Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
Computational Modeling Lab

Continuous Action Reinforcement Learning Automata, an RL technique for controlling production machines.

Abdel Rodríguez Abed

Dissertation Submitted for the degree of Doctor of Philosophy in Sciences

Supervisors:	Prof. Dr. Ann Nowé
	Prof. Dr. Peter Vrancx
	Prof. Dr. Ricardo Grau Ábalo

Abstract

Reinforcement learning (RL) has been used as an alternative to model based techniques for learning optimal controllers. The central theme in reinforcement learning research is the design of algorithms that learn control policies solely from the knowledge of transition samples or trajectories, which are collected by online interaction with the system. Such controllers have become part of our daily life by been present at from almost every home appliance to really complex machines in industry. This extension to the frontiers of applications of RL has also demanded from the learning techniques to face more and more complex learning problems.

This dissertation is centered in the changes necessary to an existing simple RL algorithm, the continuous action reinforcement learning automaton (CARLA) to control production machines. For a better introduction of the results we first present some background information about the general framework for bandit applications, Markov decision process (MDP), and RL for solving problems with discrete and continuous state-action spaces. The changes to the algorithm aim to optimize the computational and sampling cost as well as to improve local and global convergence. The standard CARLA method is not able to deal with state information. In this dissertation we also propose a method to take state information into account in order to solve more complex problems.

After introducing these theoretical results, we present a number of problems related to production machines and the appropriate way to use the described method to solve them. The applications vary in complexity in terms of stationarity, centralization and state information.

To Enislay and my family

Acknowledgements

Contents

List of Algorithms **xvii**

Glossary **xix**

Acronyms **xxv**

1 Introduction **1**

1.1 Problem statement 2

1.2 Contributions 4

1.3 Outline of the dissertation 4

2 Bandit problems **9**

2.1 Discrete bandit solvers 10

2.1.1 Learning Automata 12

2.2 Discrete games 14

2.2.1 Exploring Selfish RL 17

2.3 Continuous bandit problems 19

2.3.1 Continuous Action Learning Automaton 19

2.3.2 Hierarchical Optimistic Optimization 21

2.3.3 Continuous Action Reinforcement Learning Automata 21

2.4 Summary 25

3 Reinforcement learning **27**

3.1 Markov decision processes 27

3.1.1 Deterministic settings 27

3.1.2 Stochastic settings 31

3.2 Reinforcement learning 32

3.3 Value iteration RL 34
3.4 Policy iteration RL 37
3.5 Summary 38

4 Reinforcement Learning in continuous action-state spaces 39
4.1 Parametric approximators 40
4.2 Non-parametric approximators 41
4.3 RL architectures 44
4.3.1 Approximate value iteration 44
4.3.2 Approximate policy iteration 46
4.4 Summary 53

5 CARLA extensions 55
5.1 Speeding up calculations 55
5.2 Analysis of convergence. Dynamically tuning the spreading rate 58
5.3 Standardization of rewards 63
5.4 Analysis of convergence in games 66
5.5 ESCARLA 71
5.6 MSCARLA 75
5.6.1 Policy representation 75
5.6.2 Policy evaluation 78
5.7 Summary 79

6 State-less applications 81
6.1 Test cases 81
6.1.1 Single agent 81
6.1.2 Games 86
6.2 Real world applications 90
6.2.1 Linear motor 91
6.2.1.1 Description 91
6.2.1.2 Results 93
6.2.2 Learning PID parameters to control an autonomous vehicle 93
6.2.2.1 Description 93
6.2.2.2 Results 98

6.2.3 Wire winding machine . . . 101
6.2.3.1 Description . . . 103
6.2.3.2 Results . . . 104
6.2.4 A hydrologic problem . . . 106
6.2.4.1 Description . . . 107
6.2.4.2 Results . . . 108
6.3 Summary . . . 109

7 Multi-state applications **111**
7.1 Test cases . . . 111
7.1.1 Double integrator problem . . . 111
7.1.1.1 Description . . . 112
7.1.1.2 Results . . . 113
7.1.2 DC motor . . . 115
7.1.2.1 Description . . . 117
7.1.2.2 Results . . . 118
7.2 Real world applications . . . 121
7.2.1 Autonomous tractor . . . 121
7.2.1.1 Description . . . 122
7.2.1.2 Results . . . 123
7.2.2 Autonomous tractor plus an implement . . . 125
7.2.2.1 Description . . . 128
7.2.2.2 Results . . . 128
7.3 Summary . . . 129

8 Summary **133**
8.1 Contributions . . . 134
8.2 Future work . . . 136

List of Figures

1.1 Automatic control 2
1.2 Outline of the dissertation 7

2.1 A common interest 2-players 2-actions example game 15
2.2 The climbing game 16
2.3 The climbing game after shrinking the joint-action space 18
2.4 Sketch of CARLA action space exploration 24

3.1 The controller-environment interaction in RL 33

4.1 Tile coding approximator in 2D 42
4.2 A conceptual illustration of approximate value iteration 45
4.3 Integral Reinforcement Learning 50
4.4 Multi-agent Integral Reinforcement Learning 52

5.1 Standardization of rewards 63
5.2 A two players game with three local optima 70
5.3 Relation between covariance and exploration 72
5.4 Mapping process 74
5.5 Discounted sum over the trajectory 78

6.1 Graphic representation of the reward functions 83
6.2 Comparison on continuous action bandit problems 85
6.3 Comparison on an non-stationary bandit problem 86
6.4 Reward functions of Two 2-player games 87
6.5 Comparison on continuous action games 88

6.6 Region elimination process 89
6.7 Sketch of the linear motor set-up 92
6.8 Reward functions for the linear motor set-up 92
6.9 Results on the linear motor control task 94
6.10 Controlling an autonomous vehicle 96
6.11 A reference trajectory for the tractor 97
6.12 Error during the learning process of the slip factor 99
6.13 Average accumulated error during the learning process of the slip factor 100
6.14 Accumulated error during the learning process of the slip factor 101
6.15 Accumulated error during the learning process of the slip factor 102
6.16 Wire winding machine 103
6.17 Flatness of the surface measured as Δp 104
6.18 Wire winding results in a flat bobbin 105
6.19 Wire winding results in a round bobbin 106
6.20 Water reservoirs problem 107
6.21 Reward function of the water reservoirs problem 108

7.1 Optimal solution for the double integrator problem 113
7.2 Learned policies to control the double integrator problem using a tile coding . . 114
7.3 Learned policies to control the double integrator problem a using Gaussian kernel 116
7.4 Accumulated cost during learning process 117
7.5 Optimal solution for the DC motor problem 118
7.6 Learned policies to control the DC motor problem using a tile coding 119
7.7 Learned policies to control the DC motor problem using a Gaussian kernel . . . 120
7.8 Accumulated cost during learning process 121
7.9 Error during the learning process using MSCARLA 123
7.10 Accumulated error during the learning process using MSCARLA 124
7.11 Policy for driving the tractor when turning right 125
7.12 Policy for driving the tractor when going straight 126
7.13 Policy for driving the tractor when turning left 127
7.14 Controlling an implement 128
7.15 A reference trajectory for the tractor and the implement 129
7.16 Error during the learning process using MSCARLA 130

7.17 Accumulated error during the learning process using MSCARLA 131

List of Tables

6.1 Parameter setting of HOO for continuous action functions 84
6.2 Comparison between the different implementations of CARLA 91
6.3 Results of ESCARLA on the water reservoirs problem 109

List of Algorithms

2.1 ϵ-greedy strategy 11
2.2 ϵ-first strategy 11
2.3 CALA methods (Thathachar & Sastry, 2004, Section 1.6). 20
2.4 HOO method (Bubeck *et al.*, 2011). 22
2.5 CARLA method (Howell *et al.*, 1997). 24
3.1 Q-learning (Buşoniu *et al.*, 2010, Section 2.3). 36
3.2 SARSA-learning (Buşoniu *et al.*, 2010, Section 2.4). 36
3.3 Policy iteration. 37
4.1 Approximate Q-learning with a linear parametrization (Buşoniu *et al.*, 2010, Chapter 3). 46
4.2 SARSA with a generic approximation mapping (Buşoniu *et al.*, 2010, Chapter 3). 47
4.3 Actor-critic with Gaussian exploration (Buşoniu *et al.*, 2010, Chapter 3). 49
4.4 CACLA algorithm (van Hasselt, 2012). 50
4.5 IRL algorithm (Vrabie *et al.*, 2009). 51
5.1 CARLA method with no unnecessary numerical integration. 58
5.2 CARLA method with automatic tuning of the spreading rate. 62
5.3 CARLA method with reward transformation. 65
5.4 Exploring Selfish CARLA 74
5.5 Exploration phase (ESCARLA) 75
5.6 Synchronization phase (ESCARLA) 75
5.7 MSCARLA's general procedure. 80

Glossary

G | R

G

Greek letters

Λ	Model of a process. 19
Σ	Positive definite matrix been the Gaussian kernel width matrix. 43
α	Learning rate. 13, 19, 20, 23, 24, 34–36, 44–50, 53–60, 63, 65, 67, 69, 78, 80, 81
β	Penalty rate. 13
$\delta_{TD,k}$	Temporal difference at time step k. 37, 48–50
δ	Extra area added to the PDF during every CARLA update. 55, 56, 60, 63
ϵ	Refers to a small constant. 10, 11, 13, 20, 35, 36, 46, 47
η	Standardization factor used in the CARLA method. 23, 24, 54–60, 63–65, 67
γ	Discount factor. 10, 28–32, 34–37, 44–50, 76, 78
κ	Kernel function. 43, 75, 76
λ	Spreading rate of the CARLA method. 23, 24, 53–60, 63, 65–67, 69, 77, 78
μ	Mean. 11, 12, 19, 20, 22, 41, 55, 61–63, 80, 81, 88
ϕ	Basis function. 40, 41, 46
π	Policy for selecting actions. It is used for controllers, RL and bandit problems interchangeably. Notice that since in bandit

	applications the learner is not provided with the state of the system then when used in the case of bandit problems the function takes no parameter while in the RL case in considers only one parameter been the state of the system. 9, 14, 15, 28–32, 37, 47–50, 53, 74, 76
ψ	Tiling function. 41, 74–76
ρ	The feedback function describing the rewards. The function normally takes as parameters the action taken and the state of the system. In RL problems it can also consider both states, the one where the action was taken and the one the system transited to after applying the action. When used in stationary bandit problems, meaning that the reward does not change over time regardless the actions taken (or the state of the system), the function is used with only one parameter been the action. 9, 10, 14, 15, 28–32, 54–59, 61, 62, 64–68, 76, 80–86, 88, 89, 96
σ_L	Lower bound for the standard deviation of the policies for action selection in the CALA method. 19, 20, 81
σ	Standard deviation. 19, 20, 49, 50, 55, 60–63, 81, 88
τ	Temperature parameter. 35
θ	Function approximator's parameter. 40, 41, 43–50, 74, 75
ϱ	Parameter giving information about the reward function in the HOO method. In the original paper the authors used ρ but we switch to ϱ to avoid confusions with the reward signal. 21, 22, 81, 82
ϑ	Function approximator's state-dependent parameter. 47–50, 74–76, 78

