, (b): w.r.t. ground reference frame.

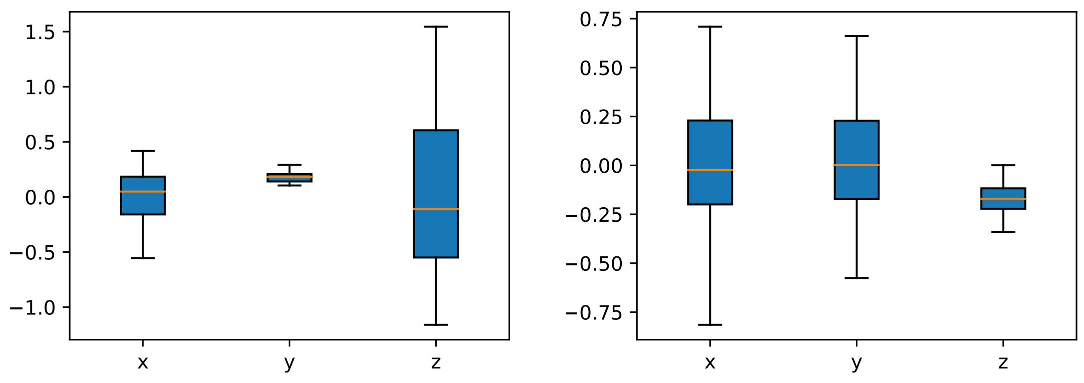

Figure 2.19: [a,b] Comparing velocity deltas of the trajectory in fig. 2.17, (a): w.r.t. camera reference frame, (b): w.r.t. ground reference frame.

to handle acceleration and deceleration, it is required to generate a trajectory that has starting, as well as stopping, zero acceleration and velocity.

The requirements for this trajectory are the same of section 2.1.1, the end-effector needs to reach a release point with velocity vector v_{toss}, as explained in the first pages of this chapter.

Contrary to what has been done in section 2.1.1, the tossing trajectory in this case is generated in Cartesian space, consider:

$$\begin{aligned} X(\tau) &= X_0 + p(\tau) \begin{bmatrix} \cos(\text{yaw}_t) \cdot \cos(\alpha) \\ \sin(\text{yaw}_t) \cdot \cos(\alpha) \\ \sin(\alpha) \end{bmatrix} \\ p(\tau) &\in \mathbb{R} \end{aligned} \tag{2.32}$$

$X(\tau)$ is the Cartesian trajectory point at time τ, and the trajectory lies on a line that is parallel to the direction of a v_{toss} aligned of yaw_t radiants w.r.t. the z axis and aligned α radiants upwards w.r.t. the horizontal plane. Once the parameters yaw_t and α are defined, the the trajectory $X(\tau)$ is entirely dependent on $p(\tau)$, which can be used to regulate the necessary properties, namely $p(\tau_0) = \dot{p}(\tau_0) = \ddot{p}(\tau_0) = 0$, $\dot{p}(\tau_1) = \|v_{toss}\|, X(\tau_1) = X_{rel}$ and $\dot{p}(\tau_2) = \ddot{p}(\tau_2) = 0$, with τ_0 being starting time, τ_1 being release time and τ_2 stopping time.

Let us define A_m and J_m being respectively, the maximum Cartesian acceleration and jerk, then $T_1 = \frac{A_m}{J_m}$ is the time required to reach maximum acceleration with jerk J_m and $T_2 = \frac{\|v_{toss}\|}{A_m} - T_1$ is the time required to reach velocity $\|v_{toss}\|$ from the moment acceleration A_m is reached. Then, the trajectory jerk can be defined as in fig. 2.20 and acceleration a, velocity v and position p can be obtained by integrating the underlying plot. The trajectory is defined such that $X(2T_1 + T_2) = X_{rel}$ and $\dot{p}(2T_1 + T_2) = \|v_{toss}\|$, in fig. 2.20 the black vertical line is the instant $\tau = T_1$ when acceleration A_m is reached, while ther red vertical line is $\tau = 2T_1 + T_2$, when velocity $\|v_{toss}\|$ is reached.

The Cartesian starting point is defined as:

$$X_0 = X_{rel} - p(2T_1 + T_2) \begin{bmatrix} \cos(\text{yaw}_t) \cdot \cos(\alpha) \\ \sin(\text{yaw}_t) \cdot \cos(\alpha) \\ \sin(\alpha) \end{bmatrix} \tag{2.33}$$

and the release point was redefined as:

$$X_{rel} = \begin{bmatrix} r\cos(\text{yaw}_t) \\ r\sin(\text{yaw}_t) \\ 0.1 \end{bmatrix} \qquad r = 0.6 \tag{2.34}$$

in the robot's base frame. Once the Cartesian trajectory is defined, it is necessary to *«convert»* it in joints space, it can be done with inverse kinematics, but in this case the *Moveit!* interface does explicitly allow to access the IK plugin, but it does allow to compute the Jacobian matrix to any configuration (section 2.1.1). The possible choices are then two: (i) rely to an IK ROS plugin or to additional sofware, like PyBullet [49]; (ii) to use differential kinematics equations. In this case it was chose to rely to the second approach, namely we move the robot to the first Cartesian position X_0 of the trajectory, with elbow up and gripper in a vertical position, read the current joint configuration q_0, the rest of the trajectory in joint space can be obtained by integrating velocity:

$$q(\tau+1) = q(\tau) + J^{\dagger}(q(\tau))v(\tau)\begin{bmatrix} \cos(\text{yaw}_t)\cdot\cos(\alpha) \\ \sin(\text{yaw}_t)\cdot\cos(\alpha) \\ \sin(\alpha) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.35}$$

in this way the trajectory keeps the end-effector at the same orientation, thus forwarding no angular velocity to the bullet.

In addition, the trajectory in fig. 2.20, is also designed to stop some centimeters after reaching the release point and to return to the starting point following the same trajectory backwards[10]. In fig. 2.21 the same trajectory in Cartesian coordinates is shown, the plots show the x, y, z components of the trajectory expressed in the robot's base frame, and compare the reference trajectory with the actual end-effector positions. Fig. 2.22 presents the same end-effector trajectory is 3D space, the 3D plot compares the reference line with the actual measured trajectory and the release point.

[10]a small recording of the robot performing said trajectory is available at `https://drive.google.com/file/d/18RCKL80oE8IzjnWqeDANHEhU3pVLE22Z/view?usp=sharing`

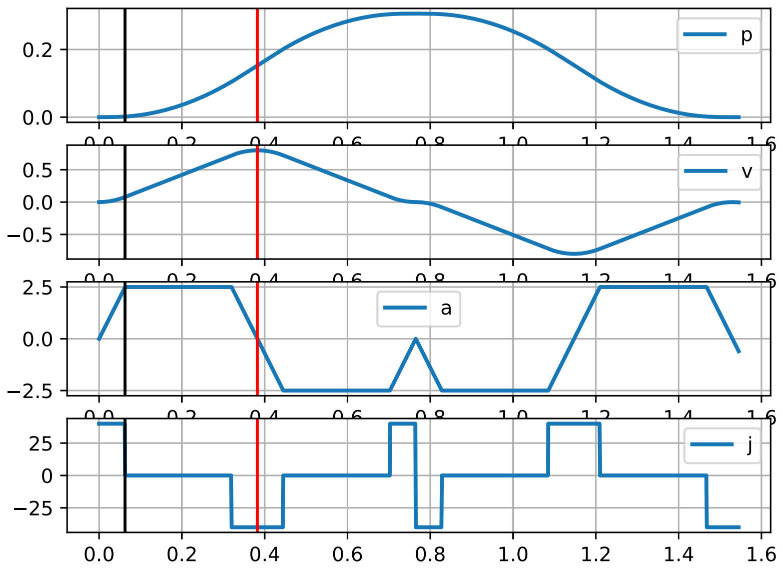

Figure 2.20: Example of Cartesian trajectory generation with target $\|v_{toss}\| = 0.8m/s$

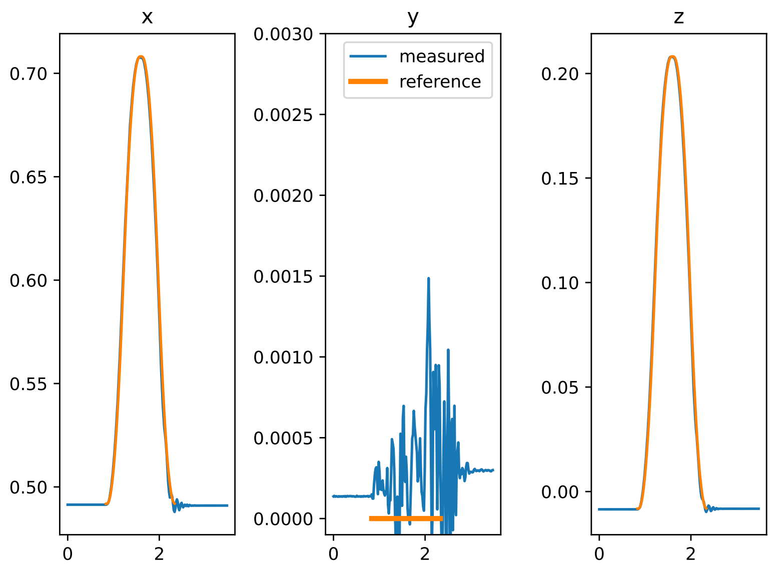

Figure 2.21: Example of a tossing trajectory in Cartesian space for the physical Panda robot, created from fig. 2.20. The plot is divided in its x, y, z components. The orange line is the reference trajectory, while the blue line is the measured end effector trajectory. The tossing orientation is yaw$_t = 0$ and $X_{rel} = [0.6,\ 0.0,\ 0.1]^T$ in the robot's base frame. The RMSE of the end-effector trajectory tracking in this case is 0.0026 m.

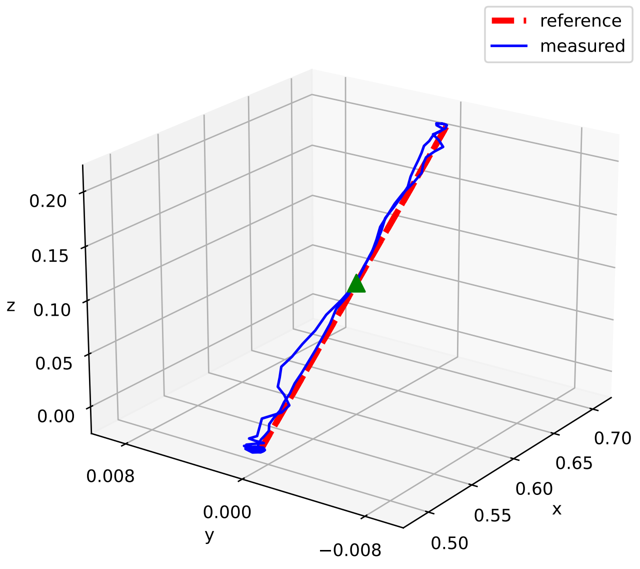

Figure 2.22: Example of a tossing trajectory in Cartesian space for the physical Panda robot, the red dashed line is the reference trajectory, the blue line is the actual measured Cartesian trajectory of the end effector and the green triangle is the release point. It is the 3D plot of fig. 2.21

Chapter 3

Results

This last chapter will present all the results obtained until now, including some achieved in DEI's laboratory, and is organized as follows:

- section 3.1 will present the most relevant results obtained by MC-PILCO, adapted as described in section 2.2, in the simulated task. Specifically, section 3.1.1 will show the performance of the model to predict the bullet trajectories and landing position, while section 3.1.2 will present all the results of policy learning obtained in different learning setups and the performances in simulation of the policies obtained with each setup.
- section 3.2 will show the preliminary results obtained with the bullet data collected in the laboratory experiments, explained in section 2.3.1.

3.1 Simulation

All the results discussed in this section were performed with maximum horizontal target distance from release $d_{max} = 50cm$ and maximum yaw orientation $\theta = \frac{\pi}{6}$, referring to the start of chapter 2, figs. 2.1 and 2.2. Moreover, the sampling frequency for both trajectory resampling and prediction with the model was set to $F_s = 50Hz$, which was enough to train a good policy for the target distances that were considered. As explained in section 2.2.1 the model suffers from *drifting* which is reduced with higher sampling frequency, this comes at the drawback of needing to simulate more samples to simulate the same interaction time. However, in this particular instance, the velocity norm of a particle, during the rollouts of policy update, inevitably accelerates and some drifting at the simulated landing is inevitable unless the sampling frequency is ridiculously high compared to the considered target distances.

See for example fig. 3.4, where the plot on the right shows the altitude of the last position of 400 particles

In addition, $F_s = 50Hz$ is a more than reasonable setting for the trajectory sampling, since most 3D cameras have a sampling rate of around $50/90Hz$.

Lastly, it is important to set a sufficient system interaction time for the rollouts in the policy update step, particularly for this application, if a particle does not cross the target altitude, then it's not possible to estimate the landing position and the cost presented in section 2.2.2 does not correctly estimate the policy performance. In all policy training performed in simulations the maximum interaction/trajectory time was set to $1s$, which leads to rollouts of 50 samples, then trajectories reach target altitude much before the maximum time, like in fig. 3.1, but that does not changhe the dependencies of the particles' last position from the policy, given that the cost only depends on the final position, not velocity.

3.1.1 Model learning

The first requirement to check when using any Model-Based RL technique is the performance of the Model of predicting the correct evolution of the controlled system, therefore in fig. 3.1 we provide an example of simulated trajectory, that is correctly predicted by a model trained on another trial. We find that in average, for the model presented in section 2.2.1, the predicted trajectory distances itself from the real trajectory in the order of millimeters. The same figure shows, on the right, the velocity norm thoughout the simulated trajectory, which is compatible with the plot in fig. 2.9.

