Transfer of reinforcement learning for a robotic skill

Dulce Adriana Gómez Rosal

Computer Science and Engineering, master's level (120 credits)
2018

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Transfer of reinforcement learning for a robotic skill

Dulce Adriana Gómez Rosal

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo, August 8th, 2018

Supervisor

Professor Dr. Ville Kyrki

Advisor

M.Sc. Murtaza Hazara
# Abstract

In this work, we develop the transfer learning (TL) of reinforcement learning (RL) for the robotic skill of throwing a ball into a basket, from a computer simulated environment to a real-world implementation. Whereas learning of the same skill has been previously explored by using a Programming by Demonstration approach directly on the real-world robot, for our work, the model-based RL algorithm PILCO was employed as an alternative as it provides the robot with no previous knowledge or hints, i.e. the robot begins learning from a tabula rasa state, PILCO learns directly on the simulated environment, and as part of its procedure, PILCO models the dynamics of the inflatable, plastic ball used to perform the task. The robotic skill is represented as a Markov Decision Process, the robotic arm is a Kuka LWR4+, RL is enabled by PILCO, and TL is achieved through policy adjustments. Two learned policies were transferred, and although the results show that no exhaustive policy adjustments are required, large gaps remain between the simulated and the real environment in terms of the ball and robot dynamics. The contributions of this thesis include: a novel TL of RL framework for teaching the basketball skill to the Kuka robotic arm; the development of a pythonised version of PILCO; robust and extendable ROS packages for policy learning and adjustment in a simulated or real robot; a tracking-vision package with a Kinect camera; and an Orocos package for a position controller in the robotic arm.

**Keywords** Transfer learning, Reinforcement learning, Simulation, Robotics
Preface

It is a banal attempt, trying to frame an immense feeling as the gratitude can be. However, as a humble endeavour to achieve so, I want to express my gratitude to the SpaceMaster staff, who provided the necessary conditions and grant to pursue this 2-years journey. I would like to thank Professor Ville Kyrki for sharing his immense knowledge with me and to Murtaza Hazara for his insights in the development of this work. Special thanks go to Vesa Korhonen and Bill Hellberg, who indirectly also contributed to this production, and particularly to Jevgeni Antonenko who paid unique attention and care when using LWRSIM. Thanks also to my dear friends, whose camaraderie became an incesant lighthouse.

And moreover, this work is thanks and dedicated to my family, in every form. For us, no distance will ever be a frontier.

Otaniemi, August 8th, 2018
Dulce Adriana Gómez Rosal
Contents

Abstract ................................................................. 3
Preface ................................................................. 4
Contents ................................................................. 5
Acronyms and symbols .................................................. 13

1 Introduction .......................................................... 14
  1.1 Problem and solution overview .................................... 15
  1.2 Structure of the Thesis .............................................. 16

2 Background ......................................................... 17
  2.1 Learning a robotic skill ............................................. 17
     2.1.1 Markov decision processes .................................... 17
     2.1.2 Reinforcement learning ....................................... 18
     2.1.3 Gaussian processes ............................................. 19
     2.1.4 RL challenges for real robots ................................. 20
     2.1.5 PILCO ............................................................. 21
  2.2 Transfer of the learning ........................................... 25
     2.2.1 TL components .................................................. 26
     2.2.2 TL for simulated RL ............................................ 26

3 Transfer of reinforcement learning for a robotic skill .......... 27
  3.1 Modeling the robotic skill .......................................... 27
     3.1.1 The policy function ............................................ 28
     3.1.2 The states and actions ......................................... 30
     3.1.3 The cost function .............................................. 32
  3.2 Transferring the robotic skill ...................................... 34
     3.2.1 Comparison of the source and the target task .............. 34
     3.2.2 The policy adjustment ......................................... 36
  3.3 System overview .................................................... 40
     3.3.1 The robotic skill in the simulated environment ............ 40
     3.3.2 The robotic skill in the real-world environment .......... 42
     3.3.3 TL of RL for the real robot ................................... 48

4 Experiments and results ............................................ 52
  4.1 RL in the source task .............................................. 52
     4.1.1 Experiments with PILCO and LWRSIM ......................... 55
List of Figures

1.1 TL Framework of basketball skill from simulated environment with RL. (Simulation image is courtesy of Source [4]). .......................... 16

2.1 Agent-environment interaction in a MDP. Image taken from [8]. 18

2.2 PILCO main components (Source: [5]) ................................. 22

2.3 Closer look to PILCO functioning. ......................................... 23

3.1 Basketball as the robotic skill in the simulated environment. .... 28

3.2 Joint names and directions movement of KLR. .......................... 31

3.3 Three different executions of PILCO resulting in different costs. 34

3.4 Detail of ball and holder in source and target task. .................... 36

3.5 Policy adjustment (Source: [6]). ........................................... 38

3.6 ROS architecture for policy learning, every thick box represent a node. 41

3.7 Basketball as the robotic skill in the real-world environment. ...... 42

3.8 ROS architecture for non-adjusted policy execution in KLR, every thick box represent a node. .................................................. 43

3.9 Layout of KLR and Kinect camera. ........................................ 44

3.10 Schematic of an OROCOS component (a.k.a. Task in OROCOS terminology) (Source: [30]). .............................................. 46

3.11 System overview of the three hardware systems: the external computer, KRC and the arm KLR (Source: [31]). ............................... 47

3.12 ROS architecture for TL, every thick box represent a node. ....... 49

4.1 PILCO learning in MATLAB environment. ............................... 53

4.2 PILCO learning in MUJOCO environment. .............................. 54

4.3 LWRsim home position. .................................................... 57

4.4 Policy learned to approach ball to basket, a.k.a. lowvel 1 policy. 58

4.5 KUKA Lightweight Robot (KLR), Kuka Robot Controller (KRC 2lr) and KUKA Control Panel (KCP). ....................................... 59

4.6 Outcome from vision system, filtering the ball and the rim of the target basket. ................................................................. 60

4.7 KLR in home position. ....................................................... 60

4.8 Comparison of action from policy lowvel 1 on source task $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, target task with non-adjusted policy $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$ and target task with adjusted policy across every iteration $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$ for each Joint. ................................. 63
4.9 Comparison of position reached on source task and target task as result of $A_t$ in every case, for each Joint. .......................... 64
4.10 Comparison of state on source task $S_t^{(S)}$ and target task $S_t^{(T)}$ as result of $A_t$ in every case. .......................... 65
4.11 Comparison of execution cost on source task and target task as result of $A_t$ in every case. .......................... 66
4.12 Comparison of action from policy lowvel1 on source task $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, target task with non-adjusted policy $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$ and target task with adjusted policy across every iteration $A_t^{(T)} =$ $\pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$ for each Joint. .......................... 67
4.13 Comparison of position reached on source task and target task as result of $A_t$ in every case, for each Joint. .......................... 68
4.14 Comparison of state on source task $S_t^{(S)}$ and target task $S_t^{(T)}$ as result of $A_t$ in every case. .......................... 69
4.15 Comparison of execution cost on source task and target task as result of $A_t$ in every case. .......................... 70

A.11 Throwing the ball backwards at the beginning of the episode. . . . . 83
A.21 Reaching dangerous positions in LWRSIM. .......................... 84
A.22 PILCO local optimal result. .......................... 85
A.31 Learning to roll ball through the robot body and execution cost. . . . 86
A.41 Policy learned to throw the ball to the air. .......................... 87

B.11 Action from policy lowvel1 on source task, $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, for each Joint. .......................... 89
B.12 Position reached on source task as result of $A_t^{(S)}$, for each Joint. . . . . 89
B.13 State $S_t^{(S)}$ on source task as result of $A_t^{(S)}$. .......................... 90
B.14 Execution cost on source task as result of $A_t^{(S)}$. .......................... 90
B.21 Action from policy lowvel2 on source task, $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, for each Joint. .......................... 91
B.22 Position reached on source task as result of $A_t^{(S)}$, for each Joint. . . . . 91
B.23 State $S_t^{(S)}$ on source task as result of $A_t^{(S)}$. .......................... 92
B.24 Execution cost on source task as result of $A_t^{(S)}$. .......................... 92

C.11 Action from policy lowvel1 on target task, $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$, for each Joint. .......................... 94
C.12 Position reached on target task as result of $A_t^{(T)}$, for each Joint. . . . . 94
C.13 State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. .......................... 95
C.14 Execution cost on target task as result of $A_t^{(T)}$. .......................... 96
C.21 Action from policy lowvel2 on target task, $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$, for each Joint. .......................... 97
C.22 Position reached on target task as result of $A_t^{(T)}$, for each Joint. . . . . 97
C.23 State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. .......................... 98
C.24 Execution cost on target task as result of $A_t^{(T)}$. .......................... 99
D.11 Test set prediction $L_2 \rightarrow g(.)$ from policy $\text{lowvel1}$, for action on each Joint. First iteration. .................................................. 101
D.12 Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy $\text{lowvel1}$, for action on each Joint. First iteration. .................................................. 102
D.13 Comparison of action from policy $\text{lowvel1}$ before and after adjustment $\pi_{adj}$, for each Joint. First iteration. .................................................. 103
D.14 Action from adjusted policy $\text{lowvel1}$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S)(S_t^{(T)}))$, for each Joint. First iteration. .................................................. 103
D.15 Position reached on target task as result of $A_t^{(T)}$, for each Joint. First iteration. .................................................. 104
D.16 State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. First iteration. .................................................. 105
D.17 Execution cost on target task as result of $A_t^{(T)}$. First iteration. .................................................. 106
D.18 Test set prediction $L_2 \rightarrow g(.)$ from policy $\text{lowvel1}$, for action on each Joint. Second iteration. .................................................. 107
D.19 Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy $\text{lowvel1}$, for action on each Joint. Second iteration. .................................................. 108
D.110 Comparison of action from policy $\text{lowvel1}$ before and after adjustment $\pi_{adj}$, for each Joint. Second iteration. .................................................. 109
D.111 Action from adjusted policy $\text{lowvel1}$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S)(S_t^{(T)}))$, for each Joint. Second iteration. .................................................. 110
D.112 Position reached on target task as result of $A_t^{(T)}$, for each Joint. Second iteration. .................................................. 110
D.113 State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Second iteration. .................................................. 111
D.114 Execution cost on target task as result of $A_t^{(T)}$. Second iteration. .................................................. 112
D.115 Test set prediction $L_2 \rightarrow g(.)$ from policy $\text{lowvel1}$, for action on each Joint. Third iteration. .................................................. 113
D.116 Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy $\text{lowvel1}$, for action on each Joint. Third iteration. .................................................. 114
D.117 Comparison of action from policy $\text{lowvel1}$ before and after adjustment $\pi_{adj}$, for each Joint. Third iteration. .................................................. 115
D.118 Action from adjusted policy $\text{lowvel1}$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S)(S_t^{(T)}))$, for each Joint. Third iteration. .................................................. 116
D.119 Position reached on target task as result of $A_t^{(T)}$, for each Joint. Third iteration. .................................................. 116
D.120 State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Third iteration. .................................................. 117
D.121 Execution cost on target task as result of $A_t^{(T)}$. Third iteration. .................................................. 118
D.122 Test set prediction $L_2 \rightarrow g(.)$ from policy $\text{lowvel1}$, for action on each Joint. Fourth iteration. .................................................. 119
D.123 Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy $\text{lowvel1}$, for action on each Joint. Fourth iteration. .................................................. 120
D.124 Comparison of action from policy $\text{lowvel1}$ before and after adjustment $\pi_{adj}$, for each Joint. Fourth iteration. .................................................. 121
D.125 Action from adjusted policy $\text{lowvel1}$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S)(S_t^{(T)}))$, for each Joint. Fourth iteration. .................................................. 122
Position reached on target task as result of $A_t^{(T)}$, for each Joint.

Fourth iteration. .......................................................... 122

State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Fourth iteration. .... 123

Execution cost on target task as result of $A_t^{(T)}$. Fourth iteration. ... 124

Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel2, for action on each Joint. First iteration. .................................................. 125

Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel2, for action on each Joint. First iteration. .................................................. 126

Comparison of action from policy lowvel2 before and after adjustment

$\pi_{adj}$, for each Joint. First iteration. .................................................. 127

Action from adjusted policy lowvel2 on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S(S_t^{(T)})))$, for each Joint. First iteration. .................................................. 128

Position reached on target task as result of $A_t^{(T)}$, for each Joint. First iteration. .................................................. 129

State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. First iteration. .... 130

Execution cost on target task as result of $A_t^{(T)}$. First iteration. ... 131

Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel2, for action on each Joint. Second iteration. .................................................. 132

Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel2, for action on each Joint. Second iteration. .................................................. 133

Comparison of action from policy lowvel2 before and after adjustment

$\pi_{adj}$, for each Joint. Second iteration. .................................................. 134

Action from adjusted policy lowvel2 on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S(S_t^{(T)})))$, for each Joint. Second iteration. .................................................. 135

Position reached on target task as result of $A_t^{(T)}$, for each Joint.

Second iteration. ........................................................................ 136

State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Second iteration. .... 137

Execution cost on target task as result of $A_t^{(T)}$. Second iteration. ... 138

Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel2, for action on each Joint. Third iteration. .................................................. 139

Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel2, for action on each Joint. Third iteration. .................................................. 140

Comparison of action from policy lowvel2 before and after adjustment

$\pi_{adj}$, for each Joint. Third iteration. .................................................. 141

Action from adjusted policy lowvel2 on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S(S_t^{(T)})))$, for each Joint. Third iteration. .................................................. 142

Position reached on target task as result of $A_t^{(T)}$, for each Joint. Third iteration. .................................................. 143

State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Third iteration. .... 144

Execution cost on target task as result of $A_t^{(T)}$. Third iteration. ... 145

Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel2, for action on each Joint. Fourth iteration. .................................................. 146

Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel2, for action on each Joint. Fourth iteration. .................................................. 147
D.224 Comparison of action from policy lowvel2 before and after adjustment
\[ \pi_{adj}, \text{ for each Joint. Fourth iteration.} \] 148

D.225 Action from adjusted policy lowvel2 on target task, \[ A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S)(S_t^{(T)})) \],
for each Joint. Fourth iteration. 149

D.226 Position reached on target task as result of \[ A_t^{(T)} \], for each Joint.
Fourth iteration. 150

D.227 State \[ S_t^{(T)} \] on target task as result of \[ A_t^{(T)} \]. Fourth iteration. 151

D.228 Execution cost on target task as result of \[ A_t^{(T)} \]. Fourth iteration. 152
List of Tables

2.1 PILCO parameters ........................................... 24
3.1 Summary of the policy function ........................................... 30
3.2 State \(S_t\) and action \(A_t\) vector ........................................... 32
3.3 Summary of the cost function ........................................... 34
3.4 Summary of TL ........................................... 39
3.5 ROS nodes of \(learn\_policy\) package ........................................... 42
3.6 ROS nodes of \(vision\_ball\), \(rbf\_executor\) and \(adjust\_policy\) packages ........................................... 50
3.7 Summary of ROS packages and OROCOS component ................. 51
4.1 Results of different PILCO settings in different simulation environment for the cartpole task ........................................... 54
4.2 Parameters used to learn \(lowvel1\) and \(lowvel2\) policies ........................................... 57
4.3 MSE of test dataset from inverse-dynamics model \(L2\) and from adjusting-policy model \(L3\) across every iteration ........................................... 72
A.11 Parameters that led to the case in which LWRSIM kept throwing the ball backwards ........................................... 84
A.21 Parameters that led to the case in which LWRSIM gets stuck in local but dangerous optimals ........................................... 85
A.31 Parameters that led to the case in which LWRSIM learns to throw ball so it rolls through the robot’s body ........................................... 86
A.41 Parameters that led to the case in which LWRSIM throws the ball into the air towards the basket ........................................... 87
Acronyms and symbols

Acronyms

ML Machine Learning
MDP Markov Decision Process
RL Reinforcement Learning
TL Transfer Learning
SL Supervised Learning
GP Gaussian Process
RBF Radial Basis Function
PILCO Probabilistic Inference for Learning COntrol
MUJOCO Multi-Joint dynamics with Contact
LWRSIM Lightweight Robot Simulator, simulation of robotic arm Kuka LWR4+ in MUJOCO
KLR Kuka Lightweight Robot LWR4+
KRC Kuka Robot Controller
FRI Fast Research Interface
ROS Robot Operating System
CS Coordinate system
OROCOS Open RObot COntrol Software

Symbols

\( S_t \) state vector obtained from environment at time \( t \)
\( A_t \) action vector executed by agent at time \( t \) as result of \( \pi(S, \theta) \)
\( c_t \) cost obtained from the environment at time \( t \)
\( \pi(S, \theta) \) policy with state \( S \) as input and parametrized by vector \( \theta \)
\( S/A_t(S/T) \) state or action vector at time \( t \) obtained from the source (S) or the target (T) task
\( g(\cdot) \) Model of inverse ball dynamic
\( \pi_{\text{adj}} \) Policy adjustment
\( \vec{p}_1 \) vector point referenced to CS 1
\( ^1R_2 \) rotation matrix from CS 1 to 2
\( ^1M_2 \) extended matrix for rigid body displacement from CS 1 to 2
Chapter 1

Introduction

It is undeniable that robots are nowadays commonplace as useful tools. Although they have been around for a considerable time, the way to programme them keeps changing with the introduction of new paradigms. As robots serve different needs, they can either be “hard-coded” to perform specific tasks, or they can be programmed to respond to an environment according to specific guidelines, or a policy. Such “flexible programming” for robot task learning can be enabled by machine learning (ML), and more specifically, by reinforcement learning (RL).