R

Roman letters

B	Optimisitc estimate to the maximum mean payof in the HOO method. 21, 22
E	Expected value. 10, 31, 32, 62

F	Cumulative density function corresponding to the probability density function been the policy for selecting actions. 55, 56, 60, 63
H	Approximation mapping. 40, 41, 43–47, 74, 75
J	Performance index. 19
K_+	Large positive constant used in the CALA method. 19, 20, 81
K_r	The horizon length for calculating the reward mean and standard deviation for the reward transformation on CARLA method. 63, 78
K_u	The horizon length. 29, 31, 76, 78
K	Gain matrix. 92, 94, 96
L	The amount of actions allowed in a discrete bandit problem. 9, 13
N	Horizon length in ϵ-first strategies. 10, 11
P	Projection mapping. 45
Q	Action-state value function. Represents the expectation of the return at a given state by taking some action (both been parameters of the function) and selecting the actions according to the current policy thereafter. 29, 30, 32, 34–36, 40, 44–46
R	The (un)discounted return. 28–32, 49, 76
T	Q-iteration mapping. 44, 45
U	Action space. It is used for controllers, RL and bandit problems interchangeably. Depending on the problem, it may refer to a subset of $\mathbb{R}^n$ or of $\mathbb{Z}^n$. 11–13, 19, 21–24, 27–29, 31, 35, 40, 41, 43, 46, 47, 53–60, 63–67, 72, 74, 80
V	State value function. Represents the expectation of the return at a given state when starting at the state described by the parameter of the function and selecting actions with the current policy (normally indicated as a superscript) thereafter. 9, 10, 30, 32, 37, 47–50
W	Weighting function. 74–78

Glossary

X	State space. It is used for controllers, RL and bandit problems interchangeably. Depending on the problem, it may refer to a subset of $\mathbb{R}^n$ or of $\mathbb{Z}^n$. 27–31, 40, 41, 43, 46, 49, 74, 78
$\mathfrak{R}$	The feedback function describing the rewards. 19
$\mathscr{N}$	Normal distribution. 19, 20, 49, 50, 54, 55
$\mathscr{Q}$	Space of all possible Q-functions. 40, 44, 45, 47
$\mathscr{T}$	Transformation of the probabilities in an LA method after exploring an action. 13
$\mathscr{V}$	Space of all possible V-functions. 47
c	Counter of the visits to each node used in HOO. 22
f	Probability density function been the policy for selecting actions. 23, 24, 53–55, 57–60, 62, 64–67, 71, 73
h	Depth in the tree of a given node used in HOO. 21, 22
i	Position within the level of the tree of a given node used in HOO. 21, 22
k	Discrete time steps. It is used for RL and bandit problems interchangeably. 9–13, 19–24, 27–31, 33–37, 43–50, 53–61, 63–67, 75, 76, 78, 82, 84, 90, 95, 96, 106, 111, 112, 116
l	Refers to each player of a game. 14, 15, 64–67, 72
p	Probability distribution over the actions. It is normally used with a subscript describing the time step and it takes one parameter been the action. 13, 19
r	The observed reward. It is used in RL and bandit problems interchangeably. It is normally used with a subscript describing the time step but it may also appear with a superscript meaning the action that produced the reward. 9, 11, 13, 19, 20, 22–24, 28, 31, 33–37, 44–50, 56, 60, 63, 76, 78, 90, 106–108, 110, 112–114, 116
t	State transition function. 28–32
u	A specific action. It is used for controllers, RL and bandit problems interchangeably. 9–13, 15–20, 22–24, 27–37, 40, 41, 43–50, 53–68, 74–76, 78, 80, 82, 84, 85, 88, 92, 94, 96, 106–108, 110–114, 116

v_1	Parameter giving information about the reward function in the HOO method. 21, 22, 81, 82
x	A particular state. It is used in RL and bandit problems interchangeably. 9, 10, 27–37, 40, 41, 43–50, 74–76, 78, 92, 94, 106, 111, 112, 116

Glossary

Acronyms

C | **D** | **E** | **F** | **H** | **L** | **M** | **N** | **P** | **R** | **S**

C

CACLA	(CACLA) continuous actor-critic learning automaton. 48, 107, 109, 112, 115
CALA	(CALA) continuous action learning automaton. 2, 19–21, 23, 25, 81–84
CARLA	(CARLA) continuous action reinforcement learning automaton. iii, 2–6, 23–25, 53, 61, 64, 67, 72–77, 79–86, 88–92, 94, 96, 103, 109, 115, 123, 128, 129, 131
CDF	(CDF) cumulative density function. 24, 55

D

DP	(DP) dynamic programming. 1

E

ESCARLA	(ESCARLA) exploring selfish continuous action reinforcement learning automaton. 4, 69, 72, 85, 86, 88, 89, 103, 122, 123, 129
ESRL	(ESRL) exploring selfish reinforcement learning. 3–6, 17, 18, 25, 53, 69, 77, 127–129

F

FALA	(FALA) finite action learning automaton. 13, 23

H

HOO	(HOO) hierarchical optimistic optimization. 21, 25, 79, 81–84

L

LA	(LA) learning automaton. 2, 4, 5, 12, 13, 17, 21, 25, 37, 69, 73, 77

M

MDP	(MDP) Markov decision process. iii, 5, 15, 27, 28, 31, 39, 53, 73, 77, 105, 115, 127
MSCARLA	(MSCARLA) multi-state continuous action reinforcement learning automaton. 73–75, 77, 107, 109, 111, 112, 115, 116, 119, 123, 130

N

NE	(NE) Nash equilibrium. 15, 69

P

PDF	(PDF) probability density function. 23, 24, 53–55, 57, 62, 64, 66, 74

R

RL	(RL) reinforcement learning. iii, 1–3, 5, 12–15, 17, 19, 23, 27, 28, 32–35, 37–41, 43, 44, 51, 53, 73, 82, 94, 101, 104, 105, 109, 118, 123, 127–129

S

SARSA	state, action, reward, (next) state, (next) action. 35, 36, 39, 46, 48, 105, 107, 109, 111, 112

Chapter 1

Introduction

Control systems have an immense impact in our daily life. Nearly all devices, from basic home appliances to complex plants and aircrafts, operate under the command of such systems. The ultimate goal of control is to influence or modify the behavior of dynamic systems to attain pre-specified goals. A common approach to achieve this, is to assign a numerical performance index to each state in the trajectory of the system and then to search for the policy that drives the system along trajectories corresponding to the best value of the performance index.

Dynamic programming (DP) (Bellman, 2003) originated in the early research of optimal control and it is still used nowadays in applications where a system model is available. Reinforcement learning (RL) (Sutton & Barto, 1998) has emerged as an alternative to this model based approach, for finding the solution in absence of a model. The central theme in RL research is the design of algorithms that learn control policies solely from the knowledge of transition samples or trajectories, which are collected by online interaction with the system.

RL is an algorithmic method for solving problems in which actions (decisions) are applied to a system over an extended period of time, in order to achieve a desired goal. The time variable is usually discrete and actions are taken at every discrete time step, leading to a sequential decision-making problem. The actions are taken in closed loop, which means that the outcome of earlier actions is monitored and taken into account when choosing new actions. Rewards are provided that evaluate the one-step decision-making performance, and the goal is to optimize the long-term performance, measured by the total reward accumulated over the course of interaction.

RL research is not limited to automatic control applications. This decision making technique has been successfully used in a variety of fields such as operations research, economics,

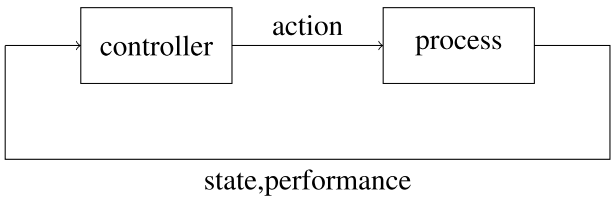


Figure 1.1: Automatic control - A suited application for RL.

and games. In automatic control, a process is to be controlled by a controller which is deciding on the best action to achieve a given goal, the general architecture is shown in figure 1.1. This controller based the decision on the output feeded back from the process. RL is a very suited technique for solving this problem if we use it for learning the control action based on the interactions with the process when a model of the system cannot be obtained at all, e.g., because the system is not fully known beforehand, is insufficiently understood, or it is just too costly to derive such a model.

1.1 Problem statement

This dissertation is centered around the framework stated above: using RL for learning optimal controllers. More specifically, we focuss our approach on learning automatons (LAs). LA are reinforcement learners that keep a probability distribution over the action space for selecting these actions. The probability distribution is updated over time based on the reward received at every time step. This technique has shown good convergence results in discrete settings. Different implementations of LAs exist for continuous action spaces. Some methods such as the continuous action learning automaton (CALA) use a parametric representation (Thathachar & Sastry, 2004) making the update of such a distribution less expensive, computationally speaking, but also restricting the search over the action space. Furthermore, these methods normally are subjected to some constrains to the reward function such as smoothness and are really affected by noise. Other methods such as the continuous action reinforcement learning automaton (CARLA) use a nonparametric representation of the probability distribution (Howell *et al.*, 1997) allowing for a richer exploration of the action space but at the same time turning the method much more demanding from the computational point of view. We present in this dissertation an improvement of the CARLA method in terms of computation effort and local convergence properties. The improved automaton performs very well in single agent problems.

Conventional controller design problems assume that all the controllers present in the system have access to the global information. However since systems tend to become more and more complex, distributed control enters the picture. A distributed control application assumes that there exist several subsystems, each one being controlled by an independent controller preferably sensing only local information. We may refer to formation stabilization, supply chains, wind farms or irrigation channels as application examples. While each application has its own specificities, the general idea is that the subsystems with their basic controllers should be brought together in a kind of plug and play manner and still operate successfully. Since these subsystems might have complex interactions this is not trivial.

The CARLA algorithm has successfully been used as conventional controllers (Howell & Best, 2000; Howell *et al.*, 1997). Even in a distributed approach, we may expect this algorithm to converge to the unique optimum when the system exhibits linear dynamics and a linear cost function is considered. If the system dynamics or its cost function are not linear, multiple optima may exist. If we use the standard CARLA algorithm for controlling each subsystem in a nonlinear interacting system while ignoring the presence of the other controllers, the system will successfully converge, however convergence to the global optimum cannot be guaranteed. In this dissertation we also propose a better exploration of the continuous action space inspired on the exploring selfish reinforcement learning (ESRL) approach for discrete action spaces (Verbeeck, 2004).