Moreover, some more intensive testing was performed on the model of section 2.2.1 on a series of simulated trials, which are presented in fig. 3.2. The test was performed by training the GP models on the data of a single exploration trial and then testing on new trials by computing: (i) MSE of the GPs on the targets from the new trajectory; (ii) MSE of the rollout prediction over the new trajectory; (iii) distance over the horizontal plane of the predicted landing distance (position of the last state of the particle) from the actual target. With each new trial, the model is retrained using all accumulated experience, just like what happens with MC-PILCO when the outer loop of algorithm 1 is repeated.

In fig. 3.2 one can see that the performances of the GPs regression are not constantly the same, which is somewhat expected, but the

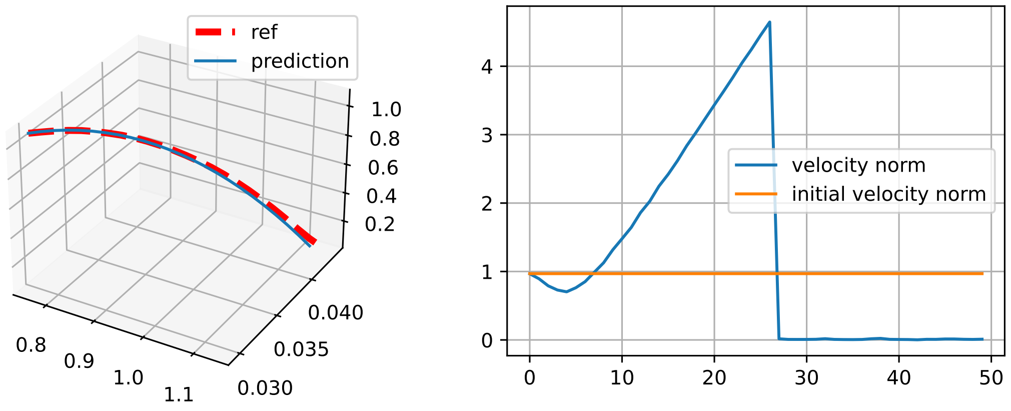

Figure 3.1: [a,b], (a): Example of a rollout (simulation) on a trajectory not used for training the model. MSE over the trajectory: 0.007 m, Error of predicted landing: 0.015 m. (b): the velocity norm along the trajectory compared to the initial velocity, which drops to zero once the particle has reached the *stop-integration* condition.

performances of rollout prediction and, more importantly, landing prediction do not change change significantly with more trials. These results are justified by the fact that the trajectories obtained in simulation are very simple, and as already anticipated, the policy update step is successful with the model trained on just one exploration toss (one single execution of the outer loop of algorithm 1).

3.1.2 Policy learning

As shown in sections 2.1 and 3.1.1 the trajectories provided by the simulation are very simple (for example see figs. 2.9 and 2.10), and the Model is able to predict the landing correctly (see for example figs. 3.1 and 3.2), therefore the policy update step of MC-PILCO manages to train an optimal policy with the model trained just with one single exploration trial. Taking advantage of the particularities of this task, it was possible to perform a series of different policy training, under different conditions, mainly to show the beneficial effects of the application of dropout in the policy optimization, as well as the modeled control noise, comparing different setups.

As it will be shown in the next pages, it is indeed possible to visualize and interpret a lot of information regarding the policy described in section 2.2.3 and how it evolves during training. Namely, by considering that the target bin is only moved in horizontal direction and its height is constant, it is possible to plot its output norm on a 2D heatmap, representing the velocity norm that the policy enforces for a certain target location. Moreover, it is possible, with the same assumption to plot the centers a_i of the policy function (eq. (2.26)), weighted by

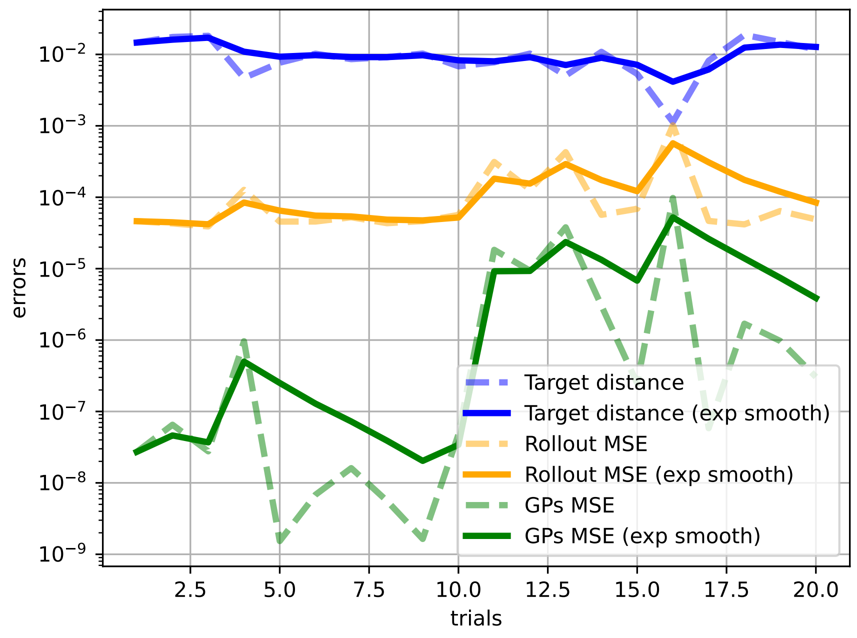

Figure 3.2: Model learning test on 20 simulated throwing trials, starting with the model trained on the data of one single exploration trial, for each test trial of the plot, the model is tested by means of: (i) distance over the $x - y$ plane of the predicted target from the actual landing; (ii) MSE of the Rollout prediction over the trajectory; (iii) MSE of the GPs on the targets. For each error computed in this test its curve is plotted (dashed) in comparison with its exponential smoothing (continuous) computed with $\alpha = 0.5$.
At each trial, after testing the Model, the GPs are retrained with the new data.

their weights w_i, on a 2D grid.

Both information can give an idea of how the policy evolves during the training steps, therefore we performed the same training with identical parameters and models for different setups, varying the centers initialization (section 2.2.3.1), the tossing noise (section 2.2.1.1) and the application of dropout (section 1.2.2.3).

The tested setups are described in table 3.1 and for each one of them we provide, in the next pages a *learning plot* and a *performance plot*. The *learning plots* show in order:

- Total rollout cost plot, shows for each epoch the mean cost between all simulated particles, if the algorithm was executed with dropout applied then vertical lines are plotted in corrispondence of the epochs were the dropout rate was reduced, as described in section 1.2.2.3;

- Success rate of the simulated particles rollout in each epoch, computed by considering the distance along the horizontal plane of the last position of each particle, a rollout is considered a successful throw if said distance is lower or equal than the nominal radius of the target bin;

- The Policy output presented as a 2D heatmap, one plot each 250 epochs;
- The Policy centers plotted on the 2D horizontal plane at height $z_t = 0.1m$ (all points have same height), the points colors are based on a scale defined w.r.t. their associated weight.

The *performance plots* instead show the outcome of 100 tossing trials performed in the Gazebo simulation with the trained policy, the plot shows the 3D position of the targets, colored in red it the target was missed by the policy, green if the target was reached succesfully. The release point at $\text{yaw}_t = 0$ is also presented for convenience of interpretation.

In addition to the tossing noise proposed in section 2.2.1.1, we also tested some alternatives, just to check if the assumptions taken in section 2.2.1.1 were meaningful. Therefore we considered the baseline case without control noise applied, a standard zero-mean noise of std $\sigma = 0.05$ and *negative* control noise, where the initial velocity imposed to each particle in policy update is: $v_0 = \pi_\theta(X_t, \alpha)(1 - \psi)$

The table contains also the performances obtained by the policy, in terms of success rate[1], on a test of 100 trials simulated with Gazebo (section 2.1). A policy is considered optimal if it provides success rate $> 95\%$, while it is considered sub-optimal for success rate $\in [80, 95]\%$, for success rate $< 80\%$ the policy is considered not satisfactory, i.e. a failure.

Consider that, as a baseline, a policy that follows standard ballistics rules reaches unsatisfactory performances: 72% target reach. The policy function is the following:

$$\pi_g(X_t) = \sqrt{\frac{gd^2}{d + h}} \tag{3.1}$$

where h is the difference between the z coordinate of the release and the target altitude, while d is the distance of the target from release in the horizontal direction.

[1]a toss is considered successful when the bullet reaches the bin, i.e. it enters the bin, eventually hitting the edges.

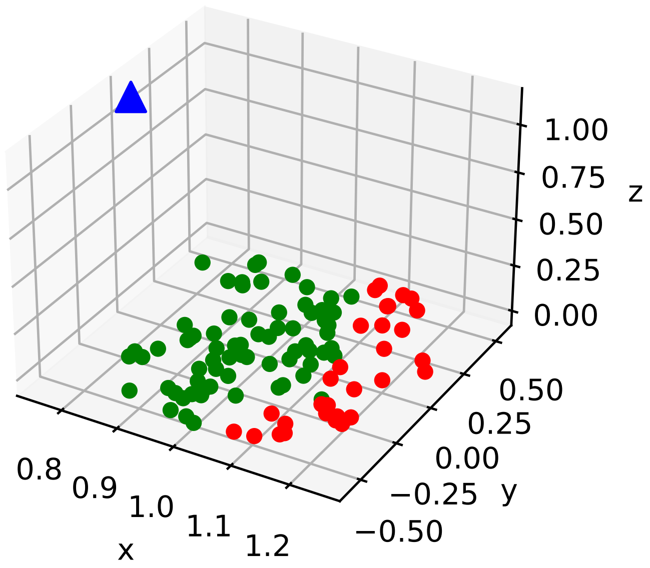

Figure 3.3: Policy derived by ballistics equations, 72% target reach. Test performed with 100 throws simulated in Gazebo, Points represent the target location of one throw, red points are missed targets, green points are reached targets. The blue triangle is the release point when the target yaw is zero, placed as a reference.

All tests presented in this section were performed with 400 particles in the policy update step and 100 centers.

Setup	Centers initialization	Modeled control noise	Dropout applied	Performance
1	Sparse	None	Yes	83%
2	Sparse	zero-mean (std=0.05)	Yes	86%
3	Sparse	overshooting	Yes	100%
4	Sparse	undershooting	Yes	67%
5	Sparse	overshooting	No	91%
6	Line	overshooting	Yes	98%
7	Line	overshooting	No	24%
8	Focus	overshooting	Yes	98%
9	Focus	overshooting	No	93%

Table 3.1: Tested policy update setups. Green: optimal policy; Yellow: suboptimal policy; Red: not satisfactory policy.

3.1.2.1 Noise models

To verify the virtue of the tossing noise model presented in section 2.2.1.1 we compared the execution of the algorithm on 4 setups, reported in the first 4 rows of table 3.1, varying only the noise model. Said rows are color-encoded based on the outcome, the first setup can be considered as a baseline, since it was performed with no modeled

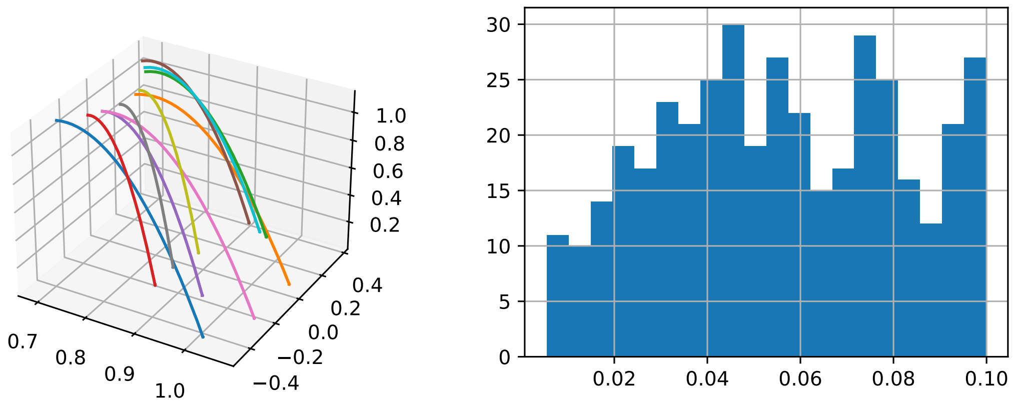

Figure 3.4: [a,b], (a): some examples of particles rollout simulated during a policy training, (b): the height of the final position reached by 400 simulated particles during an optimization step of policy update, presented as histograms of occurrences, this highlights the drifting problem

tossing noise, i.e. in the simulated rollouts computed during the policy update step, the initial velocity of each particle is exactly the vector returned by the policy (section 2.2.3). The Second setup was performed with a simple zero-mean Gaussian noise model: $\psi \sim \mathcal{N}(0, \sigma^2)$ with $\sigma = 0.05$, its performances are almost equal to the baseline, which is the expected outcome, given that the policy update step of MC-PILCO computes the policy gradient as average of many particles and the expected value of ψ is this case would be zero. In figs. 3.5 and 3.7 it is possible to observe the learning plots of these first two setups, which are almost identical in all the plots, in particular the centers of the two policies reach very similar distribution.

Instead, figs. 3.6 and 3.8 show the performance plots of the two policies, these plots show the same pattern, both policies struggle to reach the most distant targets, issue that is solved by applying a correct noise model.

A zero-mean noise model is clearly not adequate for such system, if we look back at fig. 2.12 the most obvious alternative could be to estimate a probability density function from that data. That would lead to another wrong noise model, since such PDF would not account for the relationship between the system behavior and system input, that is instead evident in fig. 2.13.

The learning plots in figs. 3.9 and 3.11 show respectively the policy update steps performed with *overshooting* and *undershooting* tossing noise, both these executions are longer in terms of number of epochs in the optimization process, w.r.t. the first two setups. This suggests that the overall dynamics models used in these optimizations are, in some way, harder to learn for the policy. The centers distribution in these setups does appear somewhat different w.r.t. the first two setups, but it is does not look as if different patterns are realized.