Due to the huge range of applications that robotics can serve, the need for such flexible programming is in great demand. RL has arisen as a promising technique for teaching robots certain skills or tasks, especially those that can not be programmed by hand (i.e. hard-coded) and for which it is easy to specify a reward or cost function, i.e. resembling in a way how humans learn through repetitions aiming for either high reward or low cost.

Despite their advantages, RL repetitive methods are discouraged in real robot implementations due to potential physical problems[1] (e.g. the considerable number of repetitions required in the experiments might lead to mechanical wear out, or the robot’s responses might vary due to increasing heat caused by the repetitions). Due to this, there is a trend to perform the RL process in a simulated environment before its implementation in the real robot. Such a procedure requires transfer of the knowledge learned from the simulation to the real-world environment. This can be accomplished with a technique known as transfer learning (TL), which consists of processing knowledge from a source task (the simulated environment) towards a target task (the real robot environment) such that the robot will display the same behaviour in the target task as in the the source task[1]. Since this is the heart of this thesis, a detailed description is presented in Section 2.
A robotic skill is the ability to perform a task that can be determined by a skill parameter; for example, to teach a robot how to throw a ball into a basket, the skill is to throw the ball and the skill parameter is the distance from the robot to the basket.

The skill is demonstrated by following a policy, which is the set of actions that the robot needs to execute in order to perform the task[2]. The policy can be represented in many ways, and for this work, a Radial Basis Functions (RBF) representation is selected.

Our basketball research provides an interesting benchmark for the learned skill, since it can be straightforward modeled as an RL problem. Correspondingly, the associated cost becomes intuitive and visually easy to track, as it is described by the distance from the place where the ball hit the floor, to the location of the target basket. The learning of basketball skill has been previously explored in [3] by using a Programming by Demonstration (PbD) approach, which provided the robot with expert knowledge by means of a human physically moving the robot in the real world. However, for our work, an alternative RL scenario is modeled and an algorithm with no previous knowledge or hints is of interest, i.e. an algorithm where the robot learns from a tabula rasa state. This algorithm is introduced in the following section.

1.1 Problem and solution overview

This thesis develops and evaluates a framework for the TL of a robotic skill learned in a simulated environment through an RL algorithm. The employed robot is the industrial robotic arm KUKA LWR4+ and the robotic skill is the throwing of a plastic, inflatable ball into a basket located at a certain distance in front of the robot. A succinct overview is displayed in Figure 1.1.

The PILCO[5] algorithm is used to learn the robotic skill, through a policy, in the simulator without previous knowledge. Afterwards, a supervised learning (SL) algorithm is employed to adjust the policy execution in the real-world environment. With this adjustment[6], it is sought that the real environment performs similarly to the simulated environment.

The contributions of this thesis are the development of a TL and RL framework for teaching the basketball skill to the Kuka robotic arm LWR4+ (KLR), a pythonised version of the PILCO algorithm, robust and extendable ROS packages for policy learning and optimization in a simulated or real robot, a tracking-vision package with a Kinect camera, and an Orocos package for position control implementation in the robotic arm.

This work is of interest for various reasons: usage of an RL algorithm learning from a tabula rasa state for this robotic skill; the proposed Markov decision process
Figure 1.1: TL Framework of basketball skill from simulated environment with RL. (Simulation image is courtesy of Source [4]).

(MDP) and the inputs-outputs selection for the algorithm are interesting for both RL and TL as it considers at the same time the ball state as the input for the policy and as the describer of the feedback cost (which after the robot looses grip of the ball, the policy stops having any effect on it); transferring knowledge from a simulated environment to the real-world is challenging as it relies on a system capable of tracking similarly the ball state; and finally, due to PILCO’s usage, we chose to learn the dynamics of the inflatable, plastic ball instead of the robot’s dynamics as we rely on the robot simulation fidelity.

Methods used to validate and verify the outcomes are the achievement of the goal, the cost function and comparison of the results between the simulation and the physical robot.

1.2 Structure of the Thesis

This document is divided in five chapters. In the second chapter, the necessary theoretical background is presented, whereas third chapter shows the structural solution and implementation. Chapter four introduces the obtained results and pertinent analysis, while the conclusions and proposed future work are summarized in chapter five. Afterwards, references and relevant bibliography are presented and the document ends with Appendices where outcomes of every experiment are detailed in their plots.
Chapter 2

Background

Sin claridad, no hay voz de sabiduría.

Sor Juana Inés de la Cruz

This chapter presents an overview of Markov decision processes (MDP) and reinforcement learning (RL) in Section 2.1, introducing how they are of use when modeling the robotic skill. Necessary concepts are grounded and given that PILCO is the RL algorithm to be used, it is presented. Finally, a state of the art overview of transfer learning (TL) for robotics is provided in Section 2.2.

2.1 Learning a robotic skill

In between the many benefits that ML has given to various fields and disciplines, robotics has received with arms wide open a particular ML contribution: reinforcement learning (RL)[7]. To understand how RL can be useful for robotics, it is pertinent to review Markov decision processes (MDP) succinctly first.

2.1.1 Markov decision processes

To introduce the MDP necessary notions, let us define the following concepts:

- **Agent** is the unit in an environment (or scenario) who is able to interact with it through an action and to receive a reward as a result of the action.

- **Environment** is the playground where the agent performs its actions and provides feedback to the actions through a reward or cost.

- **Action** $A$ is an activity that the agent can execute on the environment. It is typically a multidimensional vector containing actions for each dimension. The available actions are contained in the set of all actions $A$, which can be a discrete or continuous space.
• **State** The multidimensional vector $S$ is the way in which the environment communicates its current information. Such information lets the agent decide the action to execute at the next timestep. Depending on the environment, the set of all states $S$ can be a discrete or continuous space.

• **Reward $R$** is how “well” received an action is by the environment. It varies according to a reward function and therefore, $R_t$ represents the reward obtained from the environment at time $t$. Depending on the perspective and the implementation, a reward is also known as the inverse cost $c$ and hence, a cost function is related to the inverse of the reward function.

When a phenomenon can be described by the interaction of an “active” agent with a “responding” environment, and that interaction can get characterized by the flow of actions, states and rewards or cost, this phenomenon is said to be an MDP. An MDP is represented by $(S, A, T, c)$, where $T$ is the transition function between $S$ and $A$. Figure 2.1 shows an MDP with an agent, whom according to the current state $S_t$, takes an action $A_t$ to which the environment reacts and returns a new state $S_{t+1}$ and reward $R_{t+1}$. Given the new state and reward, the agent chooses the next action, and the loop repeats until the environment is solved or terminated (depending if the environment is episodic or not). If the interaction is time-bounded, every full interaction agent-environment is called an episode.

![Figure 2.1: Agent-environment interaction in a MDP. Image taken from [8].](image)

### 2.1.2 Reinforcement learning

RL is a ML framework which allows to learn a behaviour to maximize or minimize the payback according to a reward or cost function. Since rewards are mostly gratifying functions, an RL algorithm tries to maximize it. However if the MDP establishes a cost function instead of a reward function, then, the opposite happens, the algorithm aims to minimize it. In our implementation, a cost function $c(S_t)$ is used and associated to the cost of being in a state $S_t$ similarly to what is demonstrated in
Equation 2.1.

\[ c(S_t) = ||S_{target} - S_t|| \]  

To make clear the rest of the RL terminology, the following concepts are introduced. A **policy** \( \pi \) is a mapping from the state vector to the probabilities of selecting each possible action[8]. It can also be understood as the way in which the agent acts according to the environment state. It can be approximated with a policy function \( \pi(S, \theta) \) which gets shaped by the policy parameters \( \theta \) and use the state \( S \) as input. Similarly, the state of the environment according to the executed actions can be approximated with a value function. Such schemes allows the introduction of actor-critic methods[8], which are policy-gradient algorithms that learn approximations to both actor (policy) and critic/environment (value) functions. This is highly useful because it allows the usage of mature gradient algorithms to perform the search for an optimal policy.

Hence, for policy-search methods in episodic tasks, RL’s objective is to learn an optimal policy \( \pi^*(S, \theta) \) through the finding of optimal parameters \( \theta^* \). This optimal policy will achieve a minimal expected long-term cost \( J^*(\theta) \) such as Equation 2.2 presents.

\[ J^*(\theta) = \sum_{t=0}^{T} E[c(S_t)|\pi] \]  

While the **control problem** in RL consists of finding an optimal policy that minimizes the cost, the **prediction problem** emphasizes the policy evaluation, i.e. aims to estimate the value function for a given policy \( \pi \). Both concepts will be useful when describing PILCO, the RL algorithm employed to learn the optimal policy on the simulated environment. Before getting there, let us say some words about the function approximators for the value and policy functions used in this work.

### 2.1.3 Gaussian processes

Function approximators are used to keep a compact version of mathematical functions. This is particularly important when using RL in complex environments and hence, approximators have been historically involved representing value functions. This approach can be applied in discrete time or continuous state space systems, which avoids the discretisation of state spaces often required by many classical methods[9]. For instance, in [10], kernel based methods (support vector regression) were applied for learning the value function for a discrete state space.

Gaussian processes (GP) are models based on Gaussian distributions (Equation 2.3), capable of automatically adjusting features based on the observed data. They have been successfully employed in high-dimensional approximate RL domains[11] and to the best knowledge of the author, existing RL methods with GPs are restricted
to on-policy learning (for a detailed article on GP problems for approximate RL convergence, off-policy learning and exploration, a curious reader can look into [12], [13]).

\[ X \sim \mathcal{N}(\mu, \sigma^2) \]  

For the needs of this work, GP regression models as function approximators in continuous state-spaces and discrete time are of interest since they can be used for two distinct purposes:

- to model the dynamics of the system dynamics GP (value function or environment model) and
- to model the policy function and obtain the recommended action.

Since the policy function determines the action to be pursued, it is also named as the controller. However, in contrast with a linear controller \( Ax + b \) (from the classic control theory), our policy function is a non-linear controller determined by a full deterministic GP model. We describe a full deterministic GP model as a GP model where the posterior uncertainty about the underlying function gets ignored leaving only the mean function. Hence, with the GP model consisting of only the mean function, it becomes functionally equivalent to a Radial Basis Function (RBF) network.

An RBF network is a linear combination of Gaussian functions

\[ \psi_i = \exp\left(-\frac{||x - x_i||^2}{2\sigma^2}\right) \]  

used as basis functions for a regression problem. It typically consists in that given a set of points \( x \), the problem is to look for the corresponding function in the form of a linear combination of the basis functions. Since a typical RBF network has the form that Equation 2.5 shows,

\[ f(x) = \sum_{i=0}^{P} \theta_i \psi_i(x) \]  

to find the desired function consists of finding the right parameters vector \( \theta = [\theta_0...\theta_i...\theta_P]' \), also known as weight vector. Hence, for our case, to find the right policy function refers to find the right parameters \( \theta \).

2.1.4 RL challenges for real robots

Several methods for RL have been proposed, and when it comes to real robot implementations, certain challenges such as state- and action-space dimensionality, real world environment and safety of exploration arise.
The problem of dimensionality refers to the high-dimensional continuous state- and action-space possible for a robotic task. The situation gets more complicated if the robot is also set to learn a variety of tasks, as for each one, a special policy can be required[14], while for similar tasks, a generalizable policy can be employed[2].

RL on real robots pose a problem in different directions. One of them is the problem of exploration[15], which if disregarded, can result in a damage for the robot (just to learn that it was not the right learning direction). Along this, an execution of the policy on a real robot could lead to high costs of data generation (mechanical wear out and physical heating), noisy measurements derived from either noisy sensors (data retrieval) and or noisy readings (data transmission).

An algorithm that would overcome such challenges in a real robot would be ideal, and although several proposals have been made, due to its characteristics, PILCO was selected to perform the initial RL for our implementation. Such characteristics are summarized as scalability, exploration and data-efficiency, and further developed as follows.

2.1.5 PILCO

The Probabilistic Inference for Learning Control algorithm[5], [16] (PILCO) is a policy-gradient method that performs policy searches for a dynamic system $S_t = f(S_{t-1}, A_{t-1})$ with unknown transition dynamics $f$. Its objective is to find a deterministic policy $\pi(\theta)$ that minimizes $J^*(\theta)$ (Equation 2.2) by obtaining optimal parameters $\theta^*$. It uses SL to generate a probabilistic model of the environment dynamic (as a GP) and performs a gradient optimization as policy-search to deliver an optimal policy function $\pi(\theta^*)$.

PILCO is a model-based learning algorithm, and as such, it retains transition information during the learning process to build a model of the environment. In contrast, model-free algorithms do not learn an environment model and are mostly used in situations where is impossible to fresh-start the environment, having to learn as a single continuous process.

Despite PILCO’s need of storing this transition information, it is said to be data-efficient[16] since it needs few samples to start delivering an acceptable policy. PILCO is able to learn controllers with hundreds of parameters for high-dimensional systems since it uses a learned probabilistic GP dynamics model. The algorithm also aims to be robust to model errors and therefore, the learned dynamics model is expressed in the GP posterior and the uncertainty of this learned dynamic model is explicitly taken into account for multiple step forward predictions, policy evaluation and policy improvement. PILCO is applied for learning without expert knowledge as prior and part of its strategy consists of learning a single controller for all control dimensions jointly by taking the correlation of all control and state dimensions into
account during planning and control. Hence, it requires a probabilistic function approximator (GP) for the probabilistic dynamic model that it develops.

To understand PILCO’s functioning, Algorithm 1 shows PILCO pseudocode and Figure 2.2 displays its main elements. A single optimization is composed by the consecutive execution of these three stages: Model Learning, Policy Learning and Policy Application, from where the cycle repeats until a convergence index or until a number of optimizations $N_{opt}$ is reached.