CARLA is a bandit solver (see chapter 2 for the explanation of bandits) limited to stateless problems. It has been applied to learn the optimal parameters of linear controllers in continuous state spaces though. The application was limited to systems with linear dynamics. It is common in conventional control to build a model which is a linearized abstraction of the real plant, calculate the optimal controller and use it in the real plant. Although this approach has successfully been applied in practice, there are plenty of applications where a linearized model is not reliable enough. There already exist approaches of RL for control tasks using function approximators to enable discrete RL techniques to large and continuous spaces (Buşoniu *et al.*, 2010). Using the natural capabilities of this method for dealing with continuous optimization over the action space, we combine it with function approximators over the state space in order to obtain an RL technique for continuous action-state spaces.

1.2 Contributions

My contributions are presented in chapters 5, 6, and 7. The former chapter introduces the theoretical changes to the CARLA method. Every section of the chapter corresponds to one individual improvement.

If we want to control real production machines we can only rely on simple processors attached to these machines. This last restriction, in addition to the limited time that the learners have for responding with the appropriate control action, lead us to the first contribution being a significant reduction of the computational cost of the method. This reduction is carried out by analytical derivations of the learning rule to avoid numerical integration.

Since collecting data from a real machine is very costly in time (and most probably in effort as well) it is also necessary to use a method that can learn as fast as possible using an optimal exploration of the action space. My second contribution is an analysis of the convergence of the method and a procedure to adjust the spreading rate of the method (which is a parameter controlling the exploration ratio) that, in combination with the reward standardization, leads to a faster convergence to the optimal solution in terms of sampling.

My third contribution aims to distributed control applications. Inspired in the ESRL method for coordinating discrete LAs, with limited communication, in order to guarantee convergence to the global optimum (escaping local optima) we introduce exploring selfish continuous action reinforcement learning automaton (ESCARLA) which is a version of ESRL but for continuous action spaces. This change improve the global convergence (in terms of ratio of convergence to the global maximum) of the method in distributed control set-ups.

The last theoretical result presented in this dissertation is the combination of CARLA method with function approximators in order to deal with multi-state applications. Such a combination allows the learning method to deal with more complex production machines.

The practical applications are separated into state-less and multi-state applications. The former group is explained in chapter 6 while the latter in 7. Both chapters use a set of toy problems to support the theoretical achievements and afterwards explain the results achieved by the method in the diverse set of applications in the scope of this research.

1.3 Outline of the dissertation

The research presented in this dissertation is divided in eight chapters.

Chapter 2 introduces the general framework for bandit applications. In such a setting, a learner is posed the task to maximize the long-run pay-off with no information of the state of the system. The chapter is divided in discrete and continuous armed bandits. The framework is described for both settings and the strategies for solving them are also presented. The chapter finishes with a section describing discrete action games where two or more learners interact in order to minimize their regret while learning as well. ESRL is presented in the latter section as an exploration method for a set of independent LAs playing a repeated discrete action game.

After discussing bandit applications, the standard reinforcement learning is introduced in chapter 3. In this chapter, we formally give the definition of the Markov decision process (MDP) in order to make easier the understanding of reinforcement learning. We first present the general framework for the deterministic case. Afterwards, the stochastic case is considered. We also discuss optimality for both cases. Once the MDP definition is formalized, we proceed to describe the standard reinforcement learning. This formalization is limited to discrete action-state spaces. The three well-known classes of reinforcement learning are presented: value iteration, policy iteration and direct policy search. For each class, the most popular algorithms in the literature are also explained.

Chapter 4 describes the state of the art of reinforcement learning for large and continuous action-state spaces. This extension of the standard discrete RL techniques is performed by means of function approximators to represent the value functions over the infinite action-state space. Two classes of approximators are described: parametric and non-parametric. The former class defines a fix set of parameters in advance while in the latter class the set of parameters of the approximator is build with the available data at every instant during learning. After formally introducing the function approximators, the three RL architectures, (approximate value iteration, approximate policy iteration, and approximate policy search) are redefined and well-known algorithms are provided.

The above listed chapters introduce the necessary background of this dissertation. Chapter 5 formalizes the extensions we made to the CARLA algorithm in order to make it feasible for the applications tackled in this research. The first change in the original method is performed by pushing further the mathematical analysis of the method and avoiding the numerical integration. This enables the method to run faster on simple processors attached to the engines that are to be controlled. After obtaining this result, the analysis of the convergence of the method is performed in bandit applications where only one learner is present. This analysis makes possible to derive a simple way of tuning the spreading rate parameter of the method which in addition with the reward transformation, also introduced in this chapter, improves its performance from the sampling point of view. The convergence analysis is then performed in games where multiple learners are interacting. As a consequence of this analysis, the necessity of a better exploration protocol is demonstrated and the extension of ESRL to continuous action spaces is introduced. Finally, in order to enable the CARLA method for optimizing the control action in nonlinear dynamical systems the learning technique is combined with function approximators over the state space.

Having introduced the method developed for coping with the different applications we face in this dissertation, we separate the applications in two groups, state-less applications and multi-state applications. Chapter 6 shows the state-less applications. First, some test problems are shown in order to demonstrate the performance of the learning methods derived in this dissertation. Afterwards, the real world applications are presented and tested. Chapter 7 shows the multi-state applications. Again, some toy environments are first introduced before showing the results in the real world applications. All real world applications are in the field of optimal control and distributed control where some engines are to be automatically controlled in order to achieve a target performance. The applications come from the LoCoPro project (grant nr. 80032) of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

Chapter 8 summarizes the dissertation with the conclusions.

Figure 1.2 graphically sums up the outline of the dissertation.

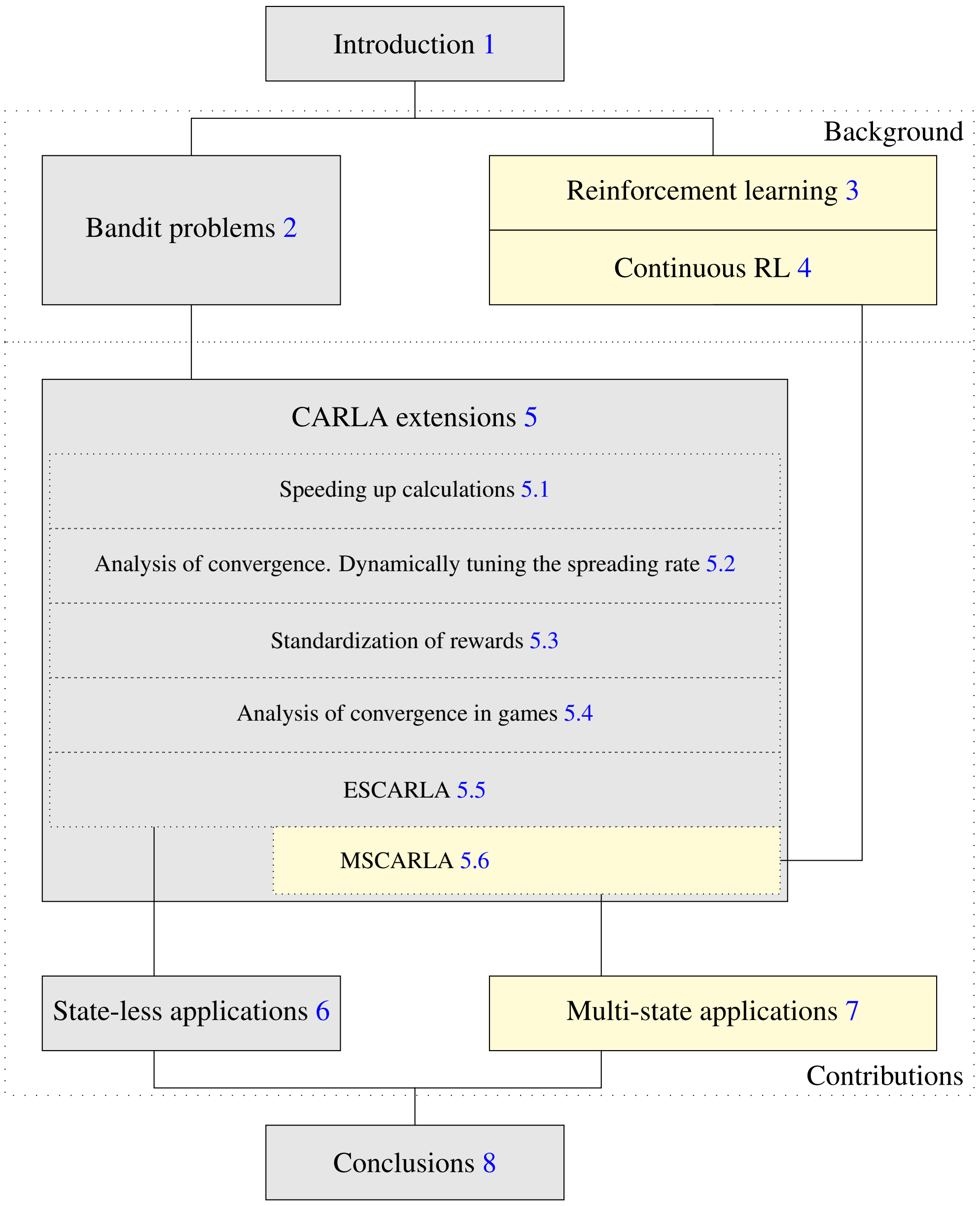

Introduction 1
Background
Bandit problems 2
Reinforcement learning 3
Continuous RL 4
CARLA extensions 5
Speeding up calculations 5.1
Analysis of convergence. Dynamically tuning the spreading rate 5.2
Standardization of rewards 5.3
Analysis of convergence in games 5.4
ESCARLA 5.5
MSCARLA 5.6
State-less applications 6
Multi-state applications 7
Contributions
Conclusions 8

Figure 1.2: Outline of the dissertation - Readers interested in state-less applications can directly read chapter 5 after chapter 2 (skipping chapters 3 and 4) and skip section 5.6 and chapter 7 afterwards. Readers interested in applications that involve information about the state of the system can only skip chapter 6

Chapter 2

Bandit problems

The multi-armed bandit problem is associated to the decision on the lever to pull at a row of slot machines. Each lever gives a random reward when pulled. The objective of the decision maker is to maximize the average reward obtained by pulling the levers. This problem was originally described by Robbins (1952) as a statistical decision model and it has attracted the attention of many researchers ever since for it poses an interesting dilemma on deciding to what extend to explore (trying out each arm to find the best one) and when to exploit the acquired knowledge (playing the arm we feel to give the best payoff). This dilemma is know as the exploration-exploitation dilemma. Each choice of a lever results in an immediate random reward, but the processes determining these rewards might evolve over time.