Most importantly, fig. 3.10 does show that the policy trained wit *overshooting* noise (described in section 2.2.1.1) reaches optimality and perfect performance, this indicates that the modeled noise describes the system behavior up to a certain margin that allows the policy to perform the task correctly.

Instead, fig. 3.12 shows the performance of the policy trained with *undershooting* tossing noise, it is noticeable that this policy completely misses the distant targets, as discussed in section 2.2.1.1, while it is possible to approximate the tossing system to the identity function for small velocities (see fig. 2.8), that is not possible for higher velocities and for these it is necessary to account for the correct functioning.

3.1.2.2 Dropout effects on policy learning

From the previous discussion, we can assume that the system is sufficiently well described by the noise model presented in section 2.2.1.1 and it can be used for all the remaining setups.

Consider the rows $3, 5, 6, 7, 8.9$ of table 3.1, which present the performance of the policies trained with the different initializations presented in section 2.2.3.1, for each of these initializations the algorithm was executed with and without the application of dropout (section 1.2.2.3).

The first, most obvious, aspect to notice about these setups is that the executions with dropout all reached optimal policy performance, while the executions without dropout reached suboptimal or not satisfactory policies.

Consider then rows 3 and 5, which present the performance of the policies trained with *sparse* center initialization, their learning plots are reported respectively in figs. 3.9 and 3.13. Between these figures it is possible to appreciate some distinctive features, namely:

- the execution without dropout is faster, the training reaches convergence in less epochs;
- the rollout cost plot in fig. 3.13 clearly presents some spikes in later stages of training, which do not appear in fig. 3.9 with the application of dropout. Each cost spike corresponds to a performance drop in the underlying success rate plot;
- the scale of the policy output heatmaps in fig. 3.13 is slightly different from the respective maps in fig. 3.9, the top value, in the same domain, is higher for the policy trained without dropout.

- The centers distributions are different between these two executions, both during the optimizations and in the late stages, in particular the centers in the execution without dropout do not move much outside the initialization area.

This last observation in particular is indicative of the fact that the execution without dropout does not perform the same exploration in the parameters space that is instead performed with the application of dropout. Indeed, each epoch of the policy update step, updates the policy following SGD approach, in this way the parameters are *«moved»* in the direction suggested by the gradient. To update the policy centers in this way directly corresponds to move their location, and in this particular case, since the target locations do not move along the z direction, the centers altitude does not change, they only move in the horizontal plane. If the optimization process does not explore a good portion of the parameter space, then the probability of finding a global minima inevitably decrease, this is confirmed by the fact that the policy trained without dropout is suboptimal (fig. 3.14).

The effects of dropout application have also been tested with other two kind of initialization, setups 6 and 7 refer to the Line initialization, where centers are positioned along a segment, respectively executed with and without dropout applied. In this particular case, the optimization executed in absence of dropout does not reach convergence, it remains stuck at the initial configuration, this could be due to the fact that the initial configuration was a local minima from where it was not possible to escape with gradient updates. Instead, the execution of MC-PILCO with setup 6, with the application of dropout the optimization is able to escape the local minima in few epochs, then, the rollout cost, success rate curves and policy output heatmaps in the learning plot of fig. 3.15 are comparable to the learning plots of setups 3 and 4 (figs. 3.9 and 3.13), as well as the number of epochs, while the distribution of centers for setup 6 is visibly different from previous executions. In particular in fig. 3.15 it is possible to appreciate the evolution of the centers distribution from the initial shape to a more expanded configuration.

Lastly, setups 8 and 9 consist of executions with the Focus centers initialization, also in this case the policy trained in presence of dropout is result of much more exploration, the optimization process that is described in fig. 3.19 was able to explore a much larger portion of the parameters space, comparing the centers distributions of figs. 3.19 and 3.21 it is possible to see that in the latter the centers did not stray far from the initial configuration. Moreover the cost and success rate plots of fig. 3.21 present the same spikes/drops of fig. 3.13 that never appeared in these tests with the application of dropout.

From a performance point of view, setups 8 and 9 have the same pattern of setups 3 and 5, in both cases the optimizations that exploited the dropout were able to explore more of the parameters space, landing in a final optimal configuration. Indeed no execution without dropout managed to reach the regions of the parameters space where the optimal policies parameters configurations lie.

Interestingly, in the learning plots associated with the setups that applied dropout in the optimization process, it is possible to recognize a visual effect in both the cost and success rate curves, indeed each time the dropout reduction is applied (vertical lines in the plots) the noise of the curve is visibly reduced. The learning plot of fig. 3.21 presents this same effect more evidently.

In all dropout setups the reductions were applied in late stage well after the initial exploration in parameters space, this happens because the cost decreased under a certain threshold and remained stable for a number of epochs. Instead the policy update executed with setup 8 (fig. 3.19) applied the first dropout reduction at early stage, this was due to the high level of variance of the cost signal $\hat{J}$ at early stage.

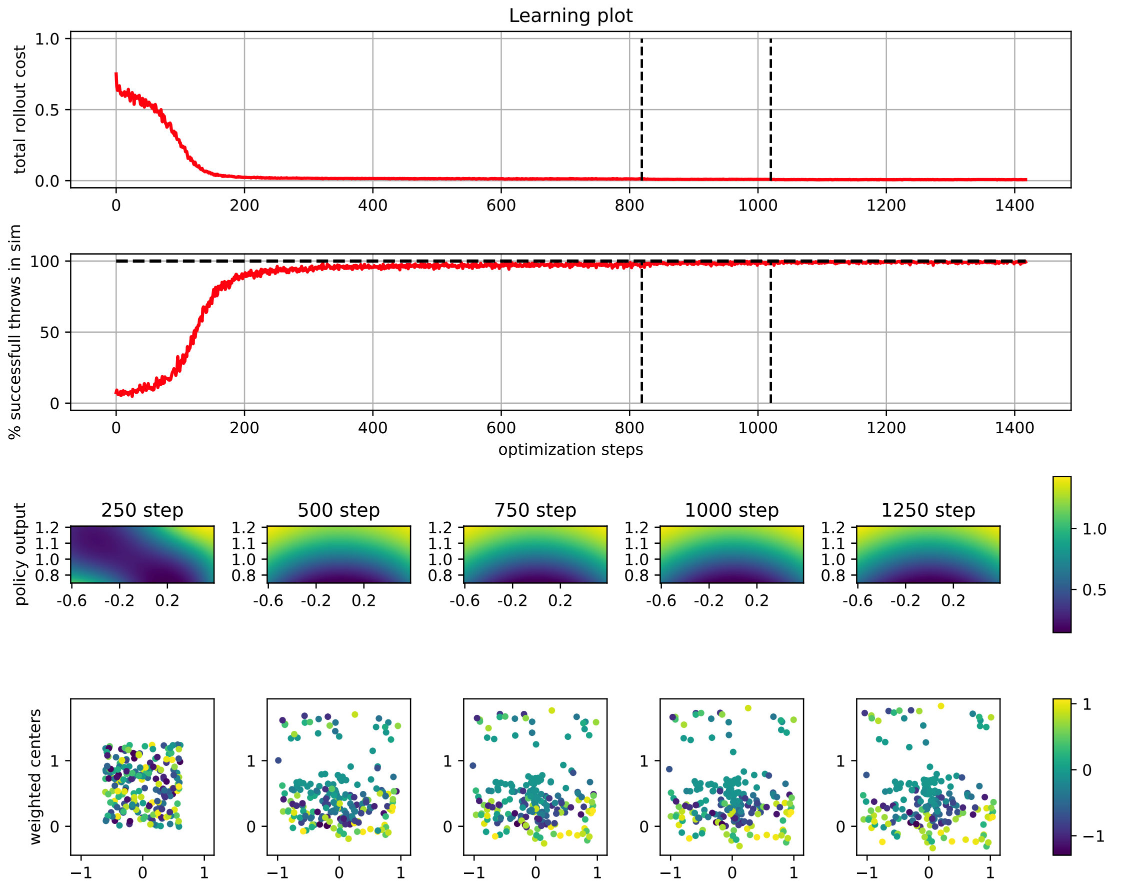

Figure 3.5: Setup 1: Sparse centers initialization, no modeled control noise, dropout applied

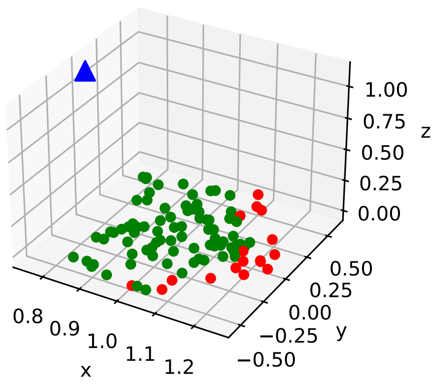

Figure 3.6: Setup 1: 83 % target reach performance.

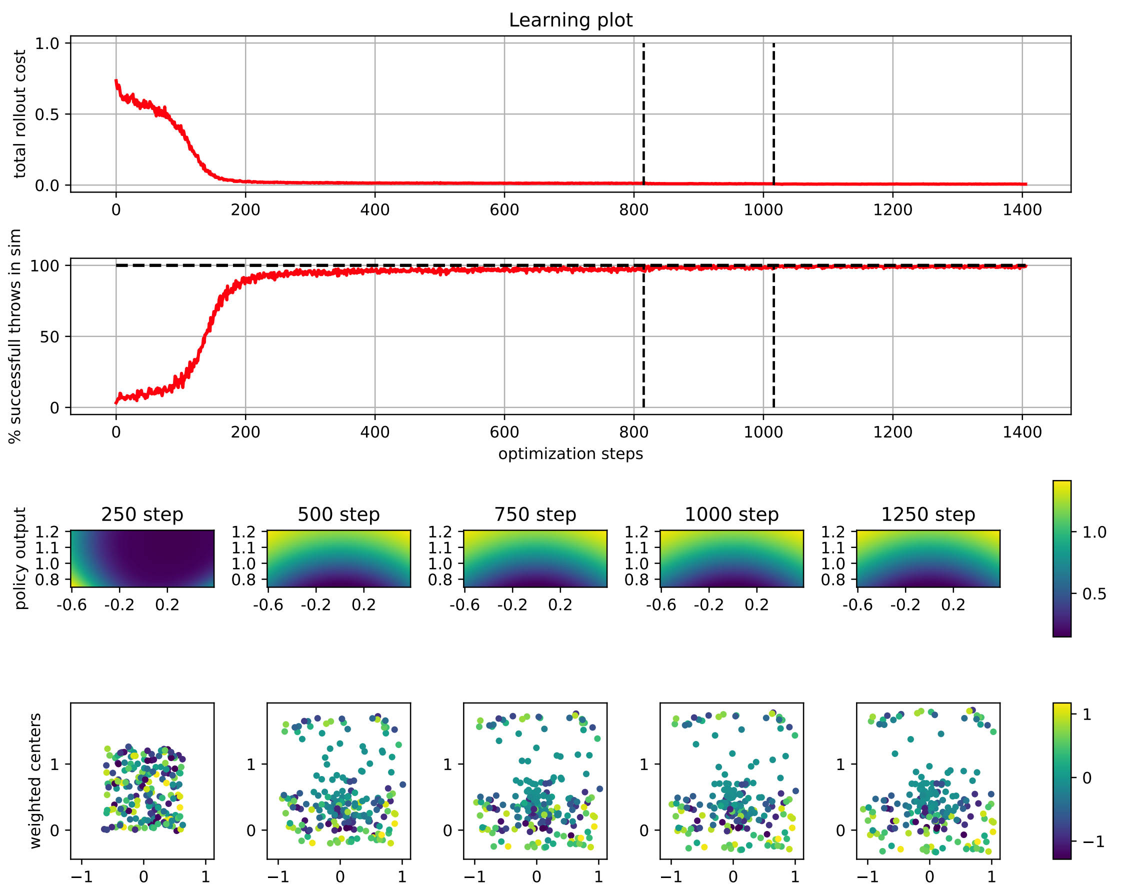

Figure 3.7: Setup 2: Sparse centers initialization, zero-mean control noise, dropout applied

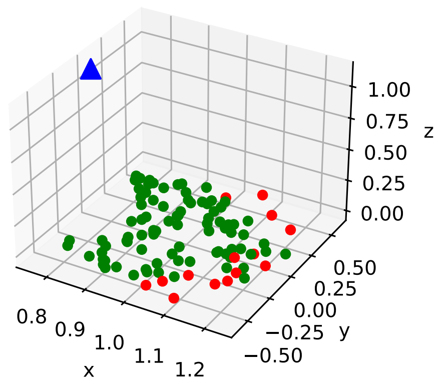

Figure 3.8: Setup 2: 86% target reach performance.