In the initialization stage, a random policy is executed with random parameters $\theta$ to gather data. Such data is then used to train the GP that will constitute the dynamic model. The second stage is policy learning, which consists of policy evaluation and improvement, whereas last stage applies the learned policy to the system. This execution is used to collect data and from there, update the model and restart the cycle until the task is learned or until an optimization limit is reached.

\begin{algorithm}
\textbf{Data:} an unknown dynamic system $S_t = f(S_{t-1}, A_{t-1})$
\textbf{Result:} An optimal policy function $\pi^*$
\begin{algorithmic}
\State \textbf{init:}
\State Sample controller parameters $\theta \sim \mathcal{N}(0, I)$,
\State apply random control signals
\State record data for training
\While {task is not learned or $N_{opt}$ is not reached}
\State Learn probabilistic GP dynamics model using training data
\State Model-based policy search:
\While {not convergence}
\State Approximate inference for policy evaluation: get $J(\theta)$
\State Gradient-based policy improvement: get $dJ(\theta)/d\theta$
\State Update parameters $\theta$
\EndWhile
\State return $\theta^*$
\State $\pi^* \leftarrow \pi(\theta^*)$
\State Apply $\pi^*$ to system for an episode and record data for training
\EndWhile
\State $\pi^*$ is obtained
\EndAlgorithm
\end{algorithm}

Algorithm 1: PILCO algorithm (Source: [5])

A closer look to the internal stages is reflected in Figure 2.3: In the first stage, the
dynamic model is approximated by learning a non-parametric, probabilistic GP from the gathered data, i.e. from the states $S_t$ and policy actions $A_t$. The non-parametric property of the GP does not require an explicit task-dependent parametrization of the dynamics of the system and the probabilistic property of the GP reduces the effect of model errors[5]. Afterwards, the policy function is optimized by gradient-minimizing the observed costs $c_t$ and actions $A_t$, from which, new parameters $\theta$ are proposed and hence, executed on the system to gather new data and restart the cycle.

Policy improvement takes place when the policy gets optimized through the training data. This is accomplished by using the gradient-based Quasi-Newton optimization method Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LM-BFGS) for parameter estimation and the specifications for this is configured through the parameters optimization length and MFEPLS which set the maximum number of line searches after which the optimizer returns the best parameter set so far, and the maximum number of function evaluations per line search (MFEPLS), respectively. The policy gets evaluated by using 2.2 and the required gradients with respect to the policy parameters are computed analytically. After policy optimization, policy application is performed with the newly learned controller, hence, policy provides the required action $A_t$ according to the retrieved state $S_{t−1}$ at every timestep $d_t$ (forward controller). This policy application is called an execution or rollout and allows to record the generated trajectory of state-action pairs, from which the training inputs and targets for the GP model are extracted.

Although PILCO is a learning algorithm that relies mainly its performance in the feeded data ($S_t$ and $A_t$ in our case), the parameters displayed in Table 2.1 can modify its execution. Although a succinct description is provided, the curious reader can find more details in PILCO’s documentation[17]. Section 4 presents results of their different configurations.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment timestep $d_t$ [s]</td>
<td>Sampling time, inverse of environment frequency, rate at which inputs and outputs are sampled.</td>
</tr>
<tr>
<td>Training timestep $d_{training}$ [s]</td>
<td>Sampling time of training data for model learning, determines the size of training dataset. $d_{training} \geq d_t$</td>
</tr>
<tr>
<td>Episode time $T$ [s]</td>
<td>Timelapse of a complete episode.</td>
</tr>
<tr>
<td>Number of RBF kernels $n_{kernels}$</td>
<td>Number of basis functions that represent the RBF policy.</td>
</tr>
<tr>
<td>Number of optimizations $N_{opt}$</td>
<td>Number of iterations for which PILCO executes a policy-search for the optimal policy $\pi(\theta^*)$</td>
</tr>
<tr>
<td>Maximum U $u_{max}$ [deg/s]</td>
<td>Vector containing the absolute value for the maximum possible output allowed by the policy. In our case, it is the maximum angular velocity for every joint (for details, see Section 3.1.2).</td>
</tr>
<tr>
<td>Optimization length</td>
<td>Maximum number of line searches after which the non-convex gradient-based optimizer returns the best parameter set so far.</td>
</tr>
<tr>
<td>Optimization MFEPLS</td>
<td>Maximum number of function evaluations per line search. Either the line search succeeds by finding a parameter set with a gradient close to 0 or it does not succeed and aborts after $n$ optimization MFEPLS many functions (and gradient) evaluations.</td>
</tr>
</tbody>
</table>

Table 2.1: PILCO parameters
2.2 Transfer of the learning

As mentioned in Section 2.1.4, unlike other well controlled RL applications, challenges for RL in robotics implementations require a special treatment. Therefore, Transfer Learning (TL) can leverage the transition of learning from a source environment towards the real-world execution.

TL refers to the procedure in which a transfer of information from a task learned on a machine is lead onto another machine[18]. As such, and similarly to RL, it is a huge field with plenty of possibilities given the differences that both machines can show, how similar the task should be, and even in how much data the first machine can access to, compared to what is available for the second machine.

In its simplest form, TL allows that the experience gained in a source task is expected to help in the learning process for a similar task. Another example can be found in a multi-task learning, in which a single model can solve multiple tasks and TL involves knowledge transfer from the solution of a simpler task to a more complex one, or from a task where there is more data to one where there is less data[18]. Most ML systems solve a single task and therefore, it can be said that TL is a step towards artificial intelligence in which a single program can solve multiple tasks.

TL can also be a method of using additional knowledge to accelerate learning. In such a layout, TL operates by taking knowledge from the process that supplies information and reusing it in a target problem aiming to reduce the amount of learning to achieve optimal results. Many more TL schemes exist depending on the problem settings and layout, and a good compass for these possibilities can be reviewed in [1]

Towards trying to find a solution for the RL challenges devised in the previous subsection and continuing with the solution provided by the PILCO learning in a simulated environment, the next aim to overcome such challenges is to enable a TL suitable from simulation learning to its real world implementation.

Imitation processes are also considered TL and works involving them and dynamic motor primitives have shown good results[19]. Dynamic motor primitives is a popular approach since their publication[20] as to represent dynamical systems as a general approach of representing control policies for basic movements. With this, some basic motor skills are learned focused on learning by imitation without subsequent self-improvement, except [21] and [22]. Yet another approach known as Apprenticeship[23] shows an interesting alternative, since it allows a teacher to make a demonstration and gets the learner to perform inverse RL.
2.2.1 TL components

Two components are of main importance for TL: a source and a target.

The source refers to the components present in the initial configuration: the source system, source task, source learning and a priori information. The target refers to their analogs: target system, target task, target learning, biased knowledge. The relation between source and target defines the whole TL process.

An example of TL can be found in [2], where the ball-in-a-cup experiment was performed with varying lengths of the cord. The source task was the learned execution for some cord lengths and the target tasks were the extrapolated lengths within a range. In this case, the transfer method was a Linear Weighted Regression (LWR). Finding such a direct relation might not always be the case, therefore every transfer method may be formulated differently.

When the experience gained in the source task is expected to help the policy execution in the target task, TL uses a transfer method that bias the priori and process it. If source and target tasks are very similar, the priori just needs some pre-processing and TL can only adjust the policy towards its execution on the target task.

2.2.2 TL for simulated RL

For the system proposed in this Thesis, the knowledge analyzed by the transfer method will be collected from the source task in a simulated environment. This knowledge is the set of policy parameters that will generate the expected behaviour, i.e. the policy \( \pi(\theta) \) that generates the actions \( A_t \) corresponding to the observed state \( S_t \). The transferred knowledge should replicate the skill taught to the robot in the simulation and is constrained by the physical capabilities that the robot has. It can be said then, that the robot behaviour is represented by a parametric policy, and such policy is described by its policy parameters.

The basketball skill proposed in [2] and [3] appears as an appealing benchmark for RL algorithms, and given the similarities that the source task (simulation) and the target task (real robot) post, the transfer method through a policy adjustment driven by SL[6] from executions in the source system represents an interesting alternative. This proposal poses a similar idea to [24], except that in it, a trajectory optimization algorithm is used to generate the training trajectories to build the dataset and drive the gradient of the policy-search. Unlike [6], this work represents a different challenge since the dynamic system to be learned by the algorithms is highly random and noisy, and the policy-search space is sensibly larger. Details about this statement are found in Section 3.2.2.
Chapter 3

Transfer of reinforcement learning for a robotic skill

This chapter presents the design and implementation details of the complete pipeline that initially learns the robotic skill in a simulated environment as source task and then adjusts the learned policy towards its execution in the physical robot as the target task. The skill as an RL problem is stated in the policy and cost function, defined in MDP and PILCO terminology in Section 3.1. Towards the aforementioned objective and similarly to RL, the adopted TL technique is specified and exposed in 3.2. With all the strategy displayed, Section 3.3 discloses the software developed and used to embody the complete proposal. Project specifications such as mathematical functions, parameters and software are exposed in this chapter.

3.1 Modeling the robotic skill

The skill used in this work as the benchmark to test the learning algorithms and adjustments is the so-called basketball skill and it refers to the task in which the Kuka robotic arm (KLR) throws a blue, plastic, inflatable ball in such a way that it falls into a red basket (bucket) located at a certain distance in front of the robot. As introduced, this skill was selected due to its simple statement as an RL problem for which the feedback reward is intuitive: a decreasing cost is obtained if the robot maneuvers the ball in a way that it falls into the basket.
The initial setting of every episode is depicted in Figure 3.1 and comprises the distance of the basket from the robot base, the plastic ball to be thrown, the robot home position (which sets the ball initial position), the ball handler used as tool in the end effector of the robotic arm and the dimensions of the table on top of which the robot is placed. These settings are detailed in Section 3.3.1 for the simulated environment and in Section 3.3.2 for the real world implementation.

3.1.1 The policy function

As introduced in Section 2.1.2, a policy defines the relation between the state vector $S_t$ and the corresponding actions vector $A_t$. For this work, such a policy function acts as a feedforward policy and is represented with an RBF network of 100 kernels for smoothness reasons. Same policy function is adopted for the simulated and for the real environment.

This learned state-feedback controller policy $\pi(S_t, \theta)$ is defined by the Equation 3.1 and it can be noticed that it corresponds to the learned policy $\tilde{\pi}(S_t, \theta)$ after a postconditioning process. Such a process is the scaling of $\tilde{\pi}(S_t, \theta)$ to the limits imposed by $u_{max}$ by using $\sigma$, in Equation 3.2 (third-order Fourier series expansion of a trapezoidal wave), as a squashing or limiter function that maps the outcomes of $\tilde{\pi}(S_t, \theta)$ to [-1,1]. Equation 3.3 shows the raw version of the policy as an RBF network, where $c_i$ are the centers of the Gaussian basis functions, $n_{kernels}$ is the number of kernels and $W$ is the weight matrix for the states that determines its preponderance[17].
\[ \pi(S_t, \theta) = u_{max}\sigma(\tilde{\pi}(S_t, \theta)), \tag{3.1} \]
\[ \sigma(x) = \frac{9\sin(x) + \sin(3x)}{8}, \tag{3.2} \]
\[ \tilde{\pi}(S_t, \theta) = \sum_{i=1}^{n_{\text{kernels}}} \theta_i \exp\left(-\frac{1}{2}(S_t - c_i)^T W (S_t - c_i)\right), \tag{3.3} \]

PILCO performs the policy search to propose good parameters \( \theta \) by using a gradient-based optimizer on a set of variables. This set of variables is composed by the policy inputs, the policy target definition and the hyperparameters. While the target definitions are always set with values close to zero due to their usage as GP training targets, the policy inputs correspond to the centers \( c_i \) of the policy in Equation 3.3 and become the training inputs of the GP for the optimization process. Since the centers \( c_i \) correspond to an RBF, the initial locations of the centers are sampled from the initial state distribution \( p(S_o) \) as an initial \( \mu_0 \).

The GP hyperparameters are the most important values of the set, since they get modified after every optimization and hold intrinsically the new parameters \( \theta \) for the policy function after every optimization. They are stored in a logarithmic scale (for the GP functions) and act as the GP hyperparameters of log-length-scales, log-signal-standard-deviation and log-noise-standard-deviation.

The policy function is summarized in Table 3.1 and more information about implementation details can be found in PILCO’s code documentation[17].
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy function</td>
<td>Equations 3.1 and 3.3</td>
</tr>
<tr>
<td>Maximum U ( u_{max} ) [deg/s]</td>
<td>Absolute value for the maximum output from the policy</td>
</tr>
<tr>
<td>Input to optimizer: inputs</td>
<td>Training input for the GP, correspond to the centers ( c_i ) of policy function</td>
</tr>
<tr>
<td>Input to optimizer: targets</td>
<td>Training target of the GP, normally set to values close to zero</td>
</tr>
<tr>
<td>Input to optimizer: hyperparameters</td>
<td>Variables that shape the policy function and on which the optimizer works, correspond to the GP logarithmic hyperparameters</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of the policy function

#### 3.1.2 The states and actions

The choice of the state and action vector is crucial as it determines how the complete problem is defined. Insight of this, our state definition needed to be measurable at every timestep for both environments. Instead of considering the angular position of every robot joint as a state (as in most robotic RL problems), we considered the tracking of the ball cartesian position \( b = [b_x, b_y, b_z]' \) and velocity \( \dot{b} = [\dot{b}_x, \dot{b}_y, \dot{b}_z]' \) as physical describers of the ball for the state vector. In this way, the dynamic model learned by PILCO (presented in Section 2.1.5) is not the robot dynamics but the ball dynamics. This choice introduced a new difficulty level, since instead of using the well-determined robot dynamics, the policy gets based on a plastic ball, an element that can be influenced by different physical factors.

The actions determination was done based on a human-similarity reason. As Figure 3.2 displays, KLR joint configuration allows to have similar movement to a human arm, hence, a human arm throw movement was considered as guideline for the ball throwing. In order to reduce the space search for PILCO, only joints A2, A3 and A5 were considered as movables, whereas the rest of the joints get fixed with
their home position. To avoid any terms confusion, we refer to joints A2, A3 and A5 as J2, J4 and J6 respectively. Accordingly, actions delivered by PILCO are angular velocities $\dot{q}_i$ for every joint $i$ that get converted into angular position $q_i$ for every timestep $\Delta t$ as Equation 3.4 shows.

$$q_i = q_{i-1} + \dot{q}_i \Delta t$$  \hspace{1cm} (3.4)

The consideration of angular velocities as policy actions allowed to drive the search exploration from each position at every time step, i.e. it allowed to perform the policy search through forward or backward movements from the current position. This enabled a steadier policy search compared to one in which actions are joint positions leading into a bouncy policy-search.

As a summary state $S_t$ and actions $A_t$ vector are exhibited in Table 3.2.
State vector

\[ \mathbf{S}_t = [b_x, b_y, b_z, \dot{b}_x, \dot{b}_y, \dot{b}_z]' \]

where \( b_x, b_y, b_z \) and \( \dot{b}_x, \dot{b}_y, \dot{b}_z \in \mathcal{S} \)
are ball cartesian position [m] and velocity [m/s] respectively.

Action vector

\[ \mathbf{A}_t = [\dot{q}_i]' \]

where \( \dot{q}_i \in \mathcal{A} \) is the angular velocity [rad/s] for each joint \( i = [A2 = J2, A3 = J4, A5 = J6] \) of KLR.