Formally, in a bandit problem, a decision maker chooses an action $u_k \in \{1, \cdots, L\}$ in discrete time with indexes $k = 0, 1, \cdots$. The state of the process generating the rewards is denoted by x_k and it is unknown to the decision maker. With every decision there is a random payoff r_k observed and we denote the function describing such values by $\rho(u_k, x_k)$. A feasible Markov policy $\pi = \{u_k\}_{k=0}^{\infty}$ selects an available alternative over time.

Several control applications can be seen as bandit problems. Situations where there are parameters to be controlled in order to have the machine operating at a target performance and such a performance may change depending on the use of the machine given some unknown function.

The value function $V(x_0)$ can be written in terms of $\rho(u, x_k)$. The problem of finding an optimal policy is the solution of the problem posed in equation (2.1).

$$V(x_0) = \max_u \left\{ E \sum_{k=0}^{\infty} \gamma^k \rho(u_k, x_k) \right\} \tag{2.1}$$

Several methods for optimally solving bandit problems have emerged. Next, in section 2.1, some general techniques for solving the many-armed bandit (or just bandit for short) problem are shown. Afterwards, in section 2.3, the extension to infinitely many-arms is shown as well as the techniques to solve it.

2.1 Discrete bandit solvers

First we list some exploration strategies that are commonly used in RL because of their simplicity but yet good performance. Then we also provide the more theoretically supported exploration strategies for discrete n-armed bandits.

Semi-uniform strategies are probably the earliest and simplest. These strategies normally behave greedy except when exploring a random action. As a first example consider the ϵ-greedy strategy. In an ϵ-greedy strategy, the decision maker behaves greedy in a proportion $1-\epsilon$ of the trials. In the remaining proportion ϵ of the trials the decision maker uniformly selects an action. The parameter ϵ is normally selected very low, e.g., 0.1, but this can vary widely depending on circumstances and predilections. Algorithm 2.1 shows this strategy. Later upgrades of the algorithm use a dynamic ϵ value, such as in the case of the ϵ-decreasing strategy, in order to reduce exploration over time. As a second example consider the ϵ-first strategy in which two delimited phases are separated: a pure exploration phase and a pure exploitation phase. If the decision maker is to perform N trials in total then the exploration phase lasts for ϵN trials while the exploitation phase takes place during the next $(1-\epsilon)N$ trials. During the exploration phase, the decision maker uniformly selects the actions while during the exploitation phase the action is selected greedily. Algorithm 2.2 shows this strategy.

Regret minimization strategies minimize the regret over time. The regret is defined as the loss in the reward in terms of the difference between the observed reward and the maximum possible reward. As an example of regret minimization strategies consider the Upper Confidence Bound (UCB) algorithm (Auer *et al.*, 2002). In this algorithm the key feature is to estimate an upper confidence bound for the regret. For this goal, the algorithm keeps a counter

Algorithm 2.1 ϵ-greedy strategy

```
Input: exploration-exploitation trade-off $\epsilon$
1: for all action $u \in U$ do
2:     initialize expected reward $\mu_u$ for lever $u$ {e.g., maximum possible reward}
3: end for
4: for $k = 1, 2, \cdots$ do
5:     $u_k = \begin{cases} \textit{a uniformly random action in } U & \textit{with probability } \epsilon \\ u \in \arg\max_{\overline{u}} \{\mu_{\overline{u}}\} & \textit{with probability } 1-\epsilon \end{cases}$
6:     apply $u_k$
7:     measure reward $r_k$
8:     update $\mu_{u_k}$ {the mean reward of the selected action}
9: end for
```

Algorithm 2.2 ϵ-first strategy

```
Input: exploration-exploitation trade-off $\epsilon$
       horizon length $N$
1: for all action $u \in U$ do
2:     initialize expected reward $\mu_u$ for lever $u$ {ex. minimum possible reward}
3: end for
4: for $k = 1$ to $\epsilon N$ do
5:     $u_k$ = a uniformly random action in $U$
6:     apply $u_k$
7:     measure reward $r_k$
8:     update $\mu_{u_k}$
9: end for
10: for $k = \epsilon N + 1$ to $N$ do
11:     $u_k = \arg\max_{\overline{u}} \{\mu_{\overline{u}}\}$
12:     apply $u_k$
13:     measure reward $r_k$
14:     update $\mu_{u_k}$
15: end for
```

of the times that each action has been selected. Let us call this counter as n_u for each action $u \in U$. Auer *et al.* (2002) propose to select the action to play at time step k greedily as:

$$u_k = \arg\max_{\bar{u}} \left\{ \mu_{\bar{u}} + \sqrt{\frac{2 \ln k}{n_{\bar{u}}}} \right\}$$

this results in a regret bounded by:

$$8 \sum_{\forall \bar{u} \neq u^*} \frac{\ln k}{\mu_{u^*} - \mu_{\bar{u}}} + \left(1 + \frac{\pi^2}{3}\right) \sum_{\bar{u}} \mu_{u^*} - \mu_{\bar{u}}$$

where u^* corresponds to the optimal action. Notice that the action is selected based on the current average payoff plus an exploration term. This exploration term is the square root of the division of the logarithm of the current time step by the times the action has been played. The division of the current time by the times that the action is been played increases as the action is less often visited so it is an exploration bonus for the actions that has not been explored recently. This exploration bonus asymptotically converges to zero over time due to the logarithm and the square root providing the algorithm with much exploration at the beginning and reducing exploration over time.

Probability matching strategies reflect the idea that the number of pulls for a given lever should match its actual probability of being the optimal. As an example of probability matching strategies we will discuss Learning Automata in section 2.1.1.

2.1.1 Learning Automata

A learning automaton (LA) is a simple model for adaptive decision making in unknown random environments. Engineering research on reinforcement learning (RL) started in the early 1960's (Tsetlin, 1962). The RL model is well presented by the following example introduced by Thathachar & Sastry (2004). Consider a student and a teacher. The student is posed a question and is given several alternative answers. The student can pick one of the alternatives, following which the teacher responds yes or no. This response is probabilistic – the teacher may say yes for wrong alternatives and vice versa. The student is expected to learn the correct alternative via such repeated interactions. While the problem is ill-posed with this generality, it becomes solvable with an added condition. It is assumed that the probability of the teacher saying "yes" is maximum for the correct alternative.

LA are useful in applications that involve optimization of a function which is not completely known in the sense that only noisy values of the function are observable for any specific values of arguments Thathachar & Sastry (2004).

A finite action learning automaton (FALA) keeps a probability distribution over the actions for selecting them over time. Let us call $p_k : U \to [0,1]$ the function representing the probability of selecting each action $u \in U$ at time k. The learning algorithm for updating $p_k(\cdot)$ is generally of the form shown in (2.2).

$$p_{k+1}(\cdot) = \mathscr{T}(p_k(\cdot), u_k, r_k) \tag{2.2}$$

A very well known scheme for updating the probability distribution is by reinforcing the probabilities of the actions that result in a favorable response or decreasing such probabilities otherwise. The general form is given by equation (2.3) where L is the number of possible actions, and parameters α and β are known as the learning rate and penalty respectively.

$$\begin{aligned} p_{k+1}(u_k) &= p_k(u_k) + \alpha r_k (1 - p_k(u_k)) - \beta(1 - r_k) p_k(u_k) \\ \forall_{u \neq u_k} : p_{k+1}(u) &= p_k(u) - \alpha r_k p_k(u) + \beta(1 - r_k)\left(\frac{1}{L-1} - p_k(u)\right) \end{aligned} \tag{2.3}$$

Three common update schemes are defined in literature (Hilgard & Bower, 1966):

- Linear Reward-Inaction (L_{R-I}) if $\beta = 0$
- Linear Reward-Penalty (L_{R-P}) if $\alpha = \beta$
- Linear Reward-ϵ Penalty ($L_{R-\epsilon P}$) if β is very small compared to α

Convergence to local optima is guaranteed with this method (Thathachar & Sastry, 2004). By using the L_{R-I} scheme the RL will converge to a pure strategy, meaning that one action will accumulate the total probability of one leaving the others with zero. The probability of converging to the action with the highest mean reward depends on the initial distribution $p_0(\cdot)$, the parameter α and the reward distribution for each action.

Most learning techniques control the exploration by some parameter like the ϵ in ϵ-greedy strategies. This make difficult to use this methods in some environments. The LA method build-in the exploration in the probability distribution kept for selecting the actions, so its exploration is not controlled by a given parameter but by the learning process itself. This is a key feature for selecting these decision makers among others.

In some applications there are several inputs that are to be handled independently from each other and communication is either not allowed or comes at a high price. Consider robotics, telecommunications and distributed control as examples. For example, a distributed control application assumes there exist several subsystems, each one being controlled by an independent controller preferably sensing only local information. As examples of applications we may refer to formation stabilization, supply chains, wind farms or irrigation channels. While each application has its own specificities, the general idea is that the subsystems with their basic controllers should be brought together in a kind of plug and play manner and still operate successfully. Since these subsystems might have complex interactions this is not trivial. The next section introduces the framework of games where multiple learners interact to achieve their individual goals.

2.2 Discrete games

When we have more than one learner influencing the environment and they all interact for achieving their individual goals we can describe the problem as a repeated normal form game from classical Game Theory. In this section we formally introduce some necessary concepts of game theory and in section 2.2.1 we describe an existing RL approach to coordinate exploration with limited communication in discrete action games.

We first discuss a formalism for presenting a game: the normal form (Gintis, 2009, Chapter 3).

The strategic form or normal form game consists of a number of players, a set of strategies for each of the players, and a payoff function that associates a payoff to each player with a choice of strategies by each player. More formally, an n-player normal form game consists of:

- A set of players $l = 1, \cdots, n$.
- A set S_l of strategy spaces for players $l = 1, \cdots, n$. We call $\boldsymbol{\pi} = (\pi_1, \cdots, \pi_n)$, where $\forall_l : \pi_l \in S_l$, a strategy profile for the game.
- A function $\rho_l : \boldsymbol{S} \rightarrow \mathbb{R}$ for each player $l = 1, \cdots, n$, where $\boldsymbol{S}$ is the space of strategy profiles so $\rho_l(\boldsymbol{\pi})$ is player l's pay-off when strategy profile $\boldsymbol{\pi}$ is chosen.