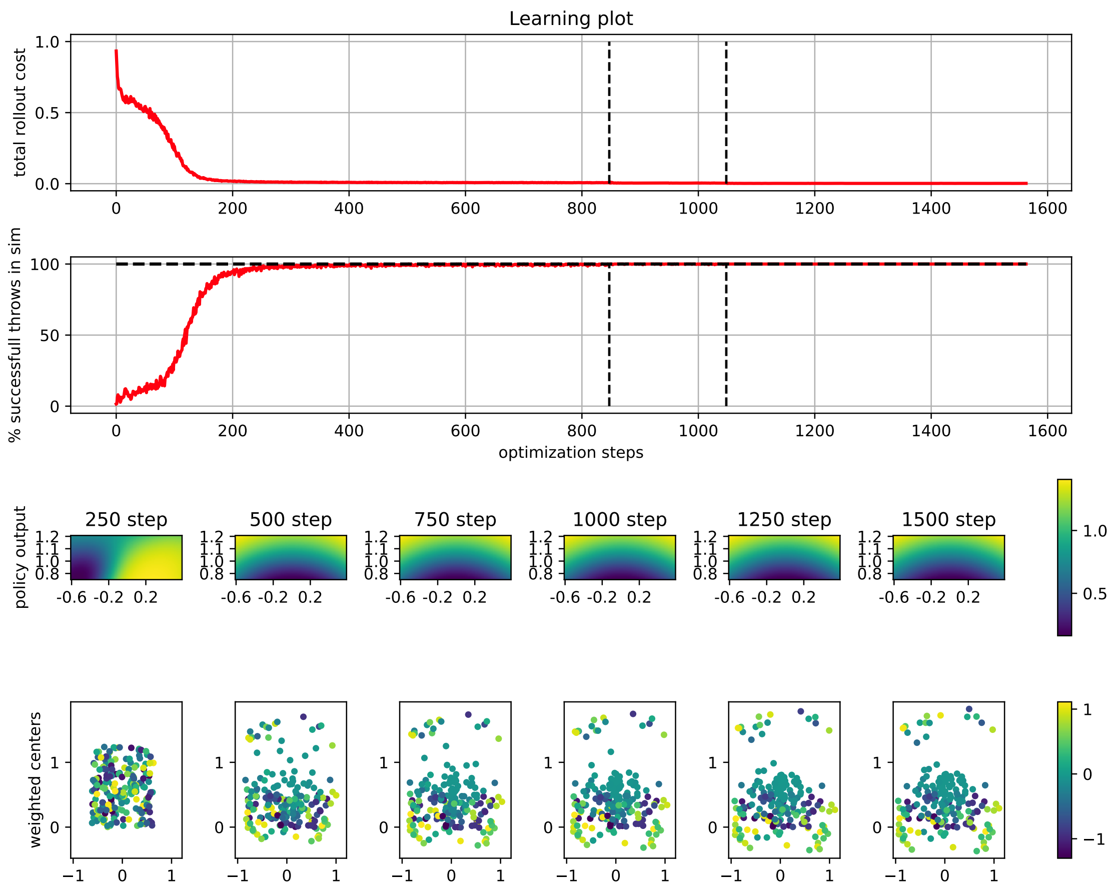

Figure 3.9: Setup 3: Sparse centers initialization, *overshooting* control noise, dropout applied

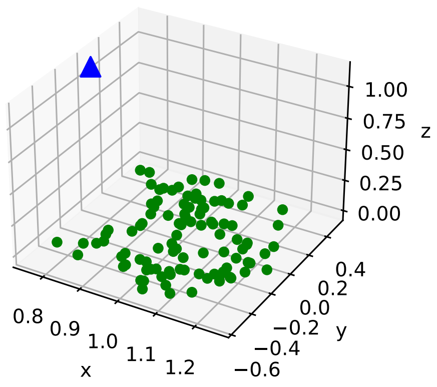

Figure 3.10: Setup 3: 100% target reach performance.

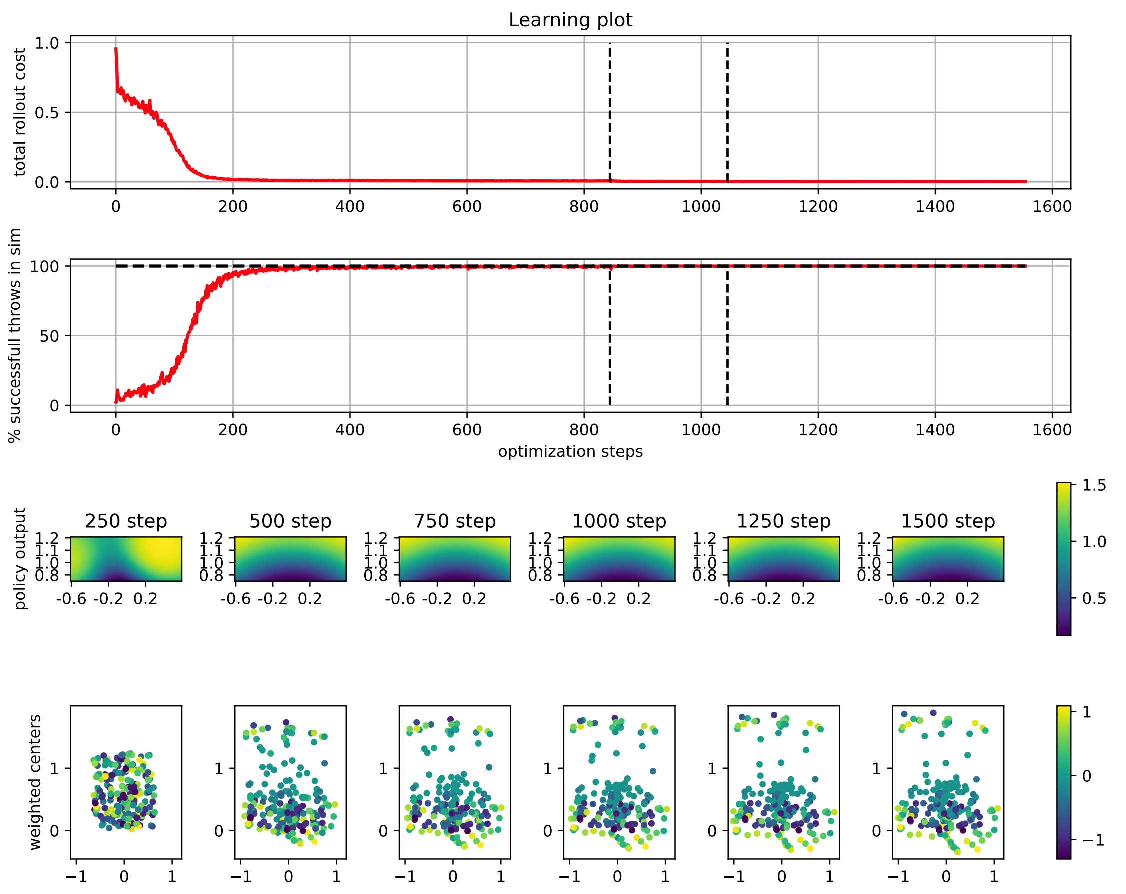

Figure 3.11: Setup 4: Sparse centers initialization, *undershooting* control noise, dropout applied

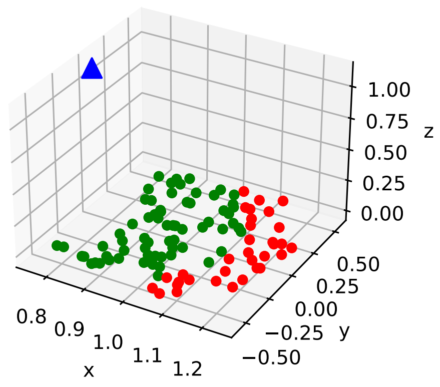

Figure 3.12: Setup 4: 67% target reach performance.

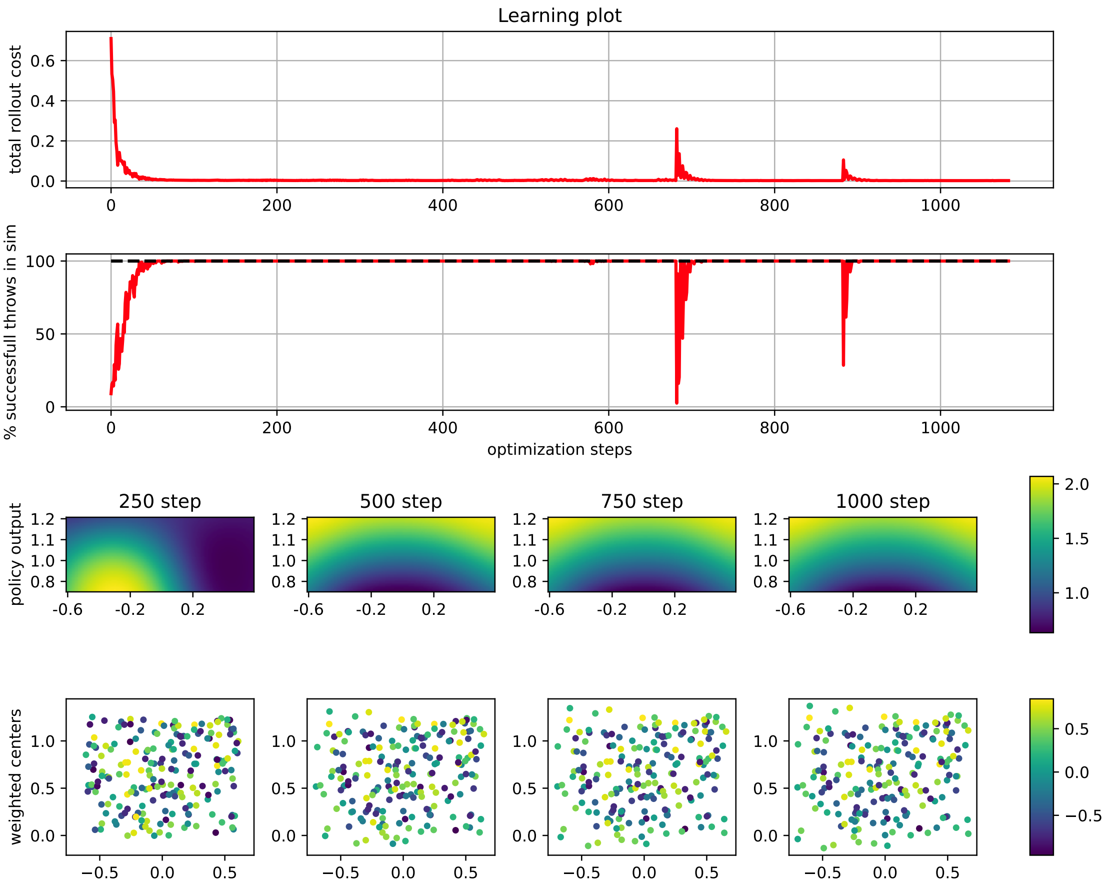

Figure 3.13: Setup 5: Sparse centers initialization, *overshooting* control noise, dropout not applied

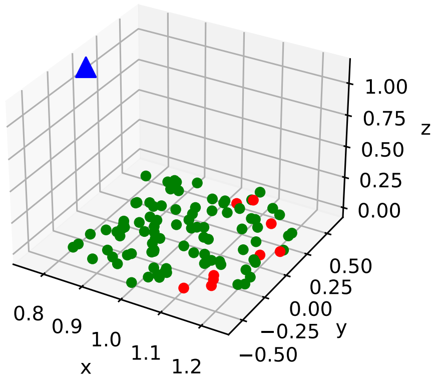

Figure 3.14: Setup 5: 91% target reach performance.

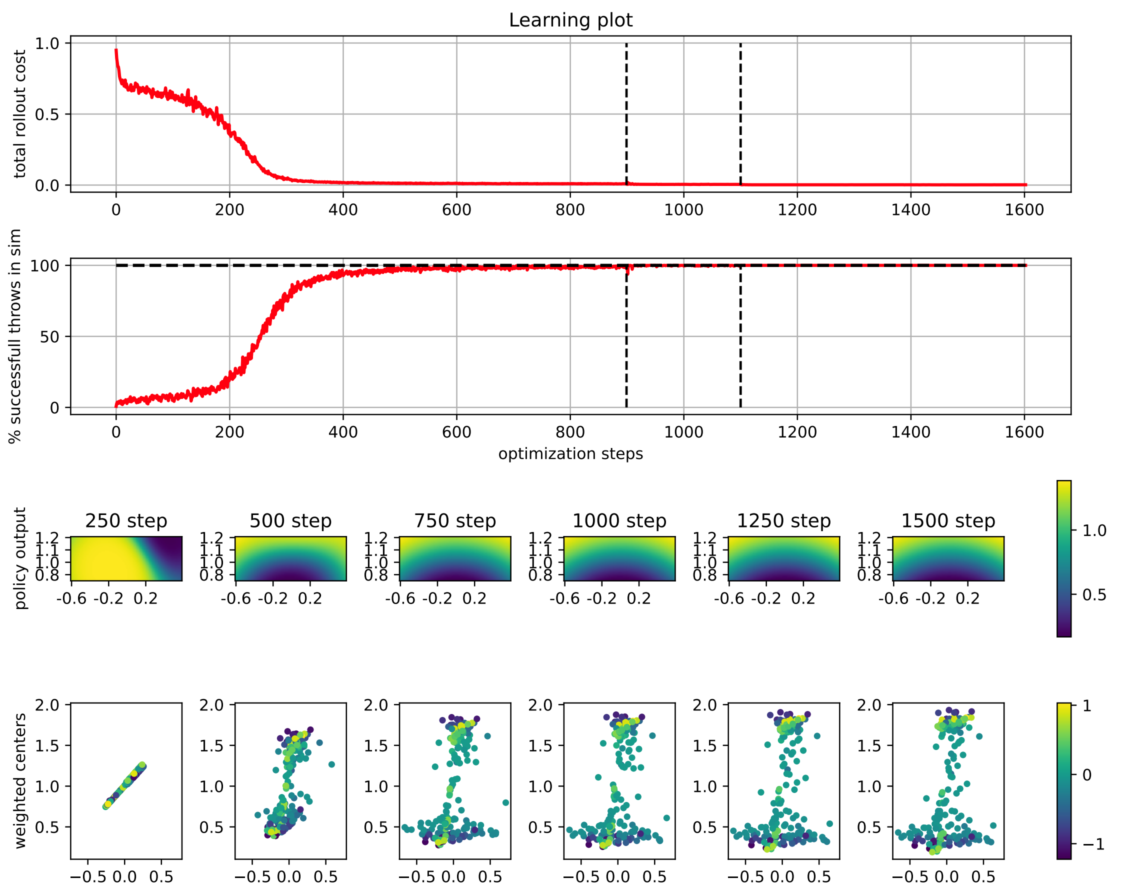

Figure 3.15: Setup 6: Line centers initialization, *overshooting* control noise, dropout applied

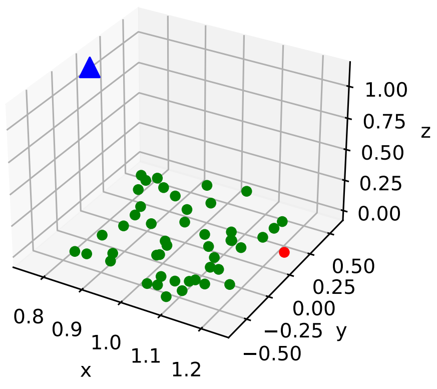

Figure 3.16: Setup 6: 98% target reach performance.