Table 3.2: State \( \mathbf{S}_t \) and action \( \mathbf{A}_t \) vector

3.1.3 The cost function

As introduced in Section 2.1.2, Equation 2.1 represents the cost associated of being in a certain state \( \mathbf{S}_t \) distinct from the target state \( \mathbf{S}_{target} \) for a time \( t \). In a similar fashion, we define the cost function in Equation 3.5 as the distance penalization between the ball and the basket, where \( \sigma_c \) is the spread. An unnormalized Gaussian function substracted from 1 represents the cost function as

\[ c(\mathbf{S}_t) = 1 - \exp\left(-\frac{||\mathbf{S}_t - \mathbf{S}_{target}||^2}{2\sigma_c^2}\right) \in [0, 1] \]  (3.5)

For implementation purposes, the cost-function definition was determined by practical factors such as the initial KLR position and the way PILCO performs its policy-search. These practical factors are as follows:

- We learn the dynamics of the ball. The ball is loosely fixed to the robot and is made of plastic that gets naturally deflated and easily affected by a number of factors.

- The skill is expected to be learned in such a way that, from the initial configuration, the robot and the ball get moved and at a given time, the robot throws the ball towards the basket.

- The moment in which the robot stops being coercive for the ball (robot loses grip of the ball as result of the previous robot movements and ball’s inertia) is called \( t_{ball\_left\_robot} \) and is important because after it, the cost function is still of interest, whereas the policy actions can be disregarded.
• No bounces are allowed, the robot is expected to throw the ball directly into the basket. The time when ball bounces for the first time on a body gets called \( t_{ball\_bounced} \).

Insight of these, the cost function was implemented such that the execution cost \( c(S_t) \) (cost obtained from a policy execution, Equation 3.5) is calculated and stored from the beginning until time \( t_{last\_cost} \). The execution costs, \( S_t \) and \( A_t \) of the episode until time \( t_{last\_cost} \) are then retrieved by PILCO as part of the episode data and used to increase the training dataset for the ball dynamics model and to perform a new policy-search optimization as described in Section 2.1.5.

The moment \( t_{last\_cost} \) is defined in Equation 3.6 where \( T \) is the complete episode duration.

\[
\begin{align*}
t_{last\_cost} &= \begin{cases} 
  t_{ball\_bounced} & \text{if } t_{ball\_bounced} > t_{ball\_left\_robot} \\
  t_{ball\_left\_robot} & \text{if } t_{ball\_left\_robot} < T \\
  T & \text{otherwise}
\end{cases}
\end{align*}
\] (3.6)

The episode normally lasts longer than \( t_{last\_cost} \) (except in the case where the robot moves so smooth that it never drops the ball) and therefore the expected cost \( L(S_t) \) is calculated for the complete episode according to Equation 3.7. This cost is calculated from the expected long term cost \( J^\pi(\theta) \) plus the contribution of the exploration parameter \( k \), where exploration is encouraged by setting \( k < 0 \).

\[
L(S_t) = \mathbb{E}_{S_t}[c(S_t)] + k\sqrt{\text{Var}_{S_t}[c(S_t)]}
\] (3.7)

In Figure 3.3, three different cost functions are result of different optimizations. In color blue, the expected cost is predicted after which the policy is executed. As result of the policy execution, the execution cost is calculated and printed in red over the predicted cost.

A summary of the cost function is presented in Table 3.3.
Figure 3.3: Three different executions of PILCO resulting in different costs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost function</td>
<td>Equation 3.5</td>
</tr>
<tr>
<td>$t_{\text{last_cost}}$</td>
<td>Equation 3.6</td>
</tr>
<tr>
<td>Execution cost</td>
<td>Cost obtained from a policy execution as result of Equation 3.5 until time $t_{\text{last_cost}}$</td>
</tr>
<tr>
<td>$c(S_t)$</td>
<td></td>
</tr>
<tr>
<td>Expected cost</td>
<td>Cost calculated from the expected long term cost of Equation 3.7. Is calculated for the complete episode for the prediction of how optimal the policy is expected to be.</td>
</tr>
<tr>
<td>$L(S_t)$</td>
<td></td>
</tr>
<tr>
<td>Target state</td>
<td>Is defined by the basket position as the position that the ball should reach. No target velocity is considered.</td>
</tr>
<tr>
<td>$S_{\text{target}}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Summary of the cost function

3.2 Transferring the robotic skill

3.2.1 Comparison of the source and the target task

Taking Section 2.2 as foreword, we shall now describe the transfer process that we call *policy adjustment*. Most of the transfer implemented was inspired by [6], who similarly to PILCO, performed the published experiments in simple scenarios such as cart pole and inverted pendulum.
In our case, the aim is to reproduce the policy learned and executed in the simulated environment LWRSIM as source task, in the real robot KLR as target task. Although the implementation details of each task are described in Sections 3.3.1 and 3.3.2, the following words about their differences is pertinent to understand the adopted transfer process.

One of the most significant difference from the source to the target environment relies on the ball. This is no small aspect, as although a good job was done when modeling the basketball scenario in LWRSIM, and the simulator is one of the most advanced physics softwares up to now, elements such as the ball, the holder and the air were modeled rather empirically as objects with contact/friction forces and inertia approximated and proposed values for viscosity and density air. Hence, there is no explicit guarantee that the ball dynamics in the real world behaves the same as it does in LWRSIM. Being that PILCO proposes the optimal policy based on the learned GP ball dynamics model, if there is difference on the ball dynamics from one environment to the other, the policy needs to be transfered in such a way that it leverages such difference. On the same statement about modelled objects, it is worth to mention that LWRSIM’s robot was approximated by defining damping and friction coefficients between joints and velocity limits for each axis according to the KLR manual specification.

Another difference between both tasks lies on the dilemma between a software trying to emulate the real world and its cost. In our case, this is mostly noticed on the way that the policy gets executed in the simulator and in the real world. Although it was not exactly measured, it was perceived that the simulator performs slightly faster than the real robot yet when timestep for both is set to the same value. Although not exhaustive research on this was done, a possible explanation might relay in the way that LWRSIM is designed, since it works as an unified system, it reduces the time for the data transfer, compared with the way that KLR works, whose performing relays in the tool chain composed by different subsystems. This is specially important for our feed forward controller, because the actions to follow are generated in real-time (compared to works where the actions are known before-hand and just followed as in LWRSIM tests or [2]). LWRSIM execution time is susceptible to factors inherent to the simulator such as to wether it is rendering or not, or a collision between modelled objects, both of which introduce extra or less computation requirement in the execution time, setting a gap between simulated and real environment.

Last but not least, the determination of the state vector $S_t$ in the real world posed a major challenge since it is the only driving factor for the policy (aside naturally to the policy parameters). While in the source task, the ball position and velocity is exactly determined at every timestep as result of the analytical model computation, its counterpart in the target task does not count with the same accuracy and compliance. Albeit implementation details of the ball tracking in the real world are covered in the upcoming section, it is appropriate to introduce that the major
challenge and difference for TL was to enable a system that delivered $S_t$ with a similar reliance than the one in LWRSIM. To exemplify this statement, we can enounce a typical case from the earlier stages of the implementation in the target environment. Since in the beginning, no filter was used for the tracking ball system, $S_t$ turned to be very noisy and caused a harmful chain reaction. Given this highly changing frequency of $S_t$, the policy delivered a highly changing action, which applied to the KLR resulted in a shaking KLR movement. Due to the forward controller nature of the policy, such shaking robot caused a bouncy ball which leded to a yet more unstable ball position reading, completing a disastrous cycle for the policy application.

3.2.2 The policy adjustment

After stating the major differences between the source and task environment, we can express that although simulation robot can be calibrated to resemble more closely KLR, ambient conditions like air density and temperature, and ball properties such as plasticity, bouncing damping coefficient, friction coefficient between the ball and the holder or weight of the ball might seem innocuous factors, but can pose a huge difference in the real world physics and will definitely play a major role for a policy that completely relies on an element such as a plastic ball.

The adopted TL aims to reproduce in KLR the behaviour displayed in LWRSIM, trying to overcome the stated differences. For this, a policy adjustment that learns how to modify the actions when applied in KLR so they incur the same $S_t$ in both environments, is developed based on [6].

Let us start by refering to the definitions from Table 3.4, where a trajectory refers to the path described by the robot as result of the policy execution. The superscript $(S)$ refers to the outcomes of the policy applied in the source task and $(P)$ refers to their analog in the target task. The TL uses the dataset $D^{(S)}$ generated from the state-action pairs of the trajectory performed in the source task $(S_t^{(S)}, A_t^{(S)})$ to train an adjustment model $\pi_{\text{adj}}$ which modifies the actions from the source policy $\pi(S_t, \theta)$ into actions that generate the source trajectories in the target environment.
Besides of $\pi_{\text{adj}}$, the other major component in our TL is the model $g$. $g$ is the model of the ball inverse dynamics, learned as a regression function $g : S^{(T)} \times S^{(T)} \rightarrow A^{(T)}$ by using the triplet $S_t^{(T)}, A_t^{(T)}, S_{t+1}^{(T)}$ as training dataset. In other words, $g$ learns the function of the pair state-action that caused that particular transition in the dynamics and the adjustment model $\pi_{\text{adj}}$ gets trained by the dataset $D^{(T)}$ formed by $(S_t^{(s)}, A_t^{(s)}, \hat{A}_t^{(T)})$, where $\hat{A}_t^{(T)} = g^{(T)}(S_t^{(s)}, S_{t+1}^{(s)})$. $\pi_{\text{adj}}$ is then able to deliver a modified action vector for the target system that optimizes the same objective as in the source task under the new dynamics learned by $g$.

As functions $\pi_{\text{adj}}$ and $g(.)$ can be stated as SL problems, a linear and a GP regression model were implemented and compared. Due to the lower MSE displayed by the GP regressor, this last approach was adopted to train the TL components $L_2$ (Equation 3.10), to learn the inverse ball dynamic $g(.)$, and $L_3$ (Equation 3.11), to learn the policy adjustment $\pi_{\text{adj}}$.

Figure 3.5 shows the described TL, where

\[
\begin{align*}
A_t^{(s)} &= \pi^{(s)}(S_t^{(s)}), \\
A_t^{(T)} &= \pi_{\text{adj}}(S_t^{(T)}, \hat{A}_t^{(T)}). 
\end{align*}
\]  

(3.8)  

and Algorithm 2 presents its pseudocode, where

- $L_1$ is the PILCO algorithm, and
- $L_2$ and $L_3$ are GP regression algorithms to predict the ball dynamic $g(.)$ and the policy adjustment $\pi_{\text{adj}}$ respectively

\[
\begin{align*}
\hat{A}_t &= L2(S_t, S_{t+1}), \\
A_t^{T} &= L3(S_t, \hat{A}_t) 
\end{align*}
\]  

(3.10)  

(3.11)

for our implementation.
Figure 3.5: Policy adjustment (Source: [6]).

Data:
Source MDP $S : (\mathcal{S}, \mathcal{A}, \mathcal{T}^{(S)}, c)$
Target MDP $T : (\mathcal{S}, \mathcal{A}, \mathcal{T}^{(T)}, c)$
Policy learning algorithm $L_1$
SL algorithm $L_2$
SL algorithm $L_3$
Number of optimizations $N_T$

Result: Policy adjustment model $\pi_{adj}$

Init:
$\pi^{(S)} \leftarrow L_1(\mathcal{S}, \mathcal{A}, \mathcal{T}^{(S)}, c)$
Sample $D^{(S)}$ by applying $\pi^{(S)}$ in $S$
$D^{(T)} \leftarrow \emptyset$
i $\leftarrow 0$
$\pi_{adj} \leftarrow \pi^{(S)}$

While $i < N_T$ do

$D^{(adj)} \leftarrow \emptyset$
Sample $D_i^{(T)}$ by applying $\pi_{adj}$ in $T$
$D^{(T)} \leftarrow D^{(T)} \cup D_i^{(T)}$
g $\leftarrow L_2(D^{(T)})$

For each $(S_t^{(S)}, A_t^{(S)}, S_{t+1}^{(S)}) \in D^{(S)}$ do

$\hat{A}_t^{(T)} \leftarrow g(S_t^{(S)}, S_{t+1}^{(S)})$
$D^{(adj)} \leftarrow D^{(adj)} \cup (S_t^{(S)}, A_t^{(S)}, \hat{A}_t^{(T)})$

End

$\pi_{adj} \leftarrow L_3(D^{(adj)})$

End

Return: $\pi_{adj}$

Algorithm 2: Policy adjustment (Source: [6]).
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_t \xrightarrow{A_t} S_{t+1}$</td>
<td>Dynamics, transition from state $S_t$ to $S_{t+1}$ caused by action $A_t$</td>
</tr>
<tr>
<td>$D^{(S)}$</td>
<td>Dataset generated in the source task as result of policy execution</td>
</tr>
<tr>
<td>$D^{(T)}$</td>
<td>Dataset generated in the target task as result of policy adjustment execution</td>
</tr>
<tr>
<td>$(S_t^{(S)}, A_t^{(S)})$</td>
<td>Trajectory sampling in source task, used to learn $\pi_{adj}$</td>
</tr>
<tr>
<td>$(S_t^{(T)}, A_t^{(T)}, S_{t+1}^{(T)})$</td>
<td>Trajectory sampling in target task, used to learn $g$</td>
</tr>
<tr>
<td>$A_t^{(S)}$</td>
<td>Action from policy in the source task</td>
</tr>
<tr>
<td>$A_t^{(T)}$</td>
<td>Action from policy adjusted for target task</td>
</tr>
<tr>
<td>$g: S^{(T)} \times S^{(T)} \rightarrow A^{(T)}$</td>
<td>Regression model for the inverse dynamic of the ball in the target task. Predicts the action that generated the dynamic</td>
</tr>
<tr>
<td>$\pi_{adj}: S \times A \rightarrow A$</td>
<td>Policy adjustment model</td>
</tr>
<tr>
<td>$L_2 \rightarrow g(.)$</td>
<td>SL algorithm to predict the ball dynamic $g(.)$</td>
</tr>
<tr>
<td>$L_3 \rightarrow \pi_{adj}$</td>
<td>SL algorithm to predict the policy adjustment $\pi_{adj}$</td>
</tr>
</tbody>
</table>

Table 3.4: Summary of TL
3.3 System overview

For easiness of reading, the overall of the system can be divided in the following three stages:

- RL of the basketball skill without previous knowledge as the source task, covered in Section 3.3.1,
- Execution of the learned policy in the physical system, as the non-adjusted policy in target task, covered in Section 3.3.2 and
- TL as the policy adjustment, covered in Section 3.3.3

To enable this system, two major tools were used: the Robot Operating System (ROS)[25] and the toolchain provided by the Open Robot Control Software (OROCOS)[26]. The ROS version is Kinetic (released in May 2016) and for OROCOS, version 2.6 was used. The rest of this chapter handles ROS and OROCOS terminology, therefore, if the reader is not familiar with it, a good starting point can be found in their respective webpages, [25] for ROS and [26] for OROCOS.

The implementation of the complete system required to develop four ROS packages and one OROCOS component, plus the usage of already implemented packages and functionalities. They are presented accordingly in the respective following Sections and summarized in Table 3.7.

3.3.1 The robotic skill in the simulated environment

As introduced in Section 1, a major drawback of RL is the high number of executions that it needs in most cases. To overcome this, we enabled the learning of the basketball skill through the PILCO algorithm in a software simulator. The basketball skill was simulated through the employment of MUJOCO[27], which is up to now, one of the most advanced physics engines. Developed by Roboti LLC, it is a proprietary software suitable for robotics and multi-body dynamics. On its side, LWRSIM[4] started as a simulation environment for the basketball skill in MUJOCO through its simulated KLR as the project for a course, and became a generic simulation environment for diverse robotic skills.