If all agents share the same pay-off function (i.e., $\forall_{i,j} : \rho_i(\cdot) = \rho_j(\cdot)$) then we refer to the game as a common interest game and we can simply represent the pay-off by a simple function

$$\begin{array}{c|cc} & u_{2,1} & u_{2,2} \\ \hline u_{1,1} & \rho(u_{1,1}, u_{2,1}) & \rho(u_{1,1}, u_{2,2}) \\ u_{1,2} & \rho(u_{1,2}, u_{2,1}) & \rho(u_{1,2}, u_{2,2}) \end{array}$$

Figure 2.1: A common interest 2-players 2-actions example game - The first index of the actions represents the player index. The second index of the actions represents the action corresponding to the current player. Since both players can choose one out of two actions, four possible joint actions exist and then the pay-off function is defined for each combination.

$\rho : \boldsymbol{S} \rightarrow \mathbb{R}$. A common way of representing the pay-off function is by means of a matrix. Consider a common interest game of players 1 and 2, with two possible actions each. There are four possible joint actions. Figure 2.1 illustrates this example. The rows correspond to the first player's actions while the columns to the second's. Each slot in the matrix represents the pay-off received by the players when choosing the corresponding combination.

The Nash equilibrium (named after John Forbes Nash) is a solution concept of a game, in which no player has anything to gain by changing only his own strategy unilaterally. If each player has chosen a strategy and no player can benefit by changing his strategy while the other players keep theirs unchanged, then the current set of strategies and the corresponding pay-offs constitute a Nash equilibrium. More formally, a strategy profile $\boldsymbol{\pi}^*$ is a Nash equilibrium (NE) if no unilateral deviation in strategy by any single player is profitable for that player, that is

$$\forall_{l,\pi_l \in S_l} : \rho_l(\boldsymbol{\pi}^*) \geq \rho_l\left(\left(\pi_1^*, \cdots, \pi_l, \cdots, \pi_n^*\right)\right)$$

Finally, a solution is Pareto optimal if no agent can be made better off without making at least one other agent worse off. The set of all Pareto optimal equilibria are referred as a Pareto front. Notice that in a common interest game, the Pareto front only contains Nash equilibria.

RL was originally developed for Markov decision processs (MDPs) (See section 3.1 for an explanation of MDP). It guarantees convergence to the optimal policy, provided that the agent can sufficiently experiment and the environment is which it is operating is Markovian. However when multiple agents apply RL in a shared environment, this might be beyond the MDP model. In such systems, the optimal policy of an agent depends not only on the environment, but on the policies of the other agents as well (Nowé *et al.*).

In common interest games at least one Pareto optimal Nash equilibrium exists. As explained above, despite the existence of an optimal solution, there is no guarantee that independent learners converge to such an equilibrium. Only convergence to (possibly suboptimal)

	u_{21}	u_{22}	u_{23}
u_{11}	11	-30	0
u_{12}	-30	7	6
u_{13}	0	0	5

Figure 2.2: The climbing game, a common interest game - The Pareto optimal Nash equilibrium (u_{11}, u_{21}) is surrounded by heavy penalties (Kapetanakis *et al.*, 2003a).

Nash equilibria can be guaranteed (Claus & Boutilier, 1998). Games such as the climbing game (Kapetanakis *et al.*, 2003a), shown in Figure 2.2 are generally accepted as hard coordination problems for independent learners. In this game, independent learners may get stuck in the suboptimal Nash equilibrium (u_{12},u_{22}) with payoff 7. Whether the agents reach the Pareto optimal Nash equilibrium with payoff 11 mainly depends on the initialization of the action probabilities when learning starts and the size of the penalties surrounding the optimum. Independent learning agents will have difficulties reaching the optimal solution because of the heavy penalties surrounding it.

The problem of avoiding local maxima and converging to the Pareto Optimal solution in games has been faced by diverse techniques.

Lenient learners (Panait *et al.*, 2006) perform an optimistic update to delete the effect of the big penalty surrounding the Pareto Optimal joint action. Independent learners update the probabilities of selecting the actions based on the average reward collected by each action. This, in addition with the explorative behavior of both agents playing the game, prevent the learners to converge to the joint action (u_{11}, u_{21}) for the average reward collected is severely affected by the surrounding penalties. Still, each learner is capable of observing the really high reward 11 for the joint action when they both play coordinated. Lenient learners perform an optimistic update in the sense that the learners collect a number of rewards (the amount of rewards is the leniency level) before they perform an update. For such an update the learners use the maximum reward instead of the average. This way, the effect of the penalties is attenuated allowing the learners to converge to the Pareto Optimal solution. The bigger the leniency level, the highest the chance to converge to the optimal solution, but also the slower the learning process.

The problem with lenient learners is that they are severely affected by noise in presence of non-deterministic rewards. To attenuate the effect of this noise, Frequency Maximum Q-Value (FMQ) (Kapetanakis & Kudenko, 2002) keeps track of the frequency that the maximum reward value has been observed for every action. Then instead of using the Q-values (The Q-values is a look-up table for keeping track of the quality of the actions, i.e., the expected reward for a given action. For more details see Section 3.1.1) directly for selecting the action, FMQ uses the summation of the Q-value and the maximum reward, times its frequency multiply by the weight controlling the importance of the FMQ heuristic. Notice that the Q-values are updated with a regular procedure but the effect of the big penalties surrounding the optimal joint-action is attenuated without being fully optimistic. This approach still fails in fully stochastic environments as shown by Kapetanakis *et al.* (2003b).

As a sequel of FMQ, the commitment frequencies protocol is presented by Kapetanakis *et al.* (2003b). A commitment sequence is a list of time slots in which the agents are committed to play the same action. This idea definitely makes the agents to coordinate on their actions avoiding helping them to avoid the big penalties surrounding the optimal solution. Even when the rewards are fully stochastic, the method performs well. The problem that still remains is that the agents are to learn coordinated. This constrain can be hard to met in a control application where different subsystems are performing independently from each other and cannot afford coordinating all the time but only at rarely specific times.

Next subsection introduces a coordination protocol for a set of LAs playing a game to ensure convergence to the Pareto Optimal solution. This coordination protocol is able to deal with stochastic rewards and do not ask for explicit coordination during learning. It only asks for very limited communication at very specific times.

2.2.1 Exploring Selfish RL

In this subsection we briefly explain exploring selfish reinforcement learning (ESRL) which was introduced for efficient exploration in discrete action spaces (Verbeeck, 2004). The main characteristic of ESRL is the way the agents coordinate their exploration. Phases in which agents act independently and behave as selfish optimizers (exploration phase) are alternated by phases in which agents are social and act so as to optimize the group objective (synchronization phase). During every exploration phase the agents each applying an RL technique will converge to a pure Nash equilibrium. In the synchronization phase the solution converged to at the end of the exploration phase is evaluated and removed from the joint-action space. In this way, new attractors can be explored. The removal is achieved by letting at least two agents exclude its private action that is part of the joint solution. Agents alternate between exploration and

	u_{21}	u_{23}
u_{11}	11	0
u_{13}	0	5

Figure 2.3: The climbing game after shrinking the joint-action space - The Pareto optimal Nash equilibrium (u_{11}, u_{21}) is still not the only one Nash equilibrium but it is now the most likely to find

synchronization phases to efficiently search the shrinking joint-action space in order to find the Pareto optimal Nash equilibrium.

If the Pareto optimal Nash equilibrium is reached, there is no room for improvement. However if this Pareto optimal Nash equilibrium is not reached, all agents will benefit from searching for it in the long run. Suppose for example that the agents of Figure 2.2 converged to joint action (u_{12},u_{22}) corresponding to a payoff of 7 for both agents. Independent learners will not have the tendency to explore further to reach the better Nash equilibrium, with payoff 11 for both agents, because of the high punishment of -30 that surrounds this interesting Nash equilibrium.

In order not to get too much punishment before discovering the best joint action (u_{11},u_{21}), the agents should coordinate their exploration. This means that the agents should decide together to search for a better Nash equilibrium. In ESRL this is achieved by making the agents exclude the action they converged to and then restarting learning in the remaining action subspace. In the case of the climbing game presented above, the joint-action space shrinks from 9 to 4 joint actions as shown in Figure 2.3. Again the learning itself can be performed in an independent manner, and this will drive the agents toward another Nash equilibrium. The average payoff received during this second period of learning can be compared with the average payoff received during the first period of learning, and as such the two Nash equilibria found can be compared against each other. This alternation between independent learning and excluding actions can be repeated until one agent has only one action left. This approach has been validated in much bigger games with more agents and more actions, as well as stochastic payoffs. Moreover the approach has also shown to beneficial in cases where agents take actions asynchronously and delayed rewards, such as for example in job scheduling settings, where agents take actions when a job is generated and jobs first enter a queue before being processed .

2.3 Continuous bandit problems

Although the first papers studied bandits with a finite number of arms, the research was soon extended into bandits with infinitely many for its significance in practical applications. Examples are learning the gain in a control system, pricing a new product with uncertain demand in order to maximize revenue, controlling the transmission power of a wireless communication system in a noisy channel to maximize the number of bits transmitted per unit of power, and calibrating the temperature or levels of other inputs to a reaction so as to maximize the yield of a chemical process (Cope, 2009). Several strategies for solving these problems also exist. In this section we briefly explain two well-known strategies and then show in more details, in section 2.3.3, an extension of RL to continuous action spaces.

2.3.1 Continuous Action Learning Automaton

We will first refer to the continuous action learning automaton (CALA) method introduced by Thathachar & Sastry (2004). Thathachar & Sastry (2004) formulated the objective of learning as an optimization of the mathematical performance index (2.4), where $\mathcal{R}(u, \Lambda)$ is the (possibly stochastic) reward function for a given model Λ, p is a parametric representation of the distribution over actions u, and U is the set of possible actions, assumed to be compact. The performance index J is the expectation of $\mathcal{R}$ with respect to the distribution p, and it therefore includes randomness in u and randomness in $\mathcal{R}$. The problem of maximizing J is analogous to the well-known *multi-armed bandit* problem, with a discrete or continuous arm set (see section 2.3).

$$J(\Lambda) = \int_U \mathcal{R}(u, \Lambda)\, \mathrm{d}p \tag{2.4}$$

Now the set of actions is a compact subset of $\mathbb{R}$. The learner keeps a probability distribution for selecting the actions which is a normal distribution with mean μ_k and standard deviation σ_k. We denote this normal distribution by $\mathcal{N}(\mu_k, \sigma_k)$. The CALA updates the parameters μ_k and σ_k at each instant k based on the reinforcement received from the environment. The final objective of the learning method is to make $\mathcal{N}(\mu_k, \sigma_k)$ converge to $\mathcal{N}(\mu^*, 0)$ where μ^* is the action maximizing the expected reward. In order to keep numerical stability, the standard deviation σ_k cannot converge to zero. Instead, this parameter will be lower bounded by σ_L.