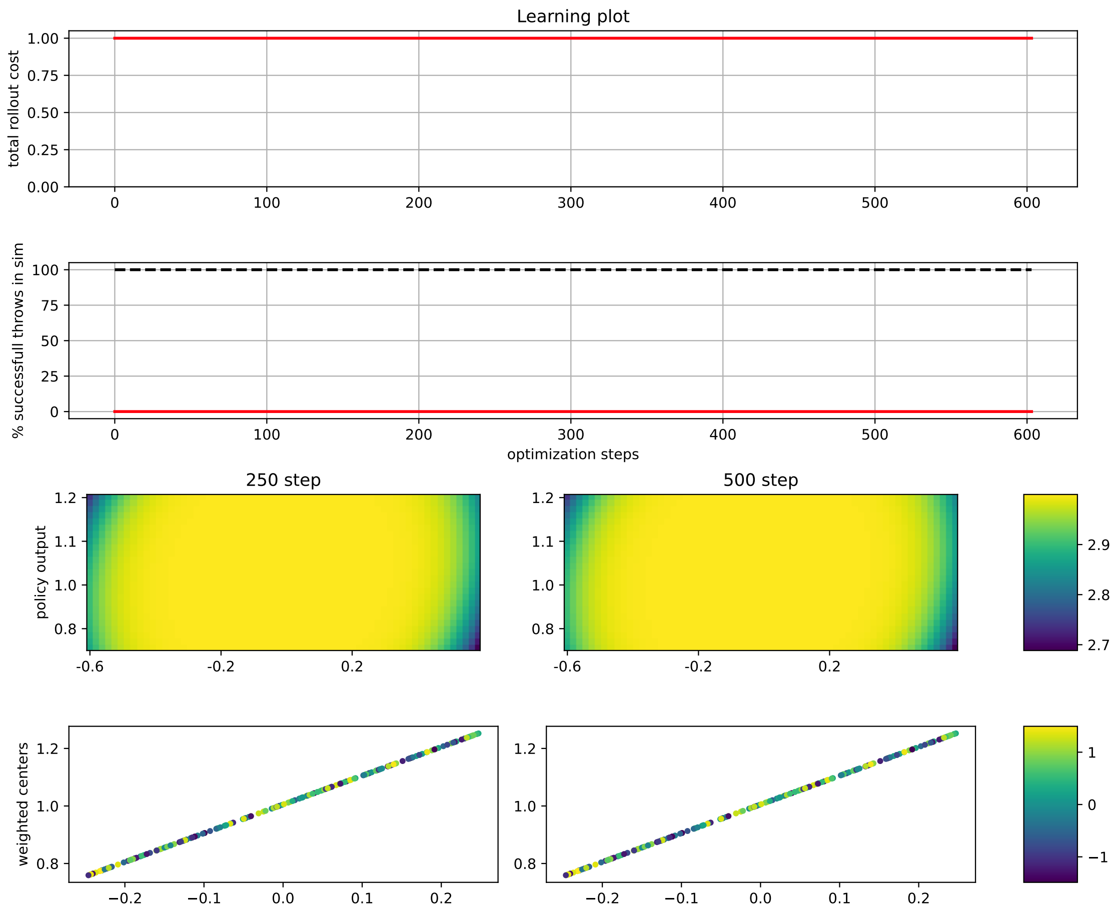

Figure 3.17: Setup 7: Line centers initialization, *overshooting* control noise, dropout not applied

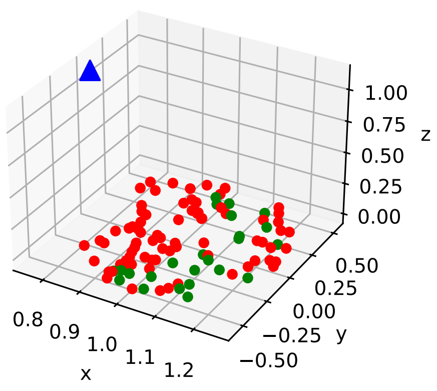

Figure 3.18: Setup 7: 24% target reach performance.

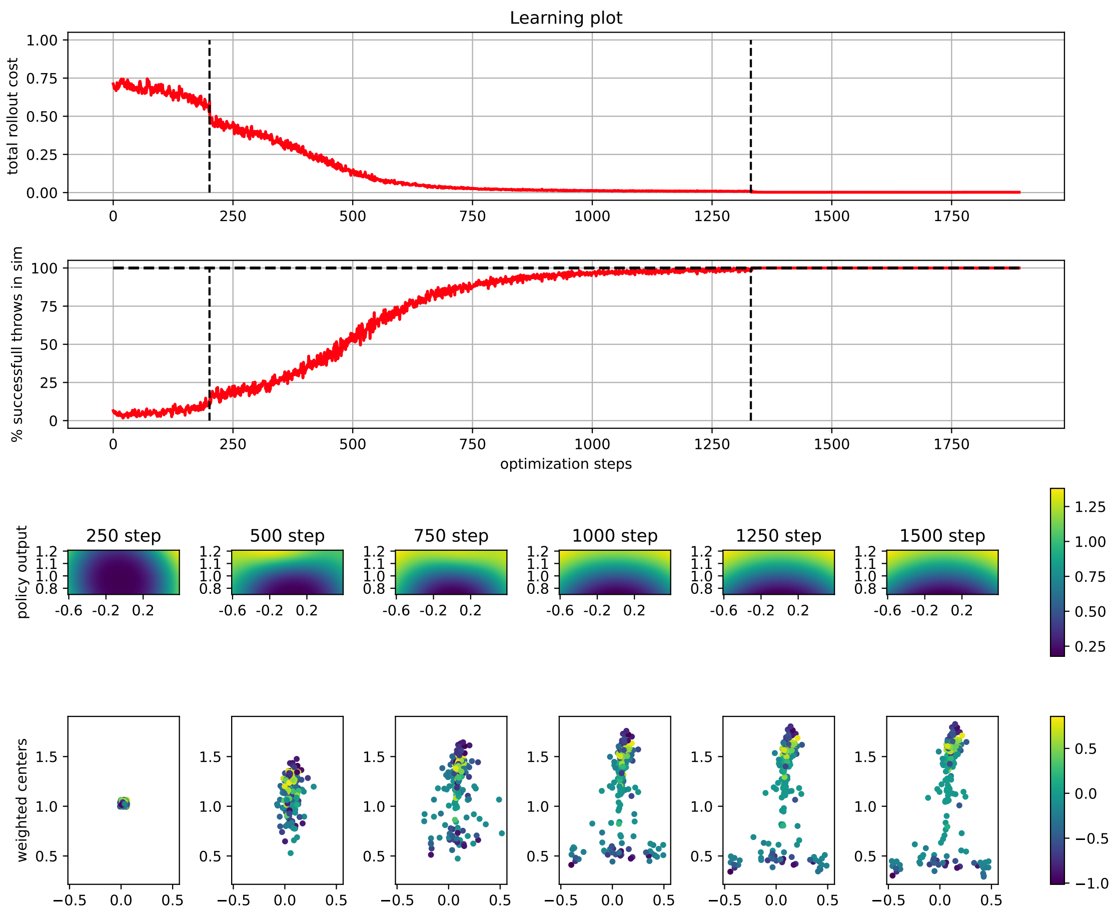

Figure 3.19: Setup 8: Focus centers initialization, *overshooting* control noise, dropout applied

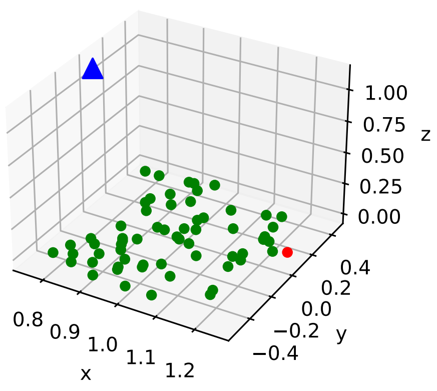

Figure 3.20: Setup 8: 98% target reach performance.

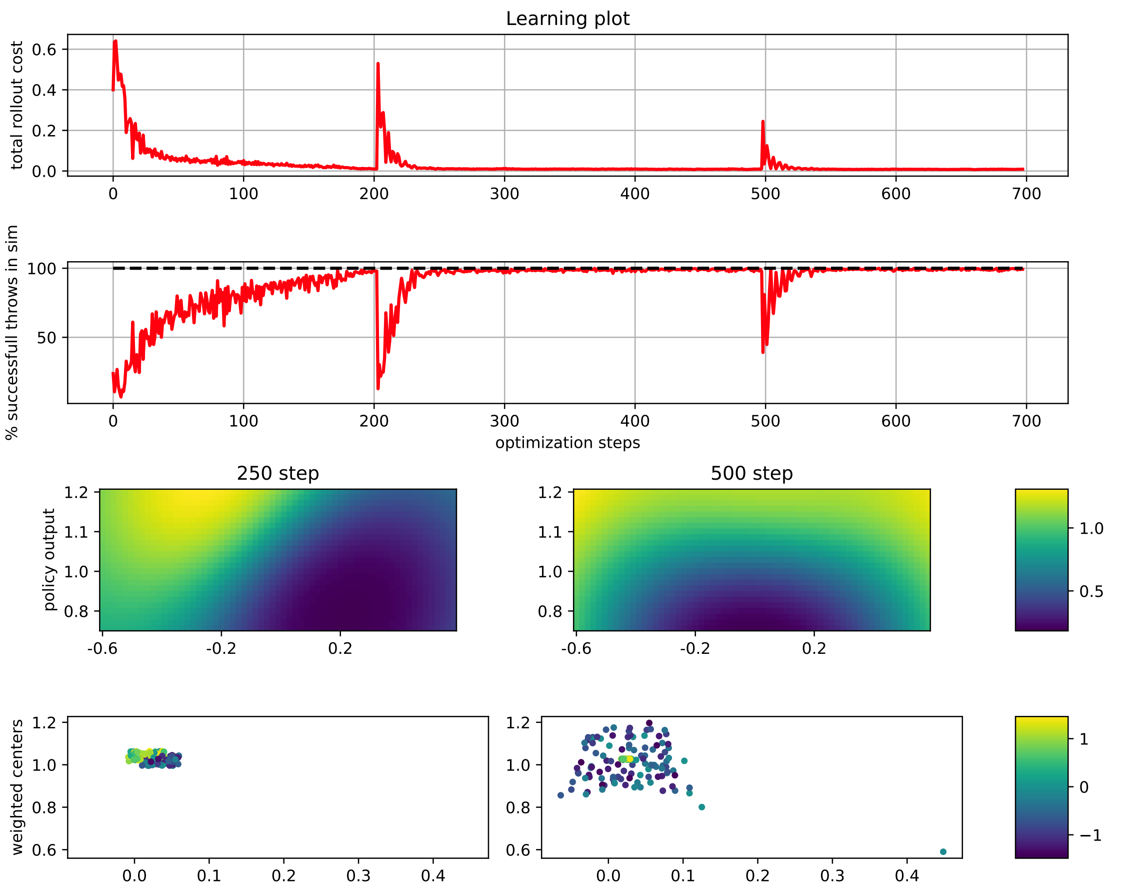

Figure 3.21: Setup 9: Focus centers initialization, *overshooting* control noise, dropout not applied

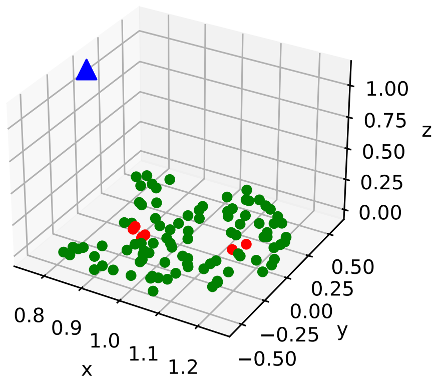

Figure 3.22: Setup 9: 93% target reach performance.

3.2 Laboratory

As stated in section 2.3, it was not feasible to replicate the whole task in laboratory yet, but it was indeed possible to collect some bullet trajectories in order to test the Model of section 2.2.1 on real data, the process of data collection is described in section 2.3.1.

The results presented in this section were all collected with the same Intel 3D camera[2] at sampling frequency of $60Hz$.

3.2.1 Model learning

The few example trajectories presented in section 2.3.1 are quite noisy, even if the worst outliers are removed, the same is true for all collected trajectories, therefore said data should not be used in such state. To reduce the white noise component a *low-pass* filter, with cutoff frequency $F_{lp} = 0.1Hz$, was applied to the trajectories before computing velocities and extracting the targets for Model Learning.

An example of such pre-processing is offered in figs. 3.23 and 3.24, in particular fig. 3.23 shows one of the trajectory collected during a laboratory session, the figure contains on the left the 3D plot of the trajectory, comparing the noisy and the filtered version, while on the right the velocities norms of the noisy and filtered trajectory are shown against time. The filtered plots are compatible with the equivalent plots obtained from the simulative data presented in fig. 2.9.