To combine the environment enabled by LWRSIM with PILCO as the policy learning algorithm, two components were used: the MATLAB API for Python[28] and the Robot Operating System (ROS)[25]. As the PILCO algorithm is written as a set of MATLAB scripts, the MATLAB API served as a software wrapper that allowed to execute most of the PILCO functionalities. On the other hand, ROS architecture allowed us to create specific nodes to set an instance of a PILCO execution and its corresponding LWRSIM as recipient for the policy execution.
As presented in Table 2.1, different combination of PILCO parameters endow different PILCO outcomes, therefore, an architecture that allowed the execution of isolated instances with different PILCO parameters settings was obligated. In response to this, the package *learn_policy* was developed.

This package counts with launch files that set the ROS parameters that every node needs. Every launch file executes an instance of the central node *learn_policy*, from the *learn_policy* package, and one instance of the node *lwrsim_server*, which is the central node of the *LWRSIM* package.

*learn_policy* executes the PILCO algorithm and establish communication with *lwrsim_server* as a client to its services allowing a synchronized execution between the simulation and the policy execution. Although more details about such services can be found in [4], it is noticeable that these services provide the state $S_t$ from the simulation and execute the policy computed by *learn_policy*, among other required functionalities used by the client node.

The learned policy gets stored as a MATLAB file that is later used in the execution and policy adjustment phases. After every policy parameters optimization, the optimized policy is executed and the resulting cost function is displayed, as Figure 3.3 previously showed. The architecture of the described subsystem is presented in Figure 3.6 where the ROS nodes and subcomponents are depicted. As a complement to the main functionality and for an easier management of PILCO learning scenarios, two extra nodes were developed in the *learn_policy* package and are presented in Table 3.5.

![Figure 3.6: ROS architecture for policy learning, every thick box represent a node.](image)
Node Description

*learn_policy*  
Node that executes the PILCO algorithm in the lwrsim node.

*learn_policy_weiter*  
Node that continues a previously started PILCO execution to achieve a higher number of optimizations $N_T$.

*play_exec*  
Node that executes a policy learned by *learn_policy* in *lwrsim_server* and displays the execution cost.

Table 3.5: ROS nodes of *learn_policy* package

### 3.3.2 The robotic skill in the real-world environment

To state the real-world environment displayed on Figure 3.7 as the target MDP $T: (S, A, T^{(T)}, c)$, three elements were required:

- a vision system able to provide $S_t$,
- an OROCOS component to perform $A_t$ in the KLR and
- a ROS node to compute the policy and to manage the vision system and the OROCOS component.

![Figure 3.7: Basketball as the robotic skill in the real-world environment.](image)

The architecture of this subsystem, in which the non-adjusted (i.e. unmodified) policy $\pi^{(S)}(S_t, \theta)$ is executed directly in the target environment, is presented in Figure
3.8. ROS nodes and their communications are depicted, whereas their respective explanation is covered in the following paragraphs.

Figure 3.8: ROS architecture for non-adjusted policy execution in KLR, every thick box represent a node.

The vision system

The vision system developed as the package vision_ball is responsible of tracking the ball and provide its 3D Cartesian position and velocity to deliver the $S_i$ that the policy requires at every timestep. The Xbox 360 Kinect camera was used for this purpose given its hardware combination of an RGB camera, an IR camera and an IR projector. Together, they are able to deliver images with voxels (depth pixels) that can create a cloud of points which associate every pixel to a cartesian position. This ability and the ROS support offered by the opensource package freenect_launch, presents the Kinect camera as a quick solution to the aforementioned task.

The node vision_ball, from the package of the same name, takes care of delivering the ball tracking data by using tools from OpenCV for image processing, from freenect_launch for publishing Kinect images and pointclouds as ROS topics, and from ROS message_filters as the time synchronizer for the Kinect topics.

To retrieve the Kinect data, vision_ball subscribes to the ROS topics published by freenect_launch in the registered modality for the raw images and the pointcloud. Information in the topics is not synchronized, therefore, message_filters synchronizes the incoming data of both topics, according to the timestamps contained in their headers, and outputs them in the form of a single object. This object contains the
color image and the cartesian position associated with each pixel at the same time, hence, the color image is filtered to obtain the pixel coordinates of the blue ball center. With this pixel information, its associated cartesian position is retrieved from the cloud of points and the camera-referenced position vector of the ball \( \vec{P}_C \) is obtained.

As we require \( S_t \) referenced in the same coordinate system (CS) than LWRSIM (robot CS), a transformation between the camera and the scene is required to translate the ball-position camera-referenced \( \vec{P}_C \) to a ball-position robot-referenced \( \vec{P}_R \), as displayed in Figure 3.9. This transformation can be approximated by a rigid body motion (referred as Direct Extrinsic Parameter Calibration in [29]), which can be modeled as a coarse rotation matrix and a translation vector between the CSs 1 and 2.

\[
\vec{P}^2 = {^1}_M {^2} \vec{P}^1 \tag{3.12}
\]

A rigid body motion from the CS 1 to 2 is determined by

\[
{^C}_M {^R} = \begin{pmatrix}
0 & R_{RC} & 0 & R_{tC} \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix} \tag{3.13}
\]

with the transformation matrix \( ^1 M_2 \) containing the rotation matrix \( R \) and the translation vector \( t \). Granted that our case required the transformation from the camera \( C \) to the robot CS \( R \), the transformation matrix \( ^C M_R \) results in Equation 3.13,

where the rotation matrix \( R \) from robot to camera CS

\[
R_{RC} = [R_{x_C}, R_{y_C}, R_{z_C}]'
\]
is result of the successive rotations, around the x-axis (Eq. 3.14), y-axis (Eq. 3.15) and/or z-axis (Eq. 3.16), necessary to align the robot CS to the camera CS. On its side, $t_{RC}$ is the translation vector which represents the displacement between the CSs with origin in the robot CS, and pointing to the origin of the camera CS.

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$ (3.14)

$$R_y = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix}$$ (3.15)

$$R_z = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$ (3.16)

With this said, Equation 3.12 turns into Equation 3.17, where $\vec{P}_C = [C_{Px}, C_{Py}, C_{Pz}]'$ is the point vector camera-referenced to be transformed, and $\vec{P}_R = [R_{Px}, R_{Py}, R_{Pz}]'$ is the same point vector, now robot-referenced. For understanding purposes, Equation 3.17 is displayed in an extended version in Equation 3.18.

$$\vec{P}_R = C_M R^{\vec{P}_C}$$ (3.17)

$$\vec{P}_R = \begin{pmatrix} 0 & R_{RC} & t_{RC} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} C_{Px} \\ C_{Py} \\ C_{Pz} \end{pmatrix} = \begin{pmatrix} R_{Px} \\ R_{Py} \\ R_{Pz} \end{pmatrix}$$ (3.18)

After the ball position and velocity are estimated, _vision_ball_ publishes this data along extra information such as the timestamp, previous data and time difference as messages in a ROS topic that are read by node _play_policy_on_real_kuka_ as previously shown in Figure 3.8.

**The OROCOS component**

The OROCOS project is a set of libraries for advanced machine and robot control[26], which offers a middleware to enable a toolchain that empowers the development of real-time software. Although an impeccable explanation of OROCOS terminology can be found in [30], we suffice by briefly explaining what a component and a deployer are, facilitating the introduction of the developed parts for this project.

A deployer is a file that dynamically creates a component. A component is a software class used by the deployer, who creates and configures an instance of the component during execution time. So, abusing the notation, we call component to the
instance of a class that gets created due to the deployer’s call. As Figure 3.10 shows, a component is a modular entity that in its simplest form, gets configured in the PreOperational phase (through the deployer), executes a callback, an algorithm or repeatedly the function updateHook() and allows a data flow through the in/output ports.

![Schematic of an OROCOS component](image)

Figure 3.10: Schematic of an OROCOS component (a.k.a. Task in OROCOS terminology) (Source: [30]).

Figure 3.11 shows the current infrastructure that can be found in our laboratory, and that through the OROCOS toolchain, allows a real-time connection between the KLR and the components FRIServer and KUKACommander. While KUKACommander establishes communication with FRIServer, FRIServer provides an interface to the Fast Research Interface (FRI) allowing us to send commands to the Kuka Robot Controller (KRC) which get reflected in KLR. Even though full details about this implementation can be found in [31], we can succinctly say that it consists of three pieces of hardwares: the external computer, the KRC and the KLR.

Under these circumstances, we developed a component and deployer that allowed us to interact with the KLR similarly to how we did it with LWRSIM in Section 3.3.1.

For this reason, among other internal functionalities, oroc_execute_on_kuka component was developed to control the KRC with a joint position control and execute the commanded positions from the ROS node play_policy_on_real_kuka in KLR through KUKACommander and FRIServer, whereas deployer_execute_on_kuka was the deployer file used to configure the components, their frequency and more importantly, the communications between the ROS nodes and oroc_execute_on_kuka via ROS topics as displayed in Figure 3.8.

The policy executor

Homogeneity reasons encouraged us to develop ROS nodes that allowed us to interact with KLR in a similar fashion to how we interacted with LWRSIM in Figure 3.6
with the `learn_policy` node (Section 3.3.1). In addition to this, an almost seamless transition was of interest for the future case in which the policy had to be executed either in the source or in the task environment (to be discussed in Section 3.3.3). Under these guidelines, the nodes `rbf_server` and `play_policy_on_real_kuka` were developed.

Briefly introduced by Figure 3.8, the node `rbf_server` executes a policy server, which due to the feed forward nature of the RBF policy $\pi(S_t, \theta)$, offers the service `get_action_from_policy_pos_vel` to guarantee synchrony between $S_t$ and $A_t$. Similarly to `learn_policy`, `rbf_server` uses the MATLAB API for Python to embody an RBF policy function previously learned with PILCO and `learn_policy`. The policy is retrieved from a MATLAB file and specified as a ROS parameter at the moment of the node creation, from where $\theta$ is obtained to enact $\pi(S_t, \theta)$. Consequently, `rbf_server` takes service requests with $S_t$ and respond back with the $A_t$ computed by $\pi(S_t, \theta)$. So, in a nutshell, `rbf_server` is exactly that: a server that enables the RBF policy function $\pi(S_t, \theta)$, receiving $S_t$ as service requests and replying with the corresponding $A_t$ as service response.

To complete the lineup shown in Figure 3.8, the launch file responsible of spawning each one of the nodes, instantiates the last and most important one: `play_policy_on_real_kuka`. It acts as the orchestra director because it mediates and controls the communications between `rbf_server`, `vision_ball` and `oroc_execute_on_kuka`. 

---

Figure 3.11: System overview of the three hardware systems: the external computer, KRC and the arm KLR (Source: [31]).
To describe its functionality in very simple words, `play_policy_on_real_kuka` gets subscribed to the ROS topic in which `vision_ball` publishes $S_t$. After a filtering and stabilizing procedure, `play_policy_on_real_kuka` request the $A_t$ corresponding to $S_t$ from `rbf_server` at the frequency $f = 1/d_t$ specified by the policy. Once $A_t$ is obtained, it establishes communication with `oroc_execute_on_kuka` through ROS topics in order to enact in KLR the action recommended by the policy, after which, a new $S_t$ is retrieved and the cycle starts all over again until the episode ends at time $T$.

### 3.3.3 TL of RL for the real robot

As announced in the beginning of this Section 3.3, implementing TL of RL, as a policy adjustment, is the last stage of the project. In the seek of its full understanding, this subsection recalls elements from the two previous stages and introduces the new ones to fulfill the major objective: to transfer the policy, learned by PILCO, from the source system $S$ (LWRSIM as simulation environment) to the target system $T$ (KLR as real world environment) through a policy adjustment.

After presenting the TL proposal in Section 3.2 and introduced the nodes `lwr-sim_server`, `rbf_server`, `play_policy_on_real_kuka`, `vision_ball` and the `oroc_execute_on_kuka` component in Sections 3.3.1 and 3.3.2, the TL scenario depicted in Figure 3.12 should appear quite understandable. Except for the `adjust_policy` and `play_policy_on_mujoco` nodes, this new scenario is very similar to the combination of those presented by Figure 3.6 and 3.8, maintaining its functionalities and communication methods.

Figure 3.12 is the implementation of the model displayed in Figure 3.5 and `adjust_policy` contains most of the steps to fulfil the Algorithm 2, whereas the full Algorithm is enacted by the participation of all the nodes.

To start unveiling the unknown components, let us start with the new core element: the node `adjust_policy`. Its main function is to act as the mediator between `rbf_server` and the respective `play_policy_on_X` by deciding on which system $S$ or $T$ a policy gets executed.

Although maybe obvious, it is pertinent to observe that similarly to what `play_policy_on_real_kuka` in Figure 3.8 accomplished, `play_policy_on_mujoco` does exactly the same for its analog in the $S$, i.e. executes the service-requested policy $\pi$ on `lwr_sim_server`. Then, to relate these nodes to Algorithm 2, we can state that

- $play_policy_on_mujoco = \text{Sample } D^{(S)} \text{ by applying } \pi^{(S)} \text{ in } S$
- $play_policy_on_real_kuka = \text{Sample } D^{(T)}_t \text{ by applying } \pi_{adj} \text{ in } T$
Apart from this mediating function, `adjust_policy` records and accumulates the trajectories $D^{(adj)}$ and $D^{(T)}$, turns them into training datasets suitable for $L_2$ and $L_3$, stores the regression models $g(.)$ and $\pi_{adj}$ and plots the outcomes. Except by the plotting functionality, all the functions in `adjust_policy`, fall under the guidance stated in Algorithm 2.

ROS nodes to implement and adjust the policy in the target environment are summarized in Table 3.6. Finally, the complete ROS packages and OROCOS components are briefed in Table 3.7.
<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vision_ball</td>
<td>Node of <em>vision_ball</em> package that publishes ball position and velocity in ros topic.</td>
</tr>
<tr>
<td>rbf_server</td>
<td>Node of <em>rbf_executor</em> package that creates a server for an RBF policy specified through a ROS parameter.</td>
</tr>
<tr>
<td>play_policy_on_mujoco</td>
<td>Node of <em>rbf_executor</em> package that request the service from <em>rbf_server</em> to execute the policy in the <em>lwrsim_server</em> node.</td>
</tr>
<tr>
<td>play_policy_on_real_kuka</td>
<td>Node of <em>rbf_executor</em> package that request the service from <em>rbf_server</em> to execute the policy in the <em>oroc_execute_on_kuka</em> component.</td>
</tr>
<tr>
<td>adjust_policy</td>
<td>Node of <em>adjust_policy</em> package that reads the executions from <em>play_policy_on_real_kuka</em> and <em>play_exec</em> to perform the TL.</td>
</tr>
</tbody>
</table>

Table 3.6: ROS nodes of *vision_ball*, *rbf_executor* and *adjust_policy* packages.
<table>
<thead>
<tr>
<th>Package / Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>learn_policy</td>
<td>ROS package developed to handle the policy learning functionalities.</td>
</tr>
<tr>
<td>rbf_executor</td>
<td>ROS package developed to handle the policy execution functionalities.</td>
</tr>
<tr>
<td>vision_ball</td>
<td>ROS package developed to handle the vision system for the ball tracking.</td>
</tr>
<tr>
<td>adjust_policy</td>
<td>ROS package developed to handle the TL.</td>
</tr>
<tr>
<td>lwrsim</td>
<td>ROS package provided by LWRSIM software.</td>
</tr>
<tr>
<td>oroc_execute_on_kuka</td>
<td>OROCOS component developed to execute the commanded positions in KLR through <em>KUKACommander</em> and <em>FRIServer</em> .</td>
</tr>
<tr>
<td>KUKA Commander</td>
<td>OROCOS component provided to establish communication with <em>FRIServer</em>.</td>
</tr>
<tr>
<td>FRIServer</td>
<td>OROCOS component provided to establish communication with FRI.</td>
</tr>
<tr>
<td>deployer_execute_on_kuka</td>
<td>Deployer file for <em>oroc_execute_on_kuka</em> execution. Sets the communications between ROS nodes and OROCOS component via ROS topics.</td>
</tr>
</tbody>
</table>

Table 3.7: Summary of ROS packages and OROCOS component
Chapter 4

Experiments and results

Practice is not the thing you do once you are good. It is the thing you do that makes you good.