The learning rule of CALA for the two parameters of the probability distribution are given in equations (2.5) and (2.6) respectively.

$$\mu_{k+1} = \mu_k + \alpha \frac{r_k^{u_k} - r_k^{\mu_k}}{\max(\sigma_L, \sigma_k)} \frac{u_k - \mu_k}{\max(\sigma_L, \sigma_k)} \tag{2.5}$$

$$\sigma_{k+1} = \sigma_k + \alpha \frac{r_k^{u_k} - r_k^{\mu_k}}{\max(\sigma_L, \sigma_k)} \left[\left(\frac{u_k - \mu_k}{\max(\sigma_L, \sigma_k)} \right)^2 - 1 \right] - \alpha K_+ (\sigma_k - \sigma_L) \tag{2.6}$$

where α is the learning rate controlling the step size, K_+ is a large positive constant, σ_L is a positive lower bound of σ, $r_k^{u_k}$ represents the reward obtained by applying the action u_k and $r_k^{\mu_k}$ represents the reward obtained by applying the action μ_k.

The idea behind the update scheme is as follows. If u_k gets better response from the environment than μ_k, then, μ_k is moved towards u_k; otherwise it is moved away. Division by $\max(\sigma_L, \sigma_k)$ is to standardize quantities. For updating the standard deviation, whenever an action that is away from the mean by more than one standard deviation results in better response from the environment or when an action choice within one standard deviation from the mean is worse, we increase the variance; otherwise we decrease it. The last term on the right hand side is a penalizing term for σ_k so as to push it towards σ_L. Algorithm 2.3 formalizes this learning technique.

Algorithm 2.3 CALA methods (Thathachar & Sastry, 2004, Section 1.6).

```
Input: learning rate $\alpha$
       standard deviation lower bound $\sigma_L$
       a large positive constant $K_+$
       a very low positive constant $\epsilon$
1: initialize $\mu_0$ and $\sigma_0$
2: repeat
3:   choose $u_k \sim \mathcal{N}(\mu_k, \sigma_k)$
4:   apply $u_k$
5:   measure rewards $r_k^{u_k}$ and $r_k^{\mu_k}$
6:   $\mu_{k+1} = \mu_k + \alpha \frac{r_k^{u_k} - r_k^{\mu_k}}{\max(\sigma_L, \sigma_k)} \frac{u_k - \mu_k}{\max(\sigma_L, \sigma_k)}$
7:   $\sigma_{k+1} = \sigma_k + \alpha \frac{r_k^{u_k} - r_k^{\mu_k}}{\max(\sigma_L, \sigma_k)} \left[ \left( \frac{u_k - \mu_k}{\max(\sigma_L, \sigma_k)} \right)^2 - 1 \right] - \alpha K_+ (\sigma_k - \sigma_L)$
8: until $|\mu_{k-1} - \mu_k| < \epsilon$ and $\sigma_k \leq \sigma_L$
```

While this method is very efficient in terms of calculations it turns out to be unfeasible from the sampling point of view. The CALA method needs to evaluate two actions at every instant, namely, $r_k^{u_k}$ and $r_k^{\mu_k}$. Gradient based methods (see section 4.3.2) have emerged based on the same idea that remove this problem but still they need smooth reward functions and are very sensitive to initialization in order to find the global maxima.

2.3.2 Hierarchical Optimistic Optimization

The second strategy that we introduce is the hierarchical optimistic optimization (HOO) (Bubeck *et al.*, 2011). The HOO strategy builds an estimate of the reward function by constructing a binary tree over the action space U. The nodes of the tree are associated with regions of the action space such that the regions associated with the nodes deeper in the tree represent increasingly smaller subset of U. The method stores some statistics at each node of the tree. In particular, it keeps track of the number of times a node has been traversed and the corresponding average of the rewards received so far. Based on these, HOO assigns an optimistic estimate (denoted by B) to the maximum mean-payoff associated with each node. The action is always selected greedily on the optimistic estimate of the payoff. Algorithm 2.4 describes this technique. The nodes are indexed by the depth $h \geq 0$ and its position within the level $1 \leq i \leq 2^h$.

This method assumes a smooth reward function and in addition it uses uses an expectation of how fast the mean pay-off changes around the maximum described by the parameters v_1 and ϱ. This information may not be fully available in practical applications. Furthermore, this method is really demanding from the memory point of view for storing new branches of the tree at every time that a new level is created. This means that as we explore more, much more memory is necessary for storing the binary tree containing the approximation of the reward function. This requirement is unfeasible for production machines with low memory capabilities. In addition, this representation of the reward function and its way to update it makes the method to adapt slowly to changes in non-stationary environments.

2.3.3 Continuous Action Reinforcement Learning Automata

In this section we explain the extended version of the LA technique for continuous action spaces. In the CALA model, explained above, the authors have chosen a specific parameterized density function, namely the Gaussian distribution. This made the calculations for updating the two parameters of the distribution simple but not every possible distribution over the action space can be represented, restricting the exploration of the action space of the learner. Furthermore, the CALA method relies on the gradient of the reward function for updating the

Algorithm 2.4 HOO method (Bubeck *et al.*, 2011).

```
Input: two real numbers $v_1 > 0$ and $\varrho \in (0,1)$ describing how fast the reward is changing
    around the maximum
1:  $tree = \{(0,1)\}$
2:  $B_{1,2} = B_{2,2} = +\infty$
3:  for $k = 1, 2, \cdots$ do
4:    $(i, h) \leftarrow (0,1)$ {Start at the root}
5:    $path \leftarrow (0,1)$ {path stores the path traverse in the tree}
6:    while $(h, i) \in tree$ do
7:      if $B_{h+1,2i-1} > B_{h+1,2i}$ then
8:        $(h, i) \leftarrow (h+1, 2i-1)$ {Select the more promising child}
9:      else if $B_{h+1,2i-1} < B_{h+1,2i}$ then
10:       $(h, i) \leftarrow (h+1, 2i)$
11:     else
12:       $Z \sim Ber(0.5)$ {Tie-breaking rule}
13:       $(h, i) \leftarrow (h+1, 2i+Z)$ {e.g., choose a child at random}
14:     end if
15:     $path \leftarrow path \bigcup \{(h, i)\}$
16:   end while
17:   $(H, I) \leftarrow (h, i)$ {The selected node}
18:   Choose an action $u_k$ in the region associated to $(U, I)$
19:   measure reward $r_k$
20:   $tree \leftarrow tree \bigcup \{(H, I)\}$ {Extend the tree}
21:   for all $(h, i) \in path$ do
22:     $c_{h,i} \leftarrow c_{h,i} + 1$ {Increment the counter of node}
23:     $\widehat{\mu}_{h,i} \leftarrow \left(1 - \frac{1}{c_{h,i}}\right)\widehat{\mu}_{h,i} + \frac{r_k}{c_{h,i}}$ {Update the mean of node}
24:   end for
25:   for all $(h, i) \in path$ do
26:     $U_{h,i} \leftarrow \widehat{\mu}_{h,i} + \sqrt{\frac{2\ln k}{c_{h,i}}} + v_1\varrho^h$ {Update the $U$ value of node}
27:   end for
28:   $B_{H+1,2I-1} \leftarrow +\infty$ {$B$-values of the children of the new leaf}
29:   $B_{H+1,2I} \leftarrow +\infty$
30:   $tree' \leftarrow tree$ {Local copy of the current tree}
31:   while $tree' \neq \{(0,1)\}$ do
32:     $(h, i) \leftarrow leaf(tree')$ {Take any remaining leaf}
33:     $B_{h,i} \leftarrow \min\{U_{h,i}, \max\{B_{h+1,2i-1}, B_{h+1,2i}\}\}$ {Backward computation}
34:     $tree' \leftarrow tree' \setminus \{(h, i)\}$ {Drop updated leaf}
35:   end while
36: end for
```

probability distribution making the method unappropriate for problems with non-differentiable reward functions.

Continuous action reinforcement learning automaton (CARLA) (Howell *et al.*, 1997) is an RL technique widely applicable to real-world problems (Rodríguez *et al.*, 2011) because it does not make any assumptions about the reward function. The sole assumption of the technique is that the action space is a compact subset of $\mathbb{R}$. Howell *et al.* (1997) represented the probability density function (PDF) f as a nonparametric function. Starting with the uniform distribution over the whole action space U and after exploring action $u_k \in U$ attime step k the PDF is updated as shown in (2.7). Parameters α and λ are referred to as learning and spreading rate respectively. The first one, α, determines the importance of new observations, the higher the rate the larger the change in the PDF from one time-step to another. The second parameter, λ, determines how much to spread the information of a given observation to the neighboring actions. Finally, η is a normalization factor so that $\int_{-\infty}^{+\infty} f_{k+1}(u)\,du = 1$.

$$f_{k+1}(u) = \begin{cases} \eta_k \left(f_k(u) + r_{k+1}(u_k)\,\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) & u \in U \\ 0 & u \notin U \end{cases} \tag{2.7}$$

Figure 2.4 shows a sketch of the CARLA action space exploration. At time-step $k = 0$ (Figure 2.4.a) the learner's strategy f is the uniform PDF and action u_0 is drawn at random. Then f is updated by adding a Gaussian bell around u_0 (red curve on top of the PDF in Figure 2.4.b). The amplitude is determined by the reward r_1 and the width by the spreading parameter λ. Then the PDF is normalized again as indicated in Figure 2.4.c by the bold line. From the new strategy f the action u_1 is drawn and the strategy is again updated by adding a Gaussian around the action as shown in 2.4.d and normalized as shown in 2.4.e. Again the resulting PDF is shown in bold. In this example both actions (u_0 and u_1 received positive rewards so the probabilities of the neighboring actions were positively reinforced). Notice that the idea remains the same as in the FALA method. We keep a probability distribution over the action space. According to this distribution, an action is selected and applied resulting in a reward of the environment. This reward is used to reinforce the selected action. The difference now is that since we are working with continuous actions we do not reinforce the selected action but but its whole neighborhood. The higher the reward the stronger the reinforcement leading to eventual convergence.

Algorithm 2.5 formalizes this method. Every time the agent selects an action, this action will be drawn stochastically according to the PDF. Therefore the cumulative density function

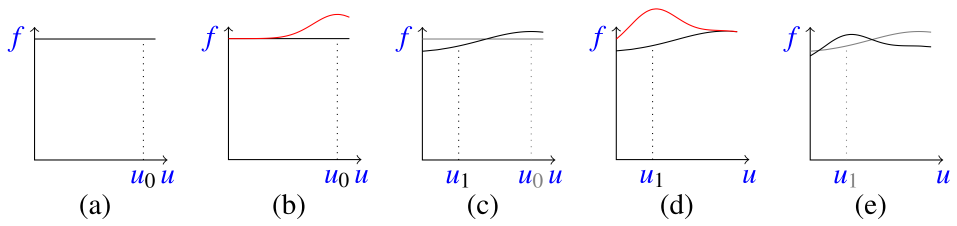


Figure 2.4: Sketch of CARLA action space exploration - The solid line represents the current PDF while the gray line represents the previous PDF. The red curve represents the PDF plus the reinforcement.