Instead, fig. 3.24 (left) shows both the noisy and filtered versions of the velocity components of the same trajectory in fig. 3.23, as well as the velocity deltas (right plot) from the original noisy trajectory in blue, and from the filtered trajectory in green, overlaid to the first. These diagrams in particular show that the filtering is able to mitigate most of the white noise effects, since the blue boxplots have all higher interquartile range. In particular the filtered plots of the deltas of x and y velocity components have null interquartile range and are just constant zero, which is expected, instead the box of the z component has still some oscillation around its average, which again, is expected.

Finally, fig. 3.25 presents the results of the same test performed in section 3.1.1 for 22 trajectory trials collected in laboratory. In this case the patterns are slightly different from fig. 3.2, but all 3 smoothed plots show clearly decreasing patters, the Rollout MSE is especially improving with the increase of training data. Most importantly, the performance of target prediction are compatible with those achieved by the model in simulation, therefore encouraging to test the policy on the real system once fixed the know issues.

Nonetheless, it was possible to run the MC-PILCO algorithm on the collected data (fig. 3.25) to verify if it is possible to train an optimal policy [3] for the considered task, the policy learning plot is presented

[2]model

[3]optimal w.r.t. the learned model

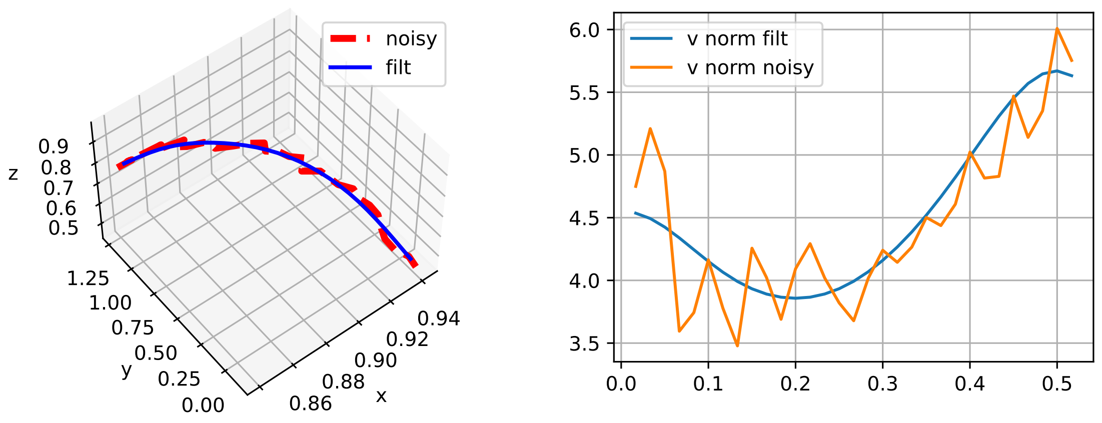

Figure 3.23: [a,b], (a): Example of a trajectory collected in laboratory (dashed red) and its filtered version (continuous blue). (b): the velocity norm along the trajectory of the original trajecotry (orange) and the velocity norm of the filtered trajectory (blue).

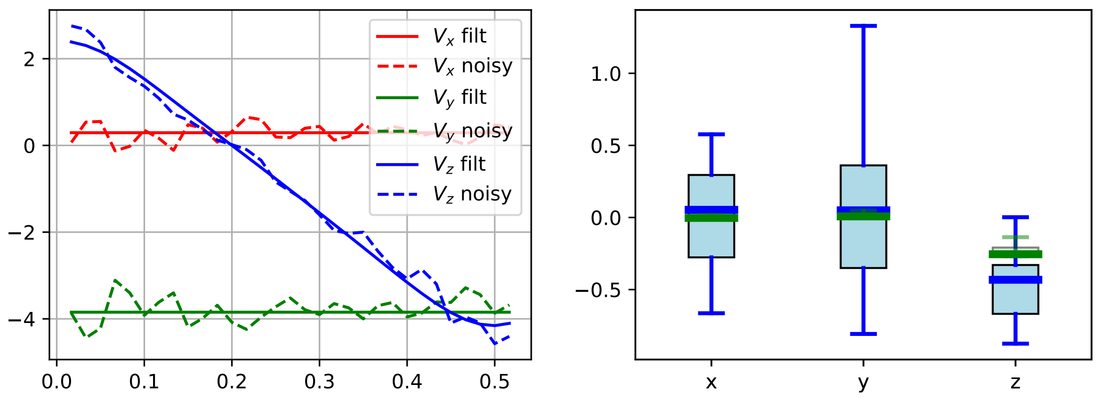

Figure 3.24: [a,b], (a): The velocity components from the example trajectory of fig. 3.23, comparing the original velocities (dashed) with the velocities from the filtered trajectory (continuous) (b): the velocity deltas from the same trajectory in a boxplot, comparing the deltas from the noisy trajectory (blue) and the deltas computed from the filtered trajectory (green).

and discussed in section 3.2.2.

In addition, we performed the same test of fig. 3.25 applying the mean function of eq. (2.16) to the GP models, results are shown in fig. 3.26, it is possible to notice that the trend of all curves is asymptotically the same of fig. 3.25, what changes is the initial part of the graphs, where performances are lower for the second test with mean function, the mean function has the effect of bootstrapping performances.

3.2.2 Policy learning

The policy, presented in section 2.2.3, is very constrained on the shooting direction, if the model predicts unnatural evolution of the

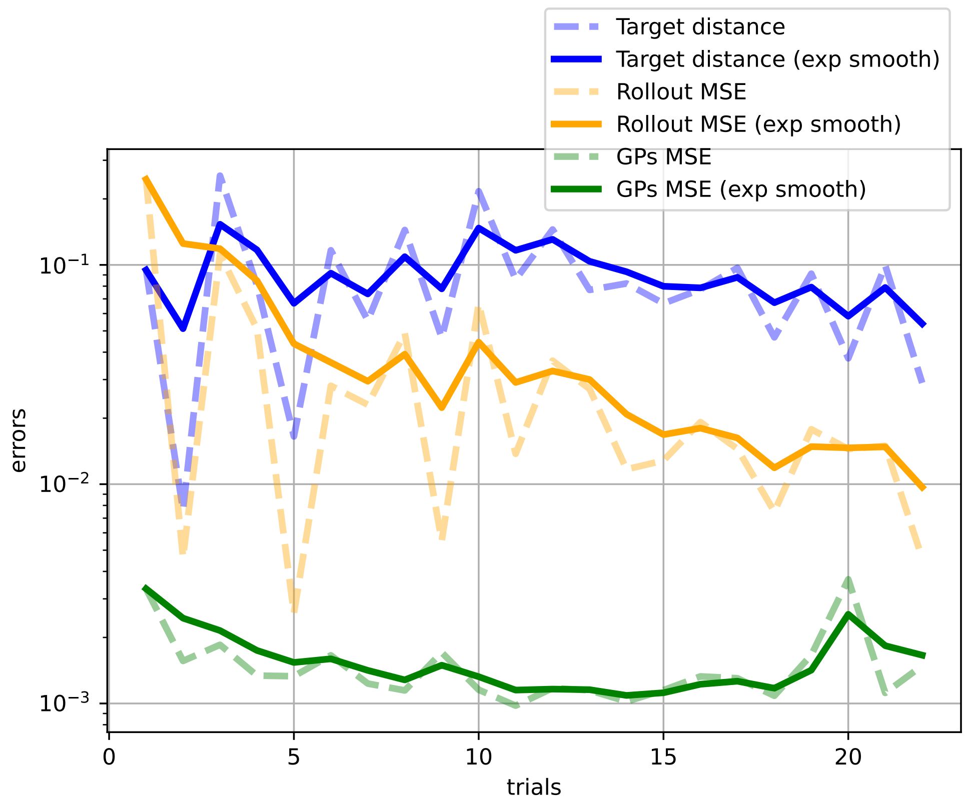

Figure 3.25: Model learning test on 22 throwing trials collected in laboratory (section 2.3.1), starting with the model trained on the data of one single exploration trial, for each test trial of the plot, the model is tested by means of: (i) distance over the $x - y$ plane of the predicted target from the actual landing; (ii) MSE of the Rollout prediction over the trajectory; (iii) MSE of the GPs on the targets. For each error computed in this test its curve is plotted (dashed) in comparison with its exponential smoothing (continuous) computed with $\alpha = 0.5$.
At each trial, after testing the Model, the GPs are retrained with the new data.

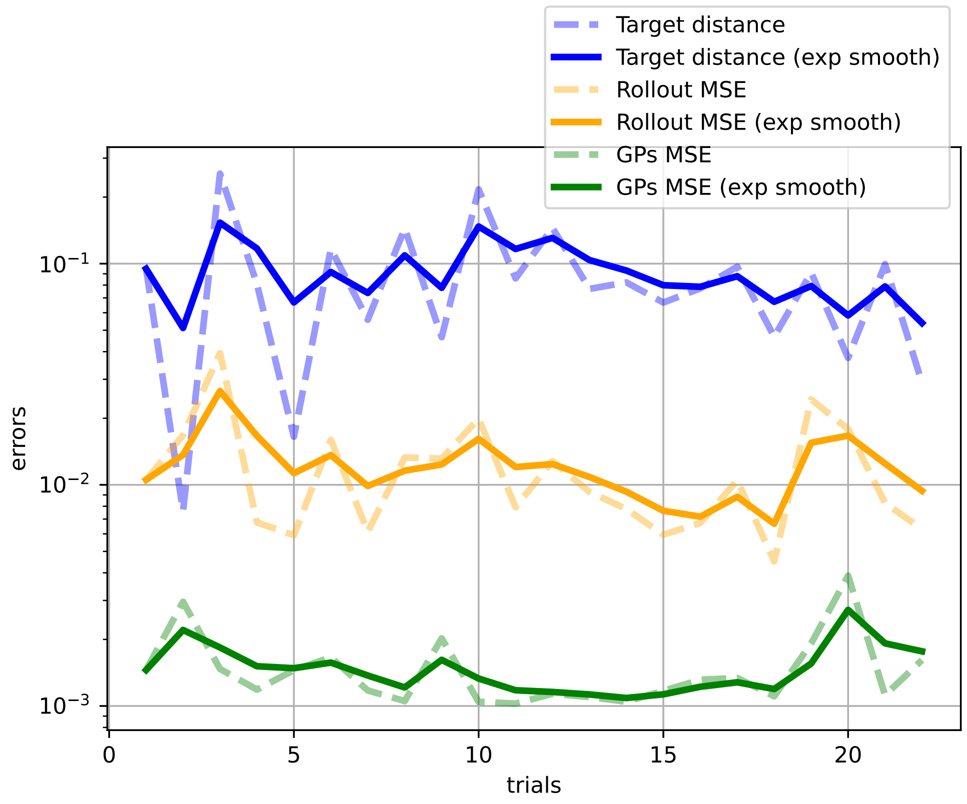

Figure 3.26: Same experiment of fig. 3.25, GPs equipped with mean function.

particles along the x, y directions, then the training process will never converge to optimal, since eq. (2.26) cannot handle such motions by definition. In any case, such effects are not present in the data presented to the model (figs. 3.23 and 3.24) mostly thanks to the estimation of the camera pose (described in section 2.3.1), and it is reasonable to expect for the policy update step to converge to a certain level of performance. Of course, expectations on the final performance should be lowered w.r.t. the simulative trials, indeed we cannot expect to have such precision to reach a target bin of the same dimensions as the one used in the simulated trials. We ran MC-PILCO with the data collected from 22 tossing trials collected in laboratory, with sparse centers initialization, no modeled tossing noise and with dropout. The reason we didn't perform the test with some modeled noise, is that the tossing system is not yet implemented in laboratory, therefore it is unknown. In fig. 3.27 it is possible to observe that the policy training yields a policy that is optimal w.r.t. the trained model, if we consider a target bin of radius $0.13m$, which is reasonable. For reference, consider that an office trash can has radius of about $0.15m$.

In fig. 3.27 the learning plot of the execution of MC-PILCO's policy update with the Model trained on real data can be observed, the plots are very simular to those of figs. 3.5, 3.7, 3.9, 3.11 and 3.13.

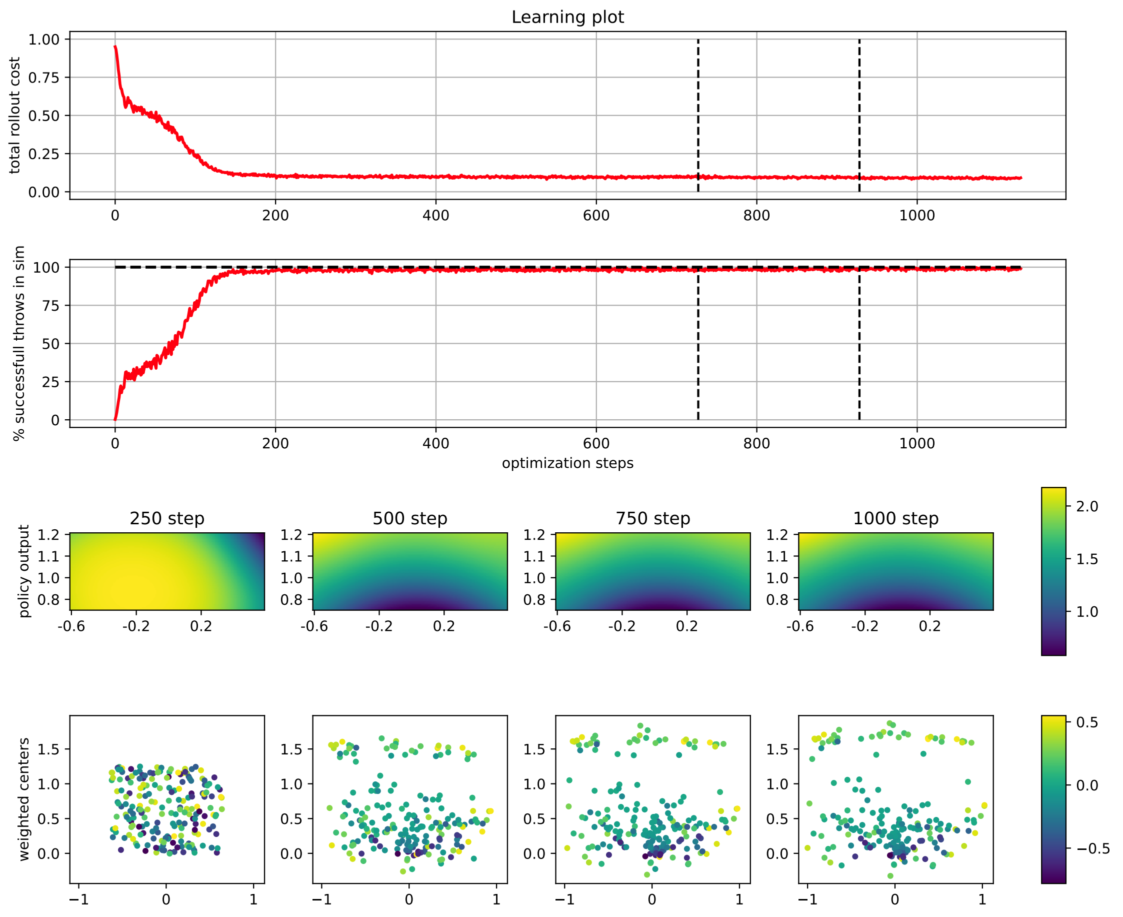

Figure 3.27: Execution of MC-PILCO with the model from 22 tossing trials collected in laboratory. Performances of target reach is simulations are computed with target radius=0.13m.