Malcolm Gladwell

This chapter presents the performed experiments and obtained results of the project. Section 4.1 initiates by presenting the testbed and the outcomes of the first stage, in which PILCO plays RL in the source task LWRSIM. Consequently, Section 4.2 displays the experiments and results of applying TL of the policy learned in the previous stage to the real KLR with a real plastic ball (brief videos of these results can be found in [32], [33] and [34]). To conclude, a pertinent discussion is present in Section 4.3.

4.1 RL in the source task

Given that PILCO is a model-based algorithm originally implemented in MATLAB for simple scenarios, we first tested our pythonised version with a test benchmark and then with LWRSIM. As result of this section, two policies $\pi(S_t, \theta)$ to perform the skill were selected to be adjusted with TL in the next Section 4.2.

The results shown were obtained in a computer with the following characteristics:

- CPU: Intel Xeon E5-2630 v4 @ 2.20 GHz x 20
- GPU: GeForce GTX 1080 Ti/PCIe/SSE2
- Architecture: 64-bit Computing
- RAM Size: 64GB
- Technology: DDR3 SDRAM
- Operating System: Ubuntu 16.04.4 LTS, 64 bits
Preliminary experiments with PILCO

As an early stage, we tested PILCO on the cartpole swing-up problem as a benchmark experiment for controllers of nonlinear dynamical systems, aiming to solve the following questions: What are the PILCO challenges when learning in the complexer scenarios provided by MUJOCO compared to the Matlab ODE scenario?, how difficult is it for PILCO to learn the dynamics model depending on the scenario settings?, given that PILCO is a model-based algorithm, what is the PILCO’s learning ability when the scenario is under/over sampled?, what happens when the training set for the model creation is diminished?, since PILCO uses inversion of matrices in the training GP model phase, how is the computing time affected as the training dataset increases after every episode execution?

The cartpole swing-up problem consists of a pole attached to a cart as a pendulum. Each episode begins with the pole hanging downwards and the goal is to move the cart in such a way that the pendulum gets balanced upright and holds this position. Similarly to the basketball skill, PILCO was requested to learn this policy as a linear combination of 100 RBFs to generate the horizontal force exerted on the cart in the range of [-10, 10]N to fulfill the task. To perform a policy optimization, PILCO requires a cubic computing time $O(n^3)$ to obtain the inverse kernel matrix for the covariance and mean of the GP model.

PILCO’s performance was evaluated in the ODE Matlab test scenarios (displayed in Figure 4.1) against the dynamic model enabled by MUJOCO (displayed in Figure 4.2) which claims to be “not merely a better simulator (…) it is the first full-featured simulator designed from the ground up for the purpose of model-based optimization”[27].

![Figure 4.1: PILCO learning in MATLAB environment.](image)

We compared the number of optimizations that PILCO required to learn the policy with two variables: across the environment timestep $d_t$ and across the size of
the training dataset used to learn the model. As a brief reminder, \(d_t\) and \(d_{\text{training}}\) (Table 2.1) were introduced as an aim to reduce the GP model training data set. The size of the training data set depends on the relation between \(d_t\) and \(d_{\text{training}}\), where \(d_t = d_{\text{training}}\) implies to learn using all the data retrieved from the whole episode, and \(d_{\text{training}} > d_t\) means to subsample the training dataset at the \(d_{\text{training}}\) rate from the complete episode data.

<table>
<thead>
<tr>
<th>Settings</th>
<th>Optimizations (N_{\text{opt}}) required in MATLAB simulation</th>
<th>Optimizations required in MUJOCO simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d_t = 0.1) s, (d_{\text{training}} = 0.1) s</td>
<td>7</td>
<td>38</td>
</tr>
<tr>
<td>(d_t = 0.1) s, (d_{\text{training}} = 0.2) s</td>
<td>13</td>
<td>61</td>
</tr>
<tr>
<td>(d_t = 0.01) s, (d_{\text{training}} = 0.01) s</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>(d_t = 0.01) s, (d_{\text{training}} = 0.02) s</td>
<td>18</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 4.1: Results of different PILCO settings in different simulation environment for the cartpole task

While cost functions shown in Figure 4.1 and 4.2 reveal that PILCO learned successfully the policy and achieve the task in both scenarios, the results of Table 4.1 show that there is a big difference when learning in ideal conditions (Matlab scenario) compared to the more realistic layout (MUJOCO scenario). It can be seen that in general from the Matlab to the MUJOCO scenario, it needs 5 or 6x more optimizations \(N_{\text{opt}}\) to learn the same policy under the same \(d_t\) and \(d_{\text{training}}\) settings.
Due to PILCO’s nature, after every episode the whole episode data gets accumulated to increase the training dataset for the next optimization. Therefore the jump from 7 to 38 (or 42) optimizations displayed in the first and third experiment becomes relevant when considering the overall computation time.

Another important observation lies in the timestep for the first and third experiment. It can be seen that when PILCO takes the full training dataset \( d_t = d_{\text{training}} \), the choice of \( d_t \) is not a major player for the learning phase since it learned with the similar \( N_{\text{opt}} \) for each scenario. This importance is opposed to an analysis of the controller execution, where the choice of \( d_t \) plays a critical role: if \( d_t \) decreases, it becomes difficult to observe the effect of the action in the environment, and if \( d_t \) increases, the scenario gets unreliable due to the undersampling.

On the other hand, despite the perks of a smaller training data set (shorter matrix inversion time), the second and fourth experiment let us observe that \( d_{\text{training}} \) has a great impact depending on the scenarios. For the ideal Matlab case, the required optimizations increased mildly (from 7 to 13, and from 7 to 18), whereas for the MUJOCO scenario, to count with less observations meant almost the same scale of extra optimizations (2x), which gets reflected into a much bigger difference (from 38 to 61, and from 42 to 65). However, an important remark is that in this case, every one of these optimization takes shorter time than when taking full training dataset.

Our experiments demonstrated that a subsampled training dataset \( d_{\text{training}} > d_t \) is faster in computing single optimizations than a full training dataset \( d_{\text{training}} = d_t \) as result of a smaller kernel matrix, but in overall a \( d_{\text{training}} \) twice \( d_t \) required almost three times the number of optimizations needed to to learn with the full training dataset for the combination in the rest of the parameters.

These experiments let us infer that a higher number of optimization \( N_{\text{opt}} \) is required when moving from an ideal to a more realistic scenario for the same task. Similarly, to count with a smaller training dataset requires a higher \( N_{\text{opt}} \). However, the computing time for a smaller training dataset is exponentially decreased \( (O(n^3)) \) which allows to perform more optimizations with a poorer dynamics-model. Since this trade-off between training dataset and number of optimizations reached optimal policies for the cartpole task, we extended this proposal to the PILCO learning with LWRSIM.

### 4.1.1 Experiments with PILCO and LWRSIM

As most of the general PILCO research questions were covered in the preliminary experiments with the cartpole, the experiments described here were conducted aiming to understand PILCO challenges for the basketball skill. In particular, what is the right combination of parameters that allow PILCO to learn the skill in LWRSIM? To pursue this answer, different scenarios were proposed, testing PILCO learning
dexterity. The most important details about these scenarios laid in how the skill was modeled, as well as the definition of the cost $c(S_t)$ and policy $\pi(S_t, \theta)$ functions presented in Section 3.1.

On the other hand, alternative questions of interest were related to the way in which PILCO learns an optimal policy: since PILCO learns from a tabula rasa state, how does it progressively learns the basketball skill?, how does PILCO change the proposed policy towards a lower cost?, does it necessarily mean an optimal policy?, does it get stucked in local optimals? Aiming to response these questions, the results shown in Appendix A summarize some effects of the parameters combination and present them as cases of interest.

In the experiments, PILCO was tested across the parameters: (reference for the notation can be found in Table 2.1)

- Environment timestep $d_t$ [s]
- Training timestep $d_{\text{training}}$ [s]
- Episode time $T$ [s]
- Number of optimizations $N_{\text{opt}}$
- Maximum U $u_{\text{max}}$ [deg/s]
- Optimization length
- Optimization MFEPLS
- Exploration parameter $k$

A home position was defined as Figure 4.3 shows; however, every learning episode started its execution from a slightly different position. Specifically, a random position in the 90% vicinity of the fixed home position was arbitrarily chosen, and afterwards, the policy started. The target position for $S_t$ was the basket location $P_{\text{target}} = [-0.768, -0.816, 0.280]$ i.e. 1.50 m $\angle 57^\circ$, referenced to the robot CS presented in Figure 3.9.

Although cartpole experiments provided insights on the algorithm particularities, PILCO required to be tested across different specifications for the larger and complexer scenario depicted by the basketball skill. The tested variables included the number of optimizations $N_{\text{opt}}$, the size of the training data to generate the GP model of the ball (influenced by the total episode time $T$, the environment timestep $d_t$ and the training timestep $d_{\text{training}}$), the optimization parameters for learning the policy (optimization length and MFEPLS), and parameters related to the cost and policy function themselves: the exploration parameter ($k$) and the maximum U ($u_{\text{max}}$) allowed, as expressed in Sections 2.1.5 and 4.1.
4.1.2 Results

Table 4.2 aims to start answering the initial research question by showing the parameters for learning the policies to be transferred.

<table>
<thead>
<tr>
<th></th>
<th>lowvel1</th>
<th>lowvel2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td>$N_{\text{opt}}$: 44</td>
<td>$N_{\text{opt}}$: 43</td>
</tr>
<tr>
<td><strong>Single optimization</strong></td>
<td>$T$ [s]: 4  $d_t$ [s]: 0.1 $d_{\text{training}}$ [s]: 0.2</td>
<td>$T$ [s]: 4  $d_t$ [s]: 0.1 $d_{\text{training}}$ [s]: 0.1</td>
</tr>
<tr>
<td><strong>Optimization parameters</strong></td>
<td>O-length: 40 O-MFEPLS: 8</td>
<td>O-length: 60 O-MFEPLS: 15</td>
</tr>
<tr>
<td><strong>RL</strong></td>
<td>$k$: -0.8 $u_{\text{max}}$ [deg/s]: $[50,50,50]$</td>
<td>$k$: -0.8 $u_{\text{max}}$ [deg/s]: $[30,30,30]$</td>
</tr>
</tbody>
</table>

Table 4.2: Parameters used to learn lowvel1 and lowvel2 policies.
While most of the remaining research questions were covered by the cases presented in Appendix A, the major question related to the right combination of parameters that allowed PILCO to learn the skill was difficult to fully answer. From the performed experiments, only the combination that enabled the case *throwing ball into the air* (Section A.4) was the only one that learned the basketball skill. However, as the ultimate goal of this work is to reproduce the policy in the real-world environment, this policy was not selected to be transferred due to the torque exceedings for the joint J2 that were not a problem in LWRSIM, but triggered failures in the real robot.

Due to this, two policies were selected to test the TL process and are stylized as *lowvel1* and *lowvel2*. Although none of them achieved an impeccable basketball skill, they were selected as a trade-off between its implementing factibility (avoidance of high torque alarms) and the skill fulfilment expressed in a low execution cost. Both policies follow a similar strategy of approaching the ball towards the basket in a slow movement and then throw it or let it fall. The performance of both policies can be found in Appendix B, where the actions $A_t$, the joint positions as result of $A_t$, the state $S_t$ and the execution cost are displayed. Figure 4.4 displays a sequence of such policy, which will be used in the next TL stage.

Figure 4.4: Policy learned to approach ball to basket, a.k.a. *lowvel1* policy.
4.2 TL for the target task

The results shown in this section were obtained in a robotic arm, similar to the one displayed in Figure 4.5, with the following characteristics and interfaces

- Robotic Arm: Kuka Lightweight Robot LWR4+ (KLR)
- Teach Pendant: Kuka Robot Controller (KRC 2lr)
- Controller: KUKA Control Panel (KCP)
- Sample rate: 100 Hz
- Fast Research Interface (FRI)
- Robot Operating System, Kinetic (ROS)
- Open Robot Control Software (OROCOS)

A succinct video of the results can be found in [32] and [33] for the policies lowvel1 and lowvel2 respectively with the basket placed in the target place when learning. For a visual evaluation of the transferability between the source and target tasks, [34] shows the execution of policy lowvel1 when the basket is in a closer location.

Figure 4.5: KUKA Lightweight Robot (KLR), Kuka Robot Controller (KRC 2lr) and KUKA Control Panel (KCP).

4.2.1 Experiments adjusting the policy in the target task

The experiments shown in this section serve to answer questions related to the TL process and the implications of the policies execution in the real-world environment as the target task. In that direction, the following questions are regarded: what is the effect of a policy adjustment compared to a non-adjusted policy?, how many iterations of the adjustment are necessary to achieve a successful transfer?, as the
policy adjustment is based on the $L_2$ and $L_3$ SL models, what is their prediction error?

To conduct the experiments that helped solving such questions, the elements introduced in 3.3.2 were utilized and a small sample is illustrated as follows. Figure 4.6 shows a screenshot of the vision system (responsible of providing $S_t$), where the image on the left is the raw image captured by the Kinect camera, and the right one is the processed image that tracks the objects of interest. While on the other hand, similarly to LWRSIM experiments, Figure 4.7 displays KLR in its home position.

![Figure 4.6: Outcome from vision system, filtering the ball and the rim of the target basket.](image1)

![Figure 4.7: KLR in home position.](image2)

As mentioned in Section 3.2 and Figure 3.5, our TL consists of a policy adjustment that transforms the policy learned in the source task ($A_t^S$) into a policy for the target system ($A_t^T$) at every timestep according to Equations 3.8 and 3.9.
Aiming to answer some of the research questions, the experiments were conducted for the policies \textit{lowvel1} and \textit{lowvel2} in five executions on KLR. The first execution used the non-adjusted policy $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$, i.e., the unmodified policy directly as learned in LWRSIM, and the following four executions consisted of adjusted policies, as iterative and subsequent iterations $N_T$ for models $L2$ and $L3$ according to the algorithm 2.

4.2.2 Results

In order to summarize and ease an overall analysis, the plots presented in this section compare the results of the five executions for both policies, however, for a closer detail, and similarly to LWRSIM experiments, an interested reader can find the individual results in Appendices C and D.

To address the research concerns, four variables of interest are plotted: the action $A_t$ for every joint ($J2$, $J4$, $J6$), the KLR joint position as result of $A_t$, the state $S_t$ as the ball dynamics, and the execution cost. Whereas, to analyze the $L2$ and $L3$ models, an error chart and plots of the adjustment of is provided for every iteration and its error is reviewed in with $a$.

In the TL terminology, it is considered that the transferability of a state is of high value if the experience displayed by the target task agrees to the source task, and represents low value when there exists a discrepancy between them. In this direction, the results obtained in the source task (LWRSIM) were compared with each of the five executions in the target task (KLR).