(CDF) is required (Parzen, 1960). Since the PDF used in the CARLA method is nonparametric, it is computationally very expensive to generate the CDF every time the function changes since a lot of numerical integration is needed. Additionally, the normalization factor η has to be calculated at every iteration also by numerical integration. This complexity makes this method unfeasible in applications where the response time is short and we cannot afford specialized hardware for the computations. On the positive side, the method has no limitations on the reward function though.

Algorithm 2.5 CARLA method (Howell *et al.*, 1997).

```
Input: learning rate $\alpha$
       spreading rate $\lambda$
1: initialize $f_0$, e.g., $f_0(u) \leftarrow \begin{cases} \frac{1}{\max U - \min U} & u \in U \\ 0 & u \notin U \end{cases}$
2: for all iteration $k = 0, 1, 2, \cdots$ do
3:     choose $u_k$ randomly drawn from $f_k(\cdot)$
4:     apply $u_k$
5:     measure reward $r_{k+1}$
6:     calculate normalization factor $\eta_k \leftarrow \frac{1}{\int_U \left( f_k(u) + r_{k+1}\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) du}$
7:     $f_{k+1}(u) \leftarrow \begin{cases} \eta_k \left( f_k(u) + r_{k+1}\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) & u \in U \\ 0 & u \notin U \end{cases}$
8: end for
```

This dissertation's contributions are centered on the CARLA method explained in this section. As a first feature, we do not have to care about the exploration-exploitation trade-off with this method since it is an LA technique and the exploration is built in the probability distribution for selecting the actions allowing for an easier usage. Secondly, independent learners may fail to find the global optimum in a game setting. Some techniques exists for a set of discrete action decision makers to converge to the optimal policy. Most of them have restrictions such as not being able to deal with stochastic pay-offs or constant coordination that are not possible in most distributed control applications. ESRL is a simple, but effective, coordination protocol based on discrete LAs for converging to optimal policies. Finally, among other LA techniques, CARLA is a gradient free algorithm that does not make any assumption on the reward function so it can be used in a wide range of applications.

2.4 Summary

In this chapter, we formalize the bandit problems in discrete and continuous action spaces. We describe a set of strategies for solving bandit applications with a discrete set of actions. Many applications, such as automatic control, are defined over a continuous action set. Three general techniques when facing continuous action sets are explained in this chapter. The first technique, CALA, is a gradient based method that is very efficient in terms of computation but it is unfeasible from the sampling point of view. Furthermore, it assumes smoothness on the reward landscape and it is very sensitive to noise. The second technique, HOO, is a gradient free method that explores the action space by building a binary tree and keeping an estimate of the average pay-off over this action space. In this case, there is no sampling drawback but still smoothness is required. Additionally, information about the shape of the reward is necessary which is most often not available in control applications. The third technique, CARLA, has no issues with the sampling or the assumptions on the reward but it is very costly from the calculation point of view for it keeps a nonparametric density function over the action space for selecting the actions so a lot of numerical integration is necessary. This is a real issue in practical applications where a response is required in a relatively short period of time and we may not add extra hardware due to the complexity of engines. In chapter 5 we show how this cost is significantly reduced by avoiding numerical integration making the method feasible for practical applications.

Additionally, the game setting was discussed. In a game setting the independent learners might converge to sub-optimal solutions. In this setting the exploration-exploitation dilemma becomes critic. Independent learners tend to learn as fast as possible, significantly reducing exploration over time. When two or more agents interact in the same environment for achieving a common or conflicting goal, they have to be more careful with their exploration for the received reward does not only depend on their individual action but on the other agent's policy as well. Section 2.2.1 explains a simple coordination protocol that helps totally independent learners to coordinate their exploration in a shared environment with very limited communication.

Chapter 3

Reinforcement learning

Learning from interactions with our environment is a straightforward idea to try when we think about learning. This is how a young child learns to catch a ball, just by trying it. Reinforcement learning (RL) is learning what to do (how to map situations to actions) so as to maximize a numerical reward signal. The learner is not told what to do in any situation as in most forms of machine learning (supervised learning), but instead has to discover which actions yield the most reward by trying them (Sutton & Barto, 1998, Chapter 1).

3.1 Markov decision processes

RL problems can be easily described using Markov decision process (MDP) (Puterman, 1994). We first introduce a simpler case of MDP, the deterministic case in section 3.1.1. Afterwards, we refer to a more general case, the stochastic case in section 3.1.2 (Buşoniu *et al.*, 2010, Chapter 2). Finally, we consider the RL problem and the most popular algorithms in the literature.

3.1.1 Deterministic settings

We describe an MDP as follows: We have a set of states X of a *process* and the set of actions U of the *controller*. Although in this chapter we refer to discrete X and U sets for RL methods, in general they can be a compact subset of $\mathbb{R}$ so we consider in this section the general case for the MDP. The process starts in the initial state of these states and moves, given the action of the controller, successively from one state to another. Each move is called a *step* or *transition*. If the process is at state x_k at time-step k and the controller picks the action u_k then the process transits to state x_{k+1} on time-step $k+1$. The state transitions are governed by the transition function $t : X \times U \to X$:

$$x_{k+1} = t(x_k, u_k)$$

At the same time, a scalar reward r_{k+1} is received by the controller. The scalar reward is determined by the reward function $\rho : X \times U \to \mathbb{R}$:

$$r_{k+1} = \rho(x_k, u_k)$$

The reward function evaluates the immediate performance of the controller but in general does not say anything about its long-term effect.

The actions of the controllers are selected following its policy $\pi : X \to U$:

$$u_k = \pi(x_k)$$

The Markov property in this framework is that given the fact that the *process* is at state x_k, and the controller selects the action u_k is enough to determine the transition to the new state x_k.

Some MDPs have terminal states that, once reached, can no longer be left. In such an state all the rewards received are 0. The RL literature often uses "trials" or "episodes" to refer to trajectories starting from some initial state and ending in a terminal state.

The goal when using RL is to maximize the long-run rewards collected from any initial state x_0. The aggregation of the rewards from the initial state is normally used. Several ways of aggregation are reported in the literature (Bertsekas & Tsitsiklis, 1996; Kaelbling *et al.*, 1996). The infinite-horizon discounted return is given by equation (3.1) where $\gamma \in [0, 1)$ is the *discount factor* and $x_{k+1} = t(x_k, \pi(x_k))$ for $k \geq 0$.

$$R^\pi(x_0) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \tag{3.1}$$

The discount factor is a measure of how "far-sighted" the controller is in considering its rewards, or a way of taking into account increasing uncertainty about future rewards. From a mathematical point of view, discounting ensures that the return will always be bounded if the rewards are bounded.

In this framework, the challenge is to maximize the long-term performance or return, while only using feedback about the immediate performance or reward. This leads to the so-called challenge of delayed rewards (Sutton & Barto, 1998). The actions taken in the present affects the future performance so the future feedback is also affected, the immediate reward gives the controller no direct information about these long-term effects though.

Other types of return can be defined. The finite-horizon discounted returns can be obtained by accumulating rewards along trajectories of a fixed, finite length K_u (the horizon). Equation (3.2) formalizes the finite-horizon discounted return.

$$\sum_{k=0}^{K_u} \gamma^k \rho\left(x_k, \pi\left(x_k\right)\right) \tag{3.2}$$

When $\gamma = 1$ the finite-horizon discounted return is transformed into the finite-horizon undiscounted return.

The policy of a controller is usually described by two functions: state-action value functions (Q-functions) and state value functions (V-functions).

The Q-function $Q^\pi : X \times U \rightarrow \mathbb{R}$ of a policy π gives the return obtained when starting from a given state, applying a given action, and following π thereafter (3.3).

$$Q^\pi\left(x, u\right) = \rho\left(x, u\right) + \gamma R^\pi\left(t\left(x, u\right)\right) \tag{3.3}$$

Equation (3.3) can be rewritten as a discounted sum (3.4) where $x_0 = x$ and $u_0 = u$.

$$Q^\pi\left(x, u\right) = \sum_{k=0}^{\infty} \gamma^k \rho\left(x_k, u_k\right) \tag{3.4}$$

Notice that equation (3.3) and (3.4) are related as shown in (3.5) where $x_0 = x$ and $u_0 = u$.

$$\begin{aligned} Q^\pi\left(x, u\right) &= \sum_{k=0}^{\infty} \gamma^k \rho\left(x_k, u_k\right) \\ &= \rho\left(x, u\right) + \sum_{k=1}^{\infty} \gamma^k \rho\left(x_k, u_k\right) \\ &= \rho\left(x, u\right) + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} \rho\left(x_k, \pi\left(x_k\right)\right) \\ &= \rho\left(x, u\right) + \gamma R^\pi\left(t\left(x, u\right)\right) \end{aligned} \tag{3.5}$$

The optimal Q-function is defined as the best Q-value in every state that can be obtained over all policies.

$$\forall_{x \in X} : Q^*\left(x, u\right) = \max_\pi Q^\pi\left(x, u\right) \tag{3.6}$$

Similarly, the optimal policy π^* is defined as the policy that, when played, maximizes the Q-function (3.7) for all states.

$$\forall_{x \in X} : \pi^* (x) = \arg\max_u Q^* (x, u) \tag{3.7}$$

The Bellman equation may be obtained from the second step at (3.5) as shown in (3.8).

$$\begin{aligned} Q^\pi (x, u) &= \rho (x, u) + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} \rho (x_k, u_k) \\ &= \rho (x, u) + \gamma \left[\rho (t (x, u), \pi (t (x, u))) + \sum_{k=2}^{\infty} \gamma^{k-2} \rho (x_k, \pi (x_k)) \right] \\ &= \rho (x, u) + \gamma Q^\pi (t (x, u), \pi (t (x, u))) \end{aligned} \tag{3.8}$$

The Bellman optimality equation characterizes Q^*, and states that the optimal value of action u taken in state x equals the sum of the immediate reward and the discounted optimal value obtained by the best action in the next state.

$$Q^* (x, u) = \rho (x, u) + \gamma \max_{u'} Q^* (t (x, u), u') \tag{3.9}$$

The V-function $V^\pi : X \to \mathbb{R}$ of a policy π is the return obtained by starting from a particular state and following π. This V-function can be computed from the Q-function of policy π as equation (3.10) shows.

$$V^\pi (x) = R^\pi (x) = Q^\pi (x, \pi (x)) \tag{3.10}$$

The optimal V-function is the best V-function that can be obtained by any policy, and can be computed from the optimal Q-function (3.11).