Chapter 4

Comments

It is clear that the approach presented in this thesis is working quite well in the simulated environment, which proves that given a sufficiently precise dynamics model, the policy learning of MC-PILCO is able to learn tasks whose outcome have long term dependencies to the control actions. The performance of the policy yieled with MC-PILCO is about the same of the result shown in [1] for the category ball in simulations, but it is evident that the two approaches take two different routes.

Without considering the difficulties of setting up the tossing system with the robotic manipulator, while this approach requires much more preliminar work for a physical implementation (3D tracking system) the actual training time required to the manipulator, in terms of exploration trials, is very limited.

Even in simulations, the approach of [1] requires hundreds of tossing trials in order to teach the robot to toss the ball, while this Model-Based approach only required a single exploration throw, in addition to the few trials needed to estimate the noise.

For example, if the tossing system tested in laboratory (see section 2.3.2) is precise enough in applying the desired initial speed at the nominal release point, the policy trained in fig. 3.27 should already be adequate to control the tossing. If that were true, we would have successfully trained the robot, without it actually doing any exploratory trials, if not, then we shoud estimate the system noise like it was done in section 2.2.1.1 and apply an adequate tossing noise in the policy update step.

The results presented in this chapter are quite promising, therefore we can define a short-term objective, that is to finalize the experiments with the ball tossing in laboratory, in order to validate all experiments performed in this thesis, this will require most probably to build a custom gripper.

Then as long term objective, once reached satisfactory performance with the ball, it would be interesting to perform the same task with different objects, as it was done in [1]. This work will likely reveal the differences between the two approaches in terms of implementation difficulties, since to track the evolution of complex objects like hammers, rods, pens it is also necessary to track their orientation in addition to the position along the trajectory. Therefore for each class of objects, we will need to perform several tossing trials, like in section 2.3.1, but it will be necessary to perform a complete pose estimation for each sample image.

Finally if we can estimate (or define) the complete geometry of the objects, then using standard physics principles we can bootstrap the Model learning exploiting the inertial properties of rigid objects, defining for example a mean function like in section 2.2.1 or defining a custom kernel. Hopefully, this could allow to appreciate the differences in terms of efficiency of MC-PILCO's Model-Based approach w.r.t. unsupervised learning techniques like the one that [1] relied on.

Furthermore, a better noise reduction for the data collection will likely improve the Model's performance, this should be done both at the source with a more robust acquisition and also with the trajectories data with some filtering more adequate that a simple low-pass.

Appendix A

Gaussian Processes

Gaussian Processes (GPs) are Supervised Learning tools for both regression and classification problems. Whereas a classification task aims at predicting discrete class labels, a regression task is concerned with the prediction of continuous quantities. For example, in a business application, one may attempt to predict the price of a house as a function of some features of the building. This appendix will introduce all the notions and concepts regarding GP regression (GPR) needed to understand the work developed for this thesis, many insights and techincal details will be omitted, refer to [35] for deeper explainations.

Let define:

$$\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) \mid i = 1, \ldots, n\} \tag{A.1}$$

as a training set with n observations, where $\boldsymbol{x}_i$ denotes an input vector (covariates) of dimension D and $\boldsymbol{y}_i$ the respective scalar output or target (dependent variable); the input vectors can be aggregated as columns in a $D \times n$ matrix, called design matrix $\boldsymbol{X}$, while the targets can be collected in a vector $\boldsymbol{Y}$, so it is possible to define the training set as: $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{Y})$.

The main interest is to infer the relationship between inputs and targets, i.e. the conditional probability distribution of the targets given the inputs.

There is not a single way to view and interpret GPs, in the following we will present two equivalent interpretations: one looking at the space of parameters of the GP, one interpreting the GP as defining a distribution over functions, and inference taking place directly in the space of functions.

A.1 Parameters-space View

GP regresssion assumes the following Bayesian probabilistic model:

$$\begin{aligned} \boldsymbol{y} &= f(\boldsymbol{x},\boldsymbol{w}) + \epsilon \\ \epsilon &\sim \mathcal{N}(0,\sigma_n^2) \end{aligned} \tag{A.2}$$

This model assumes that the input vectors $\boldsymbol{x}$ are mapped to the respective observation $\boldsymbol{y}$ by means of a parametrized function $f(\cdot,\boldsymbol{w})$ plus additive noise ϵ. The noise is assumed to follow an independent, identically distributed zero-mean Gaussian distribution with variance σ_n^2. Moreover, this model assumes that an observation $\boldsymbol{y}$ is fully determined by its input $\boldsymbol{x}$ and the parameters $\boldsymbol{w}$ of the function, with uncertainty defined by ϵ, resulting in the independence of each observations $\boldsymbol{y}_i$ from other observations $\boldsymbol{y}_j$ and inputs $\boldsymbol{x}_j$ $(i \neq j)$.

This assumed model directly originates the *likelihood*, the probability density of the observations given the parameters:

$$p(\boldsymbol{Y}|\boldsymbol{X},\boldsymbol{w}) \tag{A.3}$$

which can be factored over cases in the training set (because of the independence assumption) to give:

$$\begin{aligned} p(\boldsymbol{Y}|\boldsymbol{X},\boldsymbol{w}) &= \prod_{i=1}^{n} p(\boldsymbol{y}_i|\boldsymbol{x}_i,\boldsymbol{w}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp(-\frac{(\boldsymbol{y}_i - f(\boldsymbol{x}_i,\boldsymbol{w}))^2)}{2\sigma_n^2}) \\ &= \frac{1}{(2\pi\sigma_n^2)^{n/2}} \exp(-\frac{1}{2\sigma_n^2}|\boldsymbol{Y} - f(\boldsymbol{X},\boldsymbol{w})|^2) = \mathcal{N}(f(\boldsymbol{X},\boldsymbol{w}),\sigma_n^2 I) \end{aligned} \tag{A.4}$$

because it is assumed: $p(\boldsymbol{y}_i|\boldsymbol{x}_i,\boldsymbol{w}) = \mathcal{N}(f(\boldsymbol{x}_i,\boldsymbol{w}),\sigma_n^2)$.

Where $|z|$ is the Euclidean norm of a vector z. In the Bayesian formalism it is necessary to define a *pior* probability distribution over the parameters $\boldsymbol{w}$, expressing the belief over $\boldsymbol{w}$ before looking at $\boldsymbol{Y}$. This can be done for example modeling the prior as:

$$\boldsymbol{w} \sim \mathcal{N}(0,\Sigma_{\boldsymbol{w}}) \tag{A.5}$$

In the simplest case, the function $f(\cdot,\boldsymbol{w})$ is a linear function:

$$f(\boldsymbol{x},\boldsymbol{w}) = \boldsymbol{x}^T\boldsymbol{w} \tag{A.6}$$

and the *likelihood* becomes:

$$p(\boldsymbol{Y}|\boldsymbol{X},\boldsymbol{w}) = \mathcal{N}(\boldsymbol{X}^T\boldsymbol{w},\sigma_n^2 I) \tag{A.7}$$

Given a previously unobserved input vector x_*, also called test sample, the prediction of the respective target value f_* is computed averaging over all possible parameter values, weighted by their posterior probability.

Thus, the predictive distribution for $f_* \triangleq f(x_*,\boldsymbol{w})$ at x_*:

$$p(f_*|x_*,\boldsymbol{X},\boldsymbol{Y}) = \int p(f_*|x_*,\boldsymbol{w})p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{Y})d\boldsymbol{w} \tag{A.8}$$

The posterior distribution over the weights, is computed by Bayes' rule:

$$p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{Y}) = \frac{p(\boldsymbol{Y}|\boldsymbol{X},\boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{Y}|\boldsymbol{X})} \tag{A.9}$$

Where the term at denominator is a normalizing constant also called *marginal likelihood.* Moreover it is independent of the parameters $\boldsymbol{w}$ and is given by:

$$p(\boldsymbol{Y}|\boldsymbol{X}) = \int p(\boldsymbol{Y}|\boldsymbol{X},\boldsymbol{w})p(\boldsymbol{w})d\boldsymbol{w} \tag{A.10}$$

The posterior eq. (A.9) combines the likelihood and the prior, and captures everything known about the parameters, it can be computed analytically given the model. For example with the linear model in eq. (A.6):

$$\begin{aligned} p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{Y}) &\sim \mathcal{N}(\bar{\boldsymbol{w}},A^{-1}) \\ \bar{\boldsymbol{w}} &= \frac{1}{\sigma_n^2}A^{-1}\boldsymbol{X}\boldsymbol{Y} \\ A &= \sigma_n^{-2}\boldsymbol{X}\boldsymbol{X}^T + \Sigma_m^{-1} \end{aligned} \tag{A.11}$$

The predictive distribution of f_* in this case becomes:

$$p(f_*|x_*,\boldsymbol{X},\boldsymbol{Y}) = \mathcal{N}(x_*^T\bar{\boldsymbol{w}},x_*^T A^{-1}x_*) \tag{A.12}$$

A.2 Function-space View

This interpretation uses a Gaussian process (GP) to describe a distribution over functions. Formally, a GP is defined as *a collection of random variables, any finite number of which have a joint Gaussian* *distribution.*

Given a real function $f(\boldsymbol{x})$, a GP can be completely defined by its *mean* function $m(\boldsymbol{x})$ and *covariance* function or *kernel* $k(\boldsymbol{x},\boldsymbol{x}')$ as:

$$\begin{aligned} m(\boldsymbol{x}) &= \mathbb{E}[f(\boldsymbol{x})] \\ k(\boldsymbol{x},\boldsymbol{x}') &= \mathbb{E}[(f(\boldsymbol{x})-m(\boldsymbol{x}))(f(\boldsymbol{x}')-m(\boldsymbol{x}'))] \end{aligned} \tag{A.13}$$

The GP is then written as:

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}),k(\boldsymbol{x},\boldsymbol{x}')) \tag{A.14}$$

In the simplest case, the mean function can be zero, or can be defined a priori exploiting some knowledge of the problem.

A classical covariance function is the *squared exponential* (SE):

$$k(\boldsymbol{x},\boldsymbol{x}') = \lambda^2 \exp(-||\boldsymbol{x}-\boldsymbol{x}'||^2_\Sigma) \tag{A.15}$$

which has been exploited in this project. It is possible to see that the SE is almost unity if the input points are very close, and decreases as their distance increases. It can be shown that the SE covariance function corresponds to the linear model in appendix A.1 with an infinite number of basis functions. SE can also be obtained from the linear combination of an infinite number of Gaussian-shaped basis functions.

A Gaussian process is defined as a collection of random variables, representing the value of the function $f(\boldsymbol{x})$ at location $\boldsymbol{x}$. This implies a consistency property, also known as the marginalization property. This simply means that $(\boldsymbol{y}_1,\boldsymbol{y}_2) \sim \mathcal{N}(\mu,\Sigma) \rightarrow \boldsymbol{y}_1 \sim \mathcal{N}(\mu_1,\Sigma_{11})$.

Considering a zero-mean GP, a distribution over functions can be completely defined by means of a covariance function, for example one can draw from the distribution of functions evaluated at any number of points, by choosing the input points: $\boldsymbol{X}_*$ and defining the relative covariance matrix:

$$\begin{aligned} \mathbf{f}_* &\sim \mathcal{N}(0,K(\boldsymbol{X}_*,\boldsymbol{X}_*)) \\ K(\boldsymbol{X}_*,\boldsymbol{X}_*) &= \big(k(\boldsymbol{x}_{*i},\boldsymbol{x}_{*j})\big)_{ij} \\ \boldsymbol{X}_* &= \big[\boldsymbol{x}_{*1},\ldots,\boldsymbol{x}_{*n}\big] \end{aligned} \tag{A.16}$$

As in the appendix A.1 the main interest is to predict the target value $\mathbf{f}_*$ of a previously unobserved input point $\boldsymbol{x}_*$, in the following we will present the prediction both in absence and in presence of noise in the observations.

A.2.1 Prediction with Noiseless Observation data

In this case, the assumed model is:

$$\boldsymbol{y} = f(\boldsymbol{x}) \tag{A.17}$$

Which is conceptually equivalent to the model in eq. (A.2) with $\sigma_n = 0$.