These comparisons are analyzed for each \textit{lowvel} policy. Figures 4.8 and 4.12 compare the action $A_t$ from the policies on the source task $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, on the target task with non-adjusted policy $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$ and on the target task with adjusted policy across every optimization $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$ for each Joint. It can be seen that the actions on the source task display a smooth function. On its side, although all the actions start with a similar, yet not necessary equal, initial value, the actions in the target tasks, do not further follow the behaviour displayed by the source task and further yet, they change abruptly; hence, actions display low value transferability. Whereas among the target tasks, the actions expose a similar pattern.

From these actions, and according to Equation 3.4, positions are commanded to either LWRSIM (for source task) or KLR (for target task). This is displayed in Figures 4.9 and 4.13 respectively for each policy, as the comparison of the joint positions in the source task and target task as result of $A_t$ for every case. These observations let us notice that, despite the low value transferability of the actions, a more similar behaviour is found in the trajectory comparison of the joints position (good value transferability).
To take a look to the transferability of the ball position, [34] displays the case when the ball reaches the same position in the source and the target task. In the video, the basket is placed in a closer location to the robot, so the ball falls inside, aiding to visually compare the place where the robot falls. In the source task, the ball falls clearly inside the basket. However, when the TL is performed, the non-adjusted policy does not succeed in scoring the ball into the basket. After the policy adjustment, the balls falls inside the basket (similarly to the source task) after the first iteration. Further iterations let us see that the balls keeps falling inside the basket, although the velocity differs as the ball bouncing reflects. This let us infer that, albeit our TL can help by bridging the gap between the tasks, it may become unstable in further iterations.

To complement this transferability analysis, it is necessary to look closer into the ball behaviour. Similarly to the actions and joints comparison, Figures 4.10 and 4.14 provide a visual comparison of the states $S_t$ in each task, which is intimately related to Figures 4.11 and 4.15 where the execution cost is compared in similar fashion. From these visualizations, the most interesting aspect is that the ball seems to fall quicker in the target tasks. Another observation falls in the mismatch between the ball positions, which could obey to the coarse pose estimation provided by the vision system. However, when performing the executions ([32] and [33]) in the target task, these discrepancies did not seem to affect the policy performing in overall.
Policy: lowvel1

Figure 4.8: Comparison of action from policy lowvel1 on source task $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, target task with non-adjusted policy $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$ and target task with adjusted policy across every iteration $A_t^{(T)} = \pi_{\text{adj}}(S_t^{(T)}; \pi^{(S)}(S_t^{(T)}))$ for each Joint.
Figure 4.9: Comparison of position reached on source task and target task as result of $A_t$ in every case, for each Joint.
Figure 4.10: Comparison of state on source task $S_t^{(S)}$ and target task $S_t^{(T)}$ as result of $A_t$ in every case.
Figure 4.11: Comparison of execution cost on source task and target task as result of $A_t$ in every case.
Policy: lowvel2

Figure 4.12: Comparison of action from policy lowvel1 on source task $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, target task with non-adjusted policy $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$ and target task with adjusted policy across every iteration $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$ for each Joint.
Figure 4.13: Comparison of position reached on source task and target task as result of $A_t$ in every case, for each Joint.
Figure 4.14: Comparison of state on source task $S_t^{(S)}$ and target task $S_t^{(T)}$ as result of $A_t$ in every case.
Figure 4.15: Comparison of execution cost on source task and target task as result of $A_t$ in every case.
L2 and L3 models

The last research question of this TL, refers to the error of the models used to adjust the policy. As $L_2$ and $L_3$ are GP regression functions and SL algorithms, they were trained by using 80% of the data as training input and 20% as test set to evaluate the generalization error. Hence, the models learning were measured by computing the Minimum Squared Error (MSE) of the test set at every iteration of each progressive policy adjustment.

Table 4.3 contains the MSE for every joint of both models. It can be seen although both MSE are very small, the error on the inverse-dynamics model $L_2$ increases across the iterations, and the error of the adjusting-policy model $L_3$ tend to decrease. The increasing error on $L_2$ could be tied to the mismatch of the ball behaviour and position tracking refered in the previous section. On the other hand, although arguable and not conclusive, the $L_3$ error could be explained by the fact that no significant adjustment was performed on the policies across the iterations (as shown in previous section), and therefore the learning of such adjustment has very low variance.

To complement these results, Appendix D contains plots of the non-adjusted and adjusted action for both policies at every iteration. For policy lowvel1, test set prediction $L_2 \rightarrow g(\cdot)$ on every iteration is displayed in Figures D.11, D.18, D.115 and D.122; test set prediction $L_3 \rightarrow \pi_{adj}$ is shown in Figures D.12, D.19, D.116 and D.123; and a comparison of action before and after adjustment can be found in Figures D.13, D.110, D.117 and D.124. Same analysis is developed for policy lowvel2 with, test set prediction $L_2 \rightarrow g(\cdot)$ in Figures D.21, D.28, D.215 and D.222; test set prediction $L_3 \rightarrow \pi_{adj}$ in Figures D.22, D.29, D.216 and D.223; and actions comparison in Figures D.23, D.210, D.217 and D.224.
Table 4.3: MSE of test dataset from inverse-dynamics model $L_2$ and from adjusting-policy model $L_3$ across every iteration.

### MSE of inverse-dynamics model $L_2$

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>lowvel1 policy</th>
<th>J2</th>
<th>J4</th>
<th>J6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.06893561e-04</td>
<td>5.74653775e-05</td>
<td>2.83142035e-04</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.38103624e-04</td>
<td>6.49165678e-05</td>
<td>2.48401154e-04</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.33352658</td>
<td>0.23539029</td>
<td>0.21259497</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.35027128</td>
<td>0.23424324</td>
<td>0.25088131</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>lowvel2 policy</th>
<th>J2</th>
<th>J4</th>
<th>J6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.62702986e-05</td>
<td>1.18565364e-03</td>
<td>9.2516812e-03</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.00013688</td>
<td>0.00829667</td>
<td>0.04605272</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.00054968</td>
<td>0.01173476</td>
<td>0.01151857</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.00302236</td>
<td>0.01381966</td>
<td>0.01422727</td>
<td></td>
</tr>
</tbody>
</table>

### MSE of adjusting-policy model $L_3$

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>lowvel1 policy</th>
<th>J2</th>
<th>J4</th>
<th>J6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.69565048e-31</td>
<td>2.63928027e-31</td>
<td>1.71364088e-31</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.90067769e-17</td>
<td>2.77067981e-17</td>
<td>2.29847370e-17</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>lowvel2 policy</th>
<th>J2</th>
<th>J4</th>
<th>J6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.30831615e-10</td>
<td>9.52442312e-08</td>
<td>1.02811023e-07</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.68636607e-11</td>
<td>2.42507279e-08</td>
<td>2.69422971e-08</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.</td>
<td>0.</td>
<td>0.</td>
<td></td>
</tr>
</tbody>
</table>
4.3 Discussion

Albeit learning the skill with PILCO in LWRSIM was a large time investment, its importance laid in being the milestone on which TL would act. The cartpole experiments provided valuable insight on the algorithm particularities and the introduction of the sampling rates $d_t$ and $d_{\text{training}}$ as algorithm parameters was fundamental. Their consideration aimed to reduce the GP model training set, whose downside was the long computing time needed to obtain the inverse kernel matrix for the covariance and mean of the GP model. Hence, these experiments were profitable as they served as hints to understand the effort that PILCO requires to learn an environment complexer than a MATLAB ODE system (test provided by the original package).

When moving from the cartpole task to the basketball skill, PILCO proved to be demanding when it came about details and specifications. One critical aspect of the algorithm fell in the number of optimizations $N_{\text{opt}}$, and although PILCO is a data-efficient algorithm, we believe that due to the scenario of the basketball skill, it usually required large number of optimization trials (between 50 and 250 $N_{\text{opt}}$) to glimpse if the learning was progressing or not depending on the other parameters composition. This was a common situation, particularly when the cost function was being shaped. In that development, before foreseeing any problem with the cost function definition, we waited for long times until PILCO reached a number of optimizations in which we could foreseen if the function looked good or not.

The previous problem gets linked to its child one: the time required for a single optimization. This time was affected mainly by two big factors: the size of the training data to generate the GP model of the ball dynamics (influenced by the total episode time $T$, the environment timestep $d_t$ and the training timestep $d_{\text{training}}$) and the optimization parameters for learning the policy (optimization length and MFEPLS). As shown by lowvel1, the reduction of the training data for the ball dynamics and the reduction of the optimization parameters allowed to perform a faster optimization and hence, more chances were taken with the slightly higher $u_{\text{max}}$ (compared to lowvel2) where a bigger search space for the actions was provided. On its side, lowvel2 required a more refined optimization given its more populated training dataset.

Under these observations, we can state that these experiments are valid for a task in which a model-based RL algorithm is of interest for a complex scenario. Our implementation particularly is of interest if comparing PILCO and the model learning in relation with the size of the training dataset for a task modeled as a MUJOCO scenario. While on the other hand, the policy search performed by PILCO might need more attention to each scenario particularities as our basketball task required the tailored cost function presented in previous Section.

From the variety of policies learned (summarized as cases in appendix A), two were of special interest: the throwtoair and lowvel policies. In throwtoair, the goal
was achieved flawless and quickly. Its characteristic were the high velocities set as \( \text{max}_U \), which allowed to move the robot so fast that it could give the ball enough momentum for it to fly the necessary distance and fall into the basket. Although these movements were allowed in the simulator, in the real robot such high velocities were either impossible to achieve, or triggered torque errors for the lowest joint J2. Hence, being that it was the only policy in achieving the goal, it was not useful in practice.

On its side, lowvel policies were the counterpart to this effect. As the \( \text{u}_{\text{max}} \) was set to a smaller number, PILCO learned to move the robot forward in low motion keeping the ball holder in horizontal position, and once reached a close position to the basket, the ball would get thrown towards the basket. Due to its low velocity, the ball does not get the enough impulse to fly and just falls from the holder.

The findings related to the ball dynamics and the cost, is of highest interest. Linking this to the analysis of the joints behaviour, one conclusion would drive us to consider that the transferability of the ball dynamics was very low, because albeit a similar joints behaviour between tasks, the ball falls in a different fashion (sometimes falling to a slightly different direction in XY frame) and more importantly, the ball falls quicker in the target task. Another conclusion, is that the target task delivered less momentum of inertia to the ball and therefore, it did not traverse more distance in the XY quadrant, falling almost directly in the Z axis direction. Finally, a last mismatch between the ball in both tasks relay in the bouncing after the fall, since the target task ball never bounced the same way that it did in the source task.

The transferability process as result of an iterative optimization did not seem to show any particular improvement respect to the results displayed by the non-adjusted policy. This is unexpected and might be the result of the adjustmen policy statement, since it aims to reproduce the behaviour displayed in the source task by mainly adjusting the policy based on predictions of the ball dynamics observed in the target task. As mentioned in the results, it exists a gap between the ball positions between the source and the target tasks, which is mainly result of the ball tracking system; therefore the ball dynamics compared between the source and target task differ significatively. The increasing MSE on the inverse-dynamics \( L_2 \) and adjusting-policy \( L_3 \) models across the optimizations could be caused by the state transitions observed in the source data, result of \( A_t \) low transferability; and since the policy adjustment accumulates the training dataset at every iteration, the error caused by the low transferability is propagated in the iterations.

Alternatively, another reason for the unimproved optimization results could lay down in an overfitting effect, which could be glimpsed in the \( L_2 \) and \( L_3 \) low generalization error, and since no mechanism to prevent it was implemented, the models could suffer from it, as it can be seen in reviewed in Appendix D at the \( L_2 \) and \( L_3 \) models prediction plots.
It is important to state that the reliability of such findings is intimately chained to the ball positions mismatch generated mainly by the ball-tracking system in the real-world implementation. Since the measurements of the ball positions introduce a noise factor, the internal reliability is considered low. Tied to this, the validity of the transfer experiments is considered to be limited to the presented implementation.
Chapter 5
Looking forward

5.1 Conclusions

Learning a policy from a tabula rasa state with a model-based algorithm such as PILCO proved to be challenging for the basketball skill in a scenario where a plastic, inflatable ball determines the policy. Albeit the perks of a model-based algorithm in which PILCO models the world as a GP and retains the transition information during the learning process to build the model of the environment, PILCO also suffers from the disadvantage in which the learned model completely biases the learning. In our case, this was reflected in the fact that the ball dynamics turned out to be more stable (easily controlled and predicted) in the source task than in the target task. Hence, PILCO learned to rely on a slower ball. Such a model bias was devised, when PILCO inherently assumed that the learned dynamics model sufficiently and accurately resembled the environment. Therefore, model-biased algorithms are not widely used when learning from tabula rasa: Model bias is specially problematic when few samples and no informative prior knowledge about the task are available.

On the other hand, an RL policy based on actual state variables and performing in a real world scenario turns out to be problematic, since getting an unstable scenario becomes a big concern. Inspite of this risk of instability, there are proposals to train with noise, though, noise is benign when training controllers, compared to unexpected policy delays or a bad model, such as a deficient plastic ball definition. Linked to this, another problem concerning RL and policies is that the optimizer is in effect trying to be as aggressive as possible to get the best out of the model (as visualized in the case of throwing the ball backwards).
As a proposal to build robust policies, there is a trend to train with noise. However, our TL results demonstrate, that the policy adjustment does not necessarily deliver a robust policy in the sense of reliability. The reason could be that it is relatively simple to control with noise because there is no propagation of the error, whereas if there is a problem in the ball dynamics or in the adjustment model, that error is propagated over time. As it is inferred, there is a huge correlation between these errors, whereas noise averages to zero over time.

That said, and despite the demanding nature of the work overall, more attention to the motor limitations in the simulator could have turned into a more reliable testbed for algorithms which learn without prior knowledge. To recall *Throwing ball into the air*, as this was a case where PILCO learned a policy in an almost non-restricted environment, the transference of this policy was limited due to the KLR physical concerns of the high torque and velocities. Naturally, this is not a problem in situations where the learning starts from the same physically restricted system, as in PbD.

### 5.2 Future directions

This work could be further developed from different perspectives and in a number of variations, firstly, by proposing a different strategy for the policy learning, and secondly, by using the resources in a different way or with a different scenario. The policy gradient, considered in PILCO as a way to send feedback to the controller, provides an efficient method of learning and also facilitates the optimizer in its assurance about the policy model, reducing the amplitude of the optimizer’s actions as well as its greediness. However, other proposals for limiting the bandwidth of model-based controllers could also be considered for improvement, such as a policy for a set of multiple models, where the aim is to obtain one robust policy from the overall set. In the same learning direction, other algorithms could be employed, such as Black-Box Data-efficient Policy Search for Robotics\[35\] (Black-DROPS), another model-based policy search, or any other non-model-based policy search. Different skills with a similar RL scenario could also be proposed for this TL-of-RL framework. Another, more radical direction would be an off-policy RL as an alternative to the GP problems, such as the one devised in \[12\] which addresses GP properties like convergence guarantees, off-policy learning and exploration techniques for RL.

For a further PILCO implementation, it would be useful to consider working on the data treatment. Since PILCO does perform some predictions internally about the model, besides its policy execution for every optimization, the major share of the computing time is spent on retrieving the mean and covariances of the GPs. Most of this time is occupied by the task of inverting the Kernel Matrix in order to deliver the covariance, and as our policy is RBF-based, the covariance retrieval could be discarded, hence saving this time. Aiming to overcome this issue, we proposed a subsampling of the training dataset, and although it decreased the GP model time,
the number of optimizations required to achieve a similar result increased compared to the time with a non-subsampled training dataset.