We could rewrite the train set in matrix form as: $\mathcal{D} = (\boldsymbol{X}, \mathbf{f})$, where $\mathbf{f} = f(\boldsymbol{X})$. The joint distribution of $\mathbf{f}$ and $\mathbf{f}_* = f(\boldsymbol{X}_*)$ is then defined, according to the prior, as:

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(\boldsymbol{X},\boldsymbol{X}) & K(\boldsymbol{X},\boldsymbol{X}_*) \\ K(\boldsymbol{X}_*,\boldsymbol{X}) & K(\boldsymbol{X}_*,\boldsymbol{X}_*) \end{bmatrix}\right) \tag{A.18}$$

It is possible then, to define the posterior probability of $\mathbf{f}_*$ given $\mathcal{D}$ and $\boldsymbol{X}_*$, by *conditioning* the joint Gaussian prior distribution on the observations, which results in:

$$\begin{aligned} \mathbf{f}_* | \boldsymbol{X}_*, \boldsymbol{X}, \mathbf{f} &\sim \mathcal{N}(\hat{\mathbf{f}}_*, cov(\mathbf{f}_*)) \\ \hat{\mathbf{f}}_* &= K(\boldsymbol{X}_*,\boldsymbol{X}) K(\boldsymbol{X},\boldsymbol{X})^{-1} \mathbf{f}, \\ cov(\mathbf{f}_*)) &= K(\boldsymbol{X}_*,\boldsymbol{X}_*) - K(\boldsymbol{X}_*,\boldsymbol{X}) K(\boldsymbol{X},\boldsymbol{X})^{-1} K(\boldsymbol{X},\boldsymbol{X}_*)) \end{aligned} \tag{A.19}$$

A.2.2 Prediction with Noisy Observation data

A more realistic assumption is that the available observations are not the desired function values themselves, but noisy versions of them:

$$\begin{aligned} \mathcal{D} &= (\boldsymbol{X}, \boldsymbol{Y}) \\ \boldsymbol{y} &= f(\boldsymbol{x}) + \epsilon \end{aligned} \tag{A.20}$$

Like in eq. (A.2), it is assumed additive independent identically distributed Gaussian noise ϵ with variance σ_n^2. Therefore, the prior on the noisy observations becomes:

$$cov(\boldsymbol{Y}) = K(\boldsymbol{X},\boldsymbol{X}) + \sigma_n^2 I \tag{A.21}$$

Therefore, the introduction of noise in eq. (A.18) results in:

$$\begin{bmatrix} \boldsymbol{Y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(\boldsymbol{X},\boldsymbol{X}) + \sigma_n^2 I & K(\boldsymbol{X},\boldsymbol{X}_*) \\ K(\boldsymbol{X}_*,\boldsymbol{X}) & K(\boldsymbol{X}_*,\boldsymbol{X}_*) \end{bmatrix}\right) \tag{A.22}$$

Finally, the predictive distribution for $\mathbf{f}_*$ can be obtained exactly like in eq. (A.19):

$$\begin{aligned}
\mathbf{f}_*|\boldsymbol{X}_*,\boldsymbol{X},\boldsymbol{Y} &\sim \mathcal{N}(\hat{\mathbf{f}}_*, cov(\mathbf{f}_*)) \\
\hat{\mathbf{f}}_* &= K(\boldsymbol{X}_*,\boldsymbol{X})\boldsymbol{\alpha} \\
cov(\mathbf{f}_*)) &= K(\boldsymbol{X}_*,\boldsymbol{X}_*) - K(\boldsymbol{X}_*,\boldsymbol{X})[K(\boldsymbol{X},\boldsymbol{X}) + \sigma_n^2 I]^{-1} K(\boldsymbol{X},\boldsymbol{X}_*))
\end{aligned} \tag{A.23}$$

where

$$\boldsymbol{\alpha} = [K(\boldsymbol{X},\boldsymbol{X}) + \sigma_n^2 I]^{-1}\boldsymbol{Y} \tag{A.24}$$

Notice that the vector $\boldsymbol{\alpha}$ can be computed once and be used for all successive predictions.

In the case that there is only one test point $\boldsymbol{x}_*$, the notation can be compacted, eq. (A.23) and eq. (A.24) are reduced to:

$$\begin{aligned}
\hat{\mathbf{f}}_* &= K_*^T \boldsymbol{\alpha} \\
\mathbb{V}[\mathbf{f}_*] &= k(\boldsymbol{x}_*,\boldsymbol{x}_*) - K_*^T (K + \sigma_n^2 I)^{-1} K_*
\end{aligned} \tag{A.25}$$

with $K_* = K(\boldsymbol{X},\boldsymbol{x}_*)$ and $K = K(\boldsymbol{X},\boldsymbol{X})$.

In this case $\hat{\mathbf{f}}_*$ is used as predicted value of $\mathbf{f}_*$, while $\mathbb{V}[\mathbf{f}_*]$ is its respective variance, and it is interpreted as value of uncertainty of the prediction.

A.2.3 Selection of Hyperparameters - Hints

Following the approach presented in appendix A.2, a GP is fully determined once a Dataset $\mathcal{D}$ and a covariance function k are defined.

Covariance functions, like eq. (A.15) are defined by some *hyper-parameters*, therefore it is required to select for each hyparameter a proper value in a way that the regression performance of the final model is maximized.

The process of defining a covariance function and its hyperparameters is typically referred as *training* of a GP. A common training approach consists in selecting the hyperparameters by maximization of the so called *marginal likelihood* of the training samples:

$$p(\boldsymbol{y}|\boldsymbol{X}) = \int p(\boldsymbol{y}|\mathbf{f},\boldsymbol{X}) p(\mathbf{f}|\boldsymbol{X}) d\mathbf{f} \tag{A.26}$$

The term marginal likelihood refers to the marginalization over the function values f, following the Gaussian process model, both the prior and the likelihood are Gaussian: $\mathbf{f}|\boldsymbol{X} \sim \mathcal{N}(0, K)$, $\boldsymbol{y}|\mathbf{f} \sim \mathcal{N}(\mathbf{f}, \sigma_n^2 I)$, resulting in:

$$\boldsymbol{y} \sim \mathcal{N}(0, K + \sigma_n^2 I) \tag{A.27}$$

With these considerations in mind, it is possible to yield a convenient expression of the *log* marginal likelihood:

$$\log p(\boldsymbol{y}|\boldsymbol{X}) = -\frac{1}{2}\boldsymbol{y}^T (K + \sigma_n^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}\log|K + \sigma_n^2 I| - \frac{n}{2}\log 2\pi \tag{A.28}$$

Note that K depends on the covariance function and its hyperparameters θ, therefore it is possible, for example, to perform gradient ascent by computing $\delta\theta = \nabla_\theta \log p(\boldsymbol{y}|\boldsymbol{X})$.

Bibliography

[1] A. Zeng, S. Song, J. Lee, A. Rodriguez, and T. Funkhouser, "Tossingbot: Learning to throw arbitrary objects with residual physics," 2019.

[2] A. Elatta, L. P. Gen, F. L. Zhi, Y. Daoyuan, and L. Fei, "An overview of robot calibration," *Information Technology Journal*, vol. 3, no. 1, pp. 74–78, 2004.

[3] R. Y. Tsai, R. K. Lenz *et al.*, "A new technique for fully autonomous and efficient 3 d robotics hand/eye calibration," *IEEE Transactions on robotics and automation*, vol. 5, no. 3, pp. 345–358, 1989.

[4] A. A. Ata, "Optimal trajectory planning of manipulators: a review," *Journal of Engineering Science and technology*, vol. 2, no. 1, pp. 32–54, 2007.

[5] T. Chettibi, H. Lehtihet, M. Haddad, and S. Hanchi, "Minimum cost trajectory planning for industrial robots," *European Journal of Mechanics-A/Solids*, vol. 23, no. 4, pp. 703–715, 2004.

[6] S. S. Perumaal and N. Jawahar, "Automated trajectory planner of industrial robot for pick-and-place task," *International Journal of Advanced Robotic Systems*, vol. 10, no. 2, p. 100, 2013. [Online]. Available: https://doi.org/10.5772/53940

[7] S. D. Han, S. W. Feng, and J. Yu, "Toward fast and optimal robotic pick-and-place on a moving conveyor," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 446–453, 2019.

[8] H. Liu, X. Lai, and W. Wu, "Time-optimal and jerk-continuous trajectory planning for robot manipulators with kinematic constraints," *Robotics and Computer-Integrated Manufacturing*, vol. 29, no. 2, pp. 309–317, 2013.

[9] P.-C. Huang and A. K. Mok, "A case study of cyber-physical system design: Autonomous pick-and-place robot," in *2018 IEEE*

24th international conference on embedded and real-time computing systems and applications (RTCSA). IEEE, 2018, pp. 22–31.

[10] N. C. Dafle, A. Rodriguez, R. Paolini, B. Tang, S. S. Srinivasa, M. Erdmann, M. T. Mason, I. Lundberg, H. Staab, and T. Fuhlbrigge, "Extrinsic dexterity: In-hand manipulation with external forces," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 1578–1585.

[11] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[12] F. Amadio, A. D. Libera, R. Antonello, D. Nikovski, R. Carli, and D. Romeres, "Model-based policy search using monte carlo gradient estimation with real systems application," *arXiv preprint arXiv:2101.12115*, 2021.

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

[14] M. P. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *ICML*, 2019.

[15] A. Dalla Libera and R. Carli, "A data-efficient geometrically inspired polynomial kernel for robot inverse dynamic," *IEEE Robotics and Automation Letters*, vol. 5, no. 1, pp. 24–31, 2019.

[16] D. Romeres, D. K. Jha, A. DallaLibera, B. Yerazunis, and D. Nikovski, "Semiparametrical gaussian processes learning of forward dynamical models for navigating in a circular maze," in *2019 International Conference on Robotics and Automation (ICRA).* IEEE, 2019, pp. 3195–3202.

[17] D. Romeres, M. Zorzi, R. Camoriano, and A. Chiuso, "Online semi-parametric learning for inverse dynamics modeling," in *2016 IEEE 55th Conference on Decision and Control (CDC).* IEEE, 2016, pp. 2945–2950.

[18] D. Nguyen-Tuong and J. Peters, "Using model knowledge for learning inverse dynamics," in *2010 IEEE international conference on robotics and automation.* IEEE, 2010, pp. 2677–2682.

[19] A. Dalla Libera, F. Amadio, D. Nikovski, R. Carli, and D. Romeres, "Control of mechanical systems via feedback linearization based on black-box gaussian process models," in *2021 European Control Conference (ECC)*. IEEE, 2021, pp. 243–248.

[20] Y. Gal, R. McAllister, and C. E. Rasmussen, "Improving pilco with bayesian neural network dynamics models," in *Data-Efficient Machine Learning workshop, ICML*, vol. 4, no. 34, 2016, p. 25.

[21] D. J. Mackay, "Bayesian methods for adaptive methods," *PhD thesis. California Institute of Technology*, 1992.

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[23] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," *Advances in neural information processing systems*, vol. 31, 2018.

[24] S. Geva and J. Sitte, "A cartpole experiment benchmark for trainable controllers," *IEEE Control Systems Magazine*, vol. 13, no. 5, pp. 40–51, 1993.

[25] M. Cutler and J. P. How, "Efficient reinforcement learning for robots using informative simulated priors," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 2605–2612.

[26] K. Chatzilygeroudis, R. Rama, R. Kaushik, D. Goepp, V. Vassiliades, and J.-B. Mouret, "Black-box data-efficient policy search for robotics," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 51–58.

[27] A. J. McHutchon *et al.*, "Nonlinear modelling and control using gaussian processes," Ph.D. dissertation, Citeseer, 2015.

[28] P. Parmas, C. E. Rasmussen, J. Peters, and K. Doya, "Pipps: Flexible model-based policy search robust to the curse of chaos," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4065–4074.

[29] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[30] D. J. Rezende, S. Mohamed, and D. Wierstra, “Stochastic backpropagation and approximate inference in deep generative models,” in *International conference on machine learning.* PMLR, 2014, pp. 1278–1286.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[32] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010.* Springer, 2010, pp. 177–186.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[34] M. P. Deisenroth, D. Fox, and C. E. Rasmussen, “Gaussian processes for data-efficient learning in robotics and control,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 2, pp. 408–423, 2013.

[35] C. E. Rasmussen, “Gaussian processes in machine learning,” in *Summer school on machine learning.* Springer, 2003, pp. 63–71.

[36] R. E. Caflisch, “Monte carlo and quasi-monte carlo methods,” *Acta numerica*, vol. 7, pp. 1–49, 1998.

[37] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[38] Y. Gal and Z. Ghahramani, “Dropout as a bayesian approximation: Representing model uncertainty in deep learning,” in *international conference on machine learning.* PMLR, 2016, pp. 1050–1059.

[39] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[40] A. Koubâa *et al.*, *Robot Operating System (ROS).* Springer, 2017, vol. 1.

[41] B. Siciliano, O. Khatib, and T. Kröger, *Springer handbook of robotics.* Springer, 2008, vol. 200.

[42] D. Coleman, I. Sucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study," *arXiv preprint arXiv:1404.3785*, 2014.

[43] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo, "ros_control: A generic and simple control framework for ros," *The Journal of Open Source Software*, 2017. [Online]. Available: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf

[44] E. Olson, "Apriltag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3400–3407.

[45] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[46] E. Marchand, F. Spindler, and F. Chaumette, "Visp for visual servoing: a generic software platform with a wide class of robot control skills," *IEEE Robotics and Automation Magazine*, vol. 12, no. 4, pp. 40–52, December 2005.

[47] R. Szeliski, *Computer vision: algorithms and applications.* Springer Science & Business Media, 2010.

[48] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, p. 381–395, jun 1981. [Online]. Available: https://doi.org/10.1145/358669.358692

[49] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2021.