Another countermeasure for the same issue would be to develop a strategy to eliminate intelligently the repeated data (or data that have a certain similarity measurement, like points that are close in time) in the training dataset. As the training data for the ball dynamics is accumulated by PILCO as pairs of states and actions, there is the possibility that after a certain number of executions, some of the pairs will be repeated, which would lead to a bigger (yet no more informative) Kernel matrix. Hence, eliminating the state-action pairs that are the same in content or else occur almost simultaneously, would lead to shorter computing time. Naturally, other alternatives for GP tools software could be employed as well in order to review their performance with PILCO. Furthermore, apart from filtering the input, the Kernel matrix could be manipulated in such a way that, if the elements far from the diagonal in the matrix are set to zero, the bandwidth could be closed depending on how far from the diagonal the zeroed elements are, leading to a noise reduction.

In the hardware perspective, variations could include other ways to retrieve states, such as employing stereo cameras, or using data fusion with other sensors that are able to provide better ball-tracking. Yet, if the current Kinect camera is used, a different calibration to compute the ball position would be of interest, such as the hand-eye camera calibration devised in [36] instead of the extrinsic calibration that was performed. This would most likely overcome the discrepancies between the ball positions and possibly lead to a more valuable transferability of actions.
Bibliography


Appendix A

Cases of PILCO learning in LWRSIM

A.1 Throwing ball backwards

This case, displayed in Figure A.11, refers to the situation when the robot does not learn how to move at the beginning of every episode and drops the ball as result of a sudden movement generated by an aggressive policy exploration. As we saw in Figure 4.3, LWRSIM home position puts the balls in a somehow fragile state. If the policy dictates a bad command for Joint J6, this could make the ball drop immediately and hence, loosing the opportunity to research furthermore if the policy could have leaded to good results or not. Therefore, the action dictated for J6 is the most critical at the beginning of every episode.

![Figure A.11: Throwing the ball backwards at the beginning of the episode.](image)

This case was also influenced by an almost null exploration parameter $k$ and a very poor training dataset. Albeit a poor GP model is not reliable, it leaded to faster single optimizations, which were also speeded by the low optimization parameters in the policy search. Although less time was spent in every single episode optimization, accuracy on GP model and policy search was lost, and therefore, PILCO could not learn that the very first movement was crucial for the rest of the policy, having as consequence to always throw the ball without really learning from it.

This situation was present in the experiment with the highest number of iterations executed, $N_{opt} = 470$, and what made this high number possible were the very low optimization parameters (O-length 5 and O-MFEPLS 2), deriving in a model and
policy very fast to optimize, but very unreliable.

Almost all the experiments drove to this situation and that is why Table A.11 includes most of the tested parameters.

<table>
<thead>
<tr>
<th>General</th>
<th>$N_{opt}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 - 470</td>
</tr>
</tbody>
</table>

| Single optimization | $T$ [s]: 2-4 | $d_t$ [s]: 0.1, 0.01 | $d_{training}$ [s]: 0.1, 0.2, 0.3 0.01, 0.02 |
| Optimization parameters | O-length: 5, 10, 20 | O-MFEPLS: 2, 5 |
| RL | $k$: | $u_{max}$ [deg/s]: |
|    | $\approx 0$ | [50-100,50-100,50-100] |

Table A.11: Parameters that led to the case in which LWRSIM kept throwing the ball backwards

A.2 Dangerous positions and local optimal

Figure A.21 justifies very well the usage of a simulator when an RL algorithm without previous expert knowledge is to be tested. As we know, for an RL algorithm to learn, it needs to perform some degree of exploration and that could include unexpected outcomes such as hitting surfaces if no safety mechanisms are installed.

Figure A.21: Reaching dangerous positions in LWRSIM.

These poses were also found in the case where local optimals were reached and are described as follows. Contrary to the previous case, when PILCO learned that the movement in Joint J6 at the beginning of the episode was crucial, it became sometimes too cautious. As Figure A.22 shows, when the policy found that a slow movement towards its own retraction led to a very tiny (but steady) lower cost, it preferred to stay there and improve the cost, rather than move away from that position which would not guarantee a lower cost. Hence, this result was mainly favoured by a low exploration parameter and maxU. In this case, the size of the
training data did not play a role.

![Figure A.22: PILCO local optimal result.](image)

<table>
<thead>
<tr>
<th>General</th>
<th>$N_{opt}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 - 50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single optimization</th>
<th>$T$ [s]: 2</th>
<th>$d_t$ [s]: 0.1</th>
<th>$d_{training}$ [s]: 0.1, 0.2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Optimization parameters</th>
<th>O-length: 10, 20</th>
<th>O-MFEPLS: 2, 5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>RL</th>
<th>$k$: $\approx 0$</th>
<th>$u_{max}$ [deg/s]: [50-70,50-70,50-70]</th>
</tr>
</thead>
</table>

Table A.21: Parameters that leaded to the case in which LWRSIM gets stuck in local but dangerous optimals.

### A.3 Rolling ball through body

The strategy displayed in Figure A.31 is an interesting alternative to a lack of motion. When PILCO overcomes the previous cases and starts to propose acceptable policies, if maxU is set to low values, PILCO will learn to take advantage of that sluggishness by moving so slow and steady that it will let the ball roll over the robot body and bounce into the table. Although the cost function gets stored only until the first bounce, sometimes the complete episode results in a soft ball bouncing from the table until somewhere close to the basket.

Naturally, to overcome the first two problems, exploration is encouraged and a good model from the training set is fulfilled. Hence, we observed that PILCO learned this behaviour when the training dataset was either complete or just subsampled by the half.
Figure A.31: Learning to roll ball through the robot body and execution cost.

<table>
<thead>
<tr>
<th>General</th>
<th>$N_{\text{opt}}$: 20 - 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single optimization</td>
<td>$T$ [s]: 4</td>
</tr>
<tr>
<td>Optimization parameters</td>
<td>O-length: 20, 40</td>
</tr>
<tr>
<td>RL</td>
<td>$k$: -0.5</td>
</tr>
</tbody>
</table>

Table A.31: Parameters that led to the case in which LWRSIM learns to throw ball so it rolls through the robot’s body.

### A.4 Throwing ball into the air

The combination of parameters displayed in Table A.41 enabled PILCO to learn this policy, in which the ball gets thrown into the air as the inertial result of a fast and similarly directed movement in the joints. This happened once the algorithm overcame the case of *throwing ball backwards*, and sometimes in the early exploration of the forward movement, the algorithm also tried small magnitude and rapid movements on Joint 6 trying to juggle the ball like a tennis player in the air. Albeit its success in the task, this policy is not used for the TL stage due to its high speed requests, and hence high torque, that turned into physical limitations in the real robot.
Figure A.41: Policy learned to throw the ball to the air.

<table>
<thead>
<tr>
<th>General</th>
<th>$N_{opt}$: 100</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single</strong></td>
<td>$T \text{[s]}$: 4</td>
</tr>
<tr>
<td><strong>optimization</strong></td>
<td>O-length: 20</td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td></td>
</tr>
<tr>
<td><strong>parameters</strong></td>
<td></td>
</tr>
<tr>
<td><strong>RL</strong></td>
<td>$k$: -0.8</td>
</tr>
</tbody>
</table>

Table A.41: Parameters that led to the case in which LWRSIM throws the ball into the air towards the basket.
Appendix B

Plots of source task: LWRSIM
B.1 Policy: lowvel1

Figure B.11: Action from policy lowvel1 on source task, $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, for each Joint.

Figure B.12: Position reached on source task as result of $A_t^{(S)}$, for each Joint.
Figure B.13: State $S_t^{(S)}$ on source task as result of $A_t^{(S)}$.

Figure B.14: Execution cost on source task as result of $A_t^{(S)}$. 
B.2 Policy: *lowvel2*

![Figure B.21: Action from policy lowvel2 on source task, $A_t^{(S)} = \pi^{(S)}(S_t^{(S)})$, for each Joint.](image)

![Figure B.22: Position reached on source task as result of $A_t^{(S)}$, for each Joint.](image)
Figure B.23: State $S_t^{(S)}$ on source task as result of $A_t^{(S)}$.

Figure B.24: Execution cost on source task as result of $A_t^{(S)}$. 
Appendix C

Plots of target task: non-adjusted policy
C.1 Policy: *lowvel1*

![Figure C.11: Action from policy *lowvel1* on target task, $A_t(T) = \pi(S_t(T))$, for each Joint.](image)

![Figure C.12: Position reached on target task as result of $A_t(T)$, for each Joint.](image)
Figure C.13: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. 
Figure C.14: Execution cost on target task as result of $A_i^{(T)}$. 
C.2 Policy: lowvel2

Figure C.21: Action from policy lowvel2 on target task, $A_t^{(T)} = \pi^{(S)}(S_t^{(T)})$, for each Joint.

Figure C.22: Position reached on target task as result of $A_t^{(T)}$, for each Joint.
Figure C.23: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. 
Figure C.24: Execution cost on target task as result of \( A_t^{(T)} \).
Appendix D

Plots of target task: adjusted policies across iterations
D.1 Policy: *lowvel1*

D.1.1 Iteration 1: L2 and L3 models

Figure D.11: Test set prediction $L_2 \rightarrow g(.)$ from policy *lowvel1*, for action on each Joint. First iteration.
Figure D.12: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel1, for action on each Joint. First iteration.
D.1.2 Iteration 1: results

Figure D.13: Comparison of action from policy lowvel1 before and after adjustment $\pi_{adj}$, for each Joint. First iteration.

Figure D.14: Action from adjusted policy lowvel1 on target task, $A_t(T) = \pi_{adj}(S_t(T), \pi(S(S_t(T))))$, for each Joint. First iteration.
Figure D.15: Position reached on target task as result of $A_1^{(T)}$, for each Joint. First iteration.
Figure D.16: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. First iteration.
Figure D.17: Execution cost on target task as result of $A_t^{(T)}$. First iteration.
D.1.3 Iteration 2: L2 and L3 models

Figure D.18: Test set prediction $L_2 \to g(.)$ from policy $lowvel1$, for action on each Joint. Second iteration.
Figure D.19: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel1, for action on each Joint. Second iteration.
D.1.4 Iteration 2: results

Figure D.110: Comparison of action from policy $lowvel1$ before and after adjustment $\pi_{adj}$, for each Joint. Second iteration.
Figure D.111: Action from adjusted policy $lowvel1$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$, for each Joint. Second iteration.

Figure D.112: Position reached on target task as result of $A_t^{(T)}$, for each Joint. Second iteration.
Figure D.113: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Second iteration.
Figure D.114: Execution cost on target task as result of $A_t^{(T)}$. Second iteration.
D.1.5 Iteration 3: L2 and L3 models

Figure D.115: Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel1, for action on each Joint. Third iteration.
Figure D.116: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy *lowvel1*, for action on each Joint. Third iteration.
D.1.6 Iteration 3: results

Figure D.117: Comparison of action from policy `lowvel1` before and after adjustment $\pi_{adj}$, for each Joint. Third iteration.
Figure D.118: Action from adjusted policy `lowvel1` on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S_t^{(T)}))$, for each Joint. Third iteration.

Figure D.119: Position reached on target task as result of $A_t^{(T)}$, for each Joint. Third iteration.
Figure D.120: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Third iteration.
Figure D.121: Execution cost on target task as result of $A_i^{(T)}$. Third iteration.
D.1.7 Iteration 4: L2 and L3 models

Figure D.122: Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel1, for action on each Joint. Fourth iteration.
Figure D.123: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy $lowvel1$, for action on each Joint. Fourth iteration.
D.1.8 Iteration 4: results

Figure D.124: Comparison of action from policy lowvel1 before and after adjustment $\pi_{adj}$, for each Joint. Fourth iteration.
Figure D.125: Action from adjusted policy lowvel1 on target task, \( A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S_t^{(T)})) \), for each Joint. Fourth iteration.

Figure D.126: Position reached on target task as result of \( A_t^{(T)} \), for each Joint. Fourth iteration.
Figure D.127: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Fourth iteration.
Figure D.128: Execution cost on target task as result of $A_t^{(T)}$. Fourth iteration.
D.2 Policy: lowvel2

D.2.1 Iteration 1: L2 and L3 models

Figure D.21: Test set prediction $L_2 \rightarrow g(\cdot)$ from policy lowvel2, for action on each Joint. First iteration.
Figure D.22: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy $lowvel2$, for action on each Joint. First iteration.
D.2.2 Iteration 1: results

Figure D.23: Comparison of action from policy $\text{lowvel2}$ before and after adjustment $\pi_{\text{adj}}$, for each Joint. First iteration.
Figure D.24: Action from adjusted policy lowvel2 on target task, \( A_t^{(T)} = \pi_{\text{adj}}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)})) \), for each Joint. First iteration.
Figure D.25: Position reached on target task as result of $A_t^{(T)}$, for each Joint. First iteration.
Figure D.26: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. First iteration.
Figure D.27: Execution cost on target task as result of $A_t^{(T)}$. First iteration.
D.2.3 Iteration 2: L2 and L3 models

Figure D.28: Test set prediction $L_2 \rightarrow g(.)$ from policy lowvel2, for action on each Joint. Second iteration.
Figure D.29: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel2, for action on each Joint. Second iteration.
D.2.4 Iteration 2: results

Figure D.210: Comparison of action from policy \textit{lowvel2} before and after adjustment $\pi_{adj}$, for each Joint. Second iteration.
Figure D.211: Action from adjusted policy $lowvel2$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$, for each Joint. Second iteration.
Figure D.212: Position reached on target task as result of $A_t^{(T)}$, for each Joint. Second iteration.
Figure D.213: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Second iteration.
Figure D.214: Execution cost on target task as result of $A_t^{(T)}$. Second iteration.
D.2.5 Iteration 3: L2 and L3 models

Figure D.215: Test set prediction $L_2 \rightarrow g(.)$ from policy $lowvel2$, for action on each Joint. Third iteration.
Figure D.216: Test set prediction $L_3 \rightarrow \pi_{\text{adj}}$ from policy \textit{lowvel2}, for action on each Joint. Third iteration.
D.2.6 Iteration 3: results

Figure D.217: Comparison of action from policy \textit{lowvel2} before and after adjustment $\pi_{adj}$, for each Joint. Third iteration.
Figure D.218: Action from adjusted policy $lowvel2$ on target task, $A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi^{(S)}(S_t^{(T)}))$, for each Joint. Third iteration.
Figure D.219: Position reached on target task as result of $A_t^{(T)}$, for each Joint. Third iteration.
Figure D.220: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Third iteration.
Figure D.221: Execution cost on target task as result of $A_t^{(T)}$. Third iteration.
D.2.7 Iteration 4: L2 and L3 models

Figure D.222: Test set prediction $L_2 \rightarrow g(.)$ from policy *lowvel2*, for action on each Joint. Fourth iteration.
Figure D.223: Test set prediction $L_3 \rightarrow \pi_{adj}$ from policy lowvel2, for action on each Joint. Fourth iteration.
D.2.8 Iteration 4: results

Figure D.224: Comparison of action from policy lowvel2 before and after adjustment $\pi_{adj}$, for each Joint. Fourth iteration.
Figure D.225: Action from adjusted policy \( \text{lowvel2} \) on target task, \( A_t^{(T)} = \pi_{adj}(S_t^{(T)}, \pi(S_t^{(T)})) \), for each Joint. Fourth iteration.
Figure D.226: Position reached on target task as result of $A_t^{(T)}$, for each Joint. Fourth iteration.
Figure D.227: State $S_t^{(T)}$ on target task as result of $A_t^{(T)}$. Fourth iteration.
Figure D.228: Execution cost on target task as result of $A_t^{(T)}$. Fourth iteration.