$$V^* (x) = \max_\pi V^\pi (x) = \max_u Q^* (x, u) \tag{3.11}$$

The optimal policy π^* can be calculated from V^* given equation (3.12).

$$\pi^* (x) \in \arg\max_u \left[\rho (x, u) + \gamma V^* (t (x, u))\right] \tag{3.12}$$

3.1.2 Stochastic settings

We now consider stochastic MDPs. The main difference with the deterministic MDPs is that the transition function over the states does not occur deterministically. More specifically, the deterministic transition function t is replaced by the transition probability density function $\tilde{t} : X \times U \times X \rightarrow [0, \infty)$. Let the process be at state x_k and the controller pick the action u_k, the probability that the state of the process at next step x_{k+1} belongs to a region $X_{k+1} \subseteq X$ is determined as equation (3.13) shows. Notice that $\tilde{t}\,(x, u, \cdot)$ must be a valid density function of argument "$\cdot$" ,where the dot stands for the random variable x_{k+1}. The Markov property now is that the current state x_k and action u_k fully determine the probability density of the next state. We should not confuse the stochastic nature of an MDP with the stochastic nature of the controller. The policy of the controller can be stochastic in the sense that π generates actions with a given distribution.

$$P\left(x_k + 1 \in X_{k+1} | x_k, u_k\right) = \int\limits_{X_k+1} \tilde{t}\left(x_k, u_k, x'\right) dx' \tag{3.13}$$

Since the next state is stochastic now, the reward function must depend not only on the current state and action but also in the next state as well. Let $\tilde{\rho} : X \times U \times X \rightarrow \mathbb{R}$ be the reward function in the stochastic case as shown in equation (3.14). Notice that we are assuming that the reward is deterministic given the transition (x_k, u_k, x_{k+1}). If the reward is stochastic given a particular transition, we refer by $\tilde{\rho}$ to the expected reward.

$$r_{k+1} = \tilde{\rho}\left(x_k, u_k, x_{k+1}\right) \tag{3.14}$$

The expected infinite-horizon discounted return of an initial state x_0 under a (deterministic) policy π is denoted by (3.15). The operator E denotes the expected value and the notation $x_{k+1} \sim \tilde{t}\left(x_k, \pi\left(x_k\right), \cdot\right)$ means that the random variable x_{k+1} is drawn from the distribution $\tilde{t}\left(x_k, \pi\left(x_k\right), \cdot\right)$.

$$\begin{aligned} R^{\pi}\left(x_0\right) &= \lim_{K_u \rightarrow \infty} E_{x_{k+1} \sim \tilde{t}(x_k, \pi(x_k), \cdot)} \left\{ \sum_{k=0}^{K_u} \gamma^k r_{k+1} \right\} \\ &= \lim_{K_u \rightarrow \infty} E_{x_{k+1} \sim \tilde{t}(x_k, \pi(x_k), \cdot)} \left\{ \sum_{k=0}^{K_u} \gamma^k \tilde{\rho}\left(x_k, \pi\left(x_k\right), x_{k+1}\right) \right\} \end{aligned} \tag{3.15}$$

Similarly, the Q-function and V-function can be obtained for the stochastic case. The Q-function is the expected return under the stochastic transitions when starting at a particular state, taking a particular action, and following the policy π thereafter. Equation (3.16) shows the Q-function. The optimal Q-function and the optimal policy from optimal Q-function remain unchanged.

$$Q^{\pi}(x,u) = E_{x'\sim\tilde{t}(x,u,\cdot)}\left\{\tilde{\rho}(x,u,x') + \gamma R^{\pi}(x')\right\} \tag{3.16}$$

The Bellman equations for Q^{π} (3.17) and Q^{*} (3.18) are given in terms of expectations over the one-step stochastic transitions.

$$Q^{\pi}(x,u) = E_{x'\sim\tilde{t}(x,u,\cdot)}\left\{\tilde{\rho}(x,u,x') + \gamma Q^{\pi}(x',\pi(x'))\right\} \tag{3.17}$$

$$Q^{*}(x,u) = E_{x'\sim\tilde{t}(x,u,\cdot)}\left\{\tilde{\rho}(x,u,x') + \gamma \max_{u'} Q^{*}(x',u')\right\} \tag{3.18}$$

The definitions of the V-function V^{π} of a policy π, as well as of the optimal V-function V^{*}, are the same as for the deterministic case (3.10) and (3.11). Obtaining the optimal policy from V^{*} now involves expectation (3.19).

$$\pi^{*}(x) \in \arg\max_{u} E_{x'\sim\tilde{t}(x,u,\cdot)}\left\{\tilde{\rho}(x,u,x') + \gamma V^{*}(x')\right\} \tag{3.19}$$

The Bellman equations for V^{π} (3.20) and V^{*} (3.21) are obtained by considering the expectation over the one-step stochastic transition as well.

$$V^{\pi}(x) = E_{x'\sim\tilde{t}(x,\pi(x),\cdot)}\left\{\tilde{\rho}(x,\pi(x),x') + \gamma V^{\pi}(x')\right\} \tag{3.20}$$

$$V^{*}(x) = \max_{u} E_{x'\sim\tilde{t}(x,u,\cdot)}\left\{\tilde{\rho}(x,u,x') + \gamma V^{*}(x')\right\} \tag{3.21}$$

Most RL methods focuss around solving the Bellman optimality equations without having information about the model, i.e., without knowing the transition function, nor the reward function. The next section will introduce these methods.

3.2 Reinforcement learning

In RL a controller interacts with the process by means of three signals: a state signal describing the process, the action signal allowing the controller to influence the process, and the reward signal providing the controller with a feedback on its immediate performance. Figure 3.1

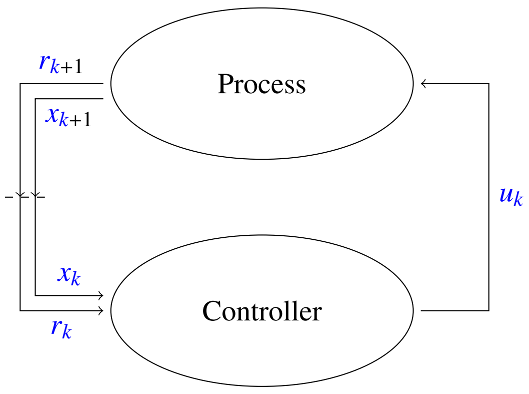


Figure 3.1: The controller-environment interaction in RL - The state x_k is sensed by the controller. It takes the action u_k so the process transits from x_k to x_{k+1} and feeds the controller back with the reward r_{k+1}. The state x_{k+1} turns into the next x_k for the controller at next time step (Sutton & Barto, 1998, Chapter 3).

shows the flow of interaction in RL. The controller senses the state x_k of the process and takes the action u_k making the process to transit from x_k to x_{k+1}. In response, the process feeds the controller back with the signal r_{k+1} (Buşoniu *et al.*, 2010, Chapter 2).

The behavior of the controller is dictated by its policy, a function from states into actions. It maps states of the environment into actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus-response rules or associations.

The feedback function defines the goal in an RL problem. It represents a mapping from each perceived state of the environment to a single number, a reward, indicating the desirability of that state. The goal of the controller is to maximize the return in the long run. Rewards might be interpreted as pleasure or pain in a biological system. The controller cannot alter this function but it is the basis for finding its policy (Sutton & Barto, 1998, Chapter 1).

The RL framework has been used in problems from a variety of fields, including, automatic control, operations research, economics, and games. Automatic control is one of the most important fields of origin for RL. In automatic control, RL can alternatively be seen as adaptive optimal control (Sutton *et al.*, 1992; Vrabie *et al.*, 2009).

RL algorithms are broken down in two main categories: value and policy iteration. Value iteration methods calculate the value function associated to the current or optimal policy. This value function is updated with every new observation while iterating with the process. Within value iteration, some methods use an explorative policy for collecting samples but use the optimal value function for updating. These methods are known as off-policy methods. In contrast, on-policy methods update their value function with the same policy they use for exploring the environment (Sutton & Barto, 1998). Storing the value function and then using some greedy strategy is not always efficient, specially if the action space is significantly large (see next chapter for examples). In such cases, some methods do not store the value function but a representation of the policy directly. These methods are referred to as policy iteration methods and work by iteratively evaluating and improving the current policy.

Within each of the two classes of RL algorithms, we can also differentiate offline and online algorithms. Offline RL algorithms use data collected in advance, whereas online RL algorithms learn a solution by directly interacting with the process. In this dissertation we focus in online learning.

3.3 Value iteration RL

Value iteration techniques iteratively compute an optimal value function using the Bellman optimality equation in order to derive an optimal policy. The most widely used algorithm of value iteration RL is Q-learning. Q-learning starts from an arbitrary initial Q-function Q_0 and updates using observed state transitions and rewards, i.e., data tuples of the form $(x_k, u_k, x_{k+1}, r_{k+1})$ (Watkins & Dayan, 1992; Watkins, 1989). After each transition, the Q-function is updated using such a data tuple, as shown in equation (3.22) where $\alpha_k \in (0, 1]$ is the learning rate. The term between square brackets is the temporal difference, i.e., the difference between the updated estimate $r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1}, u')$ of the optimal Q-value of (x_k, u_k) and the current estimate $Q_k(x_k, u_k)$.

$$Q_{k+1}(x_k, u_k) = Q_k(x_k, u_k) + \alpha_k \left[r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1}, u') - Q_k(x_k, u_k) \right] \tag{3.22}$$

As the number of transitions k approaches infinity, Q-learning asymptotically converges to Q^* if the state and action spaces are discrete and finite, and the following conditions (Jaakkola *et al.*, 1994; Tsitsiklis, 1994; Watkins & Dayan, 1992) hold:

- The sum $\sum_{k=0}^{\infty} \alpha_k^2$ produces a finite value, whereas the sum $\sum_{k=0}^{\infty} \alpha_k$ produces an infinite value.
- All the state-action pairs are (asymptotically) visited infinitely often.

The first condition is normally satisfied by setting $\alpha_k = \frac{1}{k}$. The second condition can equally be met by properly controlling the exploration-exploitation rate of the controller. It is enough if all actions are eligible by the controller in every state, i.e. the actions have a nonzero probability of being selected in every encountered state. The controller also has to exploit its current knowledge in order to obtain good performance, e.g., by selecting greedy actions in the current Q-function. This is a typical illustration of the exploration-exploitation (see chapter 2) trade-off in online RL. We consider two well-known strategies in RL for controlling the exploration: the ϵ-greedy exploration (Sutton & Barto, 1998, Section 2.2), and the Boltzmann exploration (Sutton & Barto, 1998, Section 2.3).

A classical way to balance exploration and exploitation in Q-learning is ϵ-greedy exploration which selects actions according to the equation (3.23) where $\epsilon_k \in (0, 1)$ is the exploration probability at step k.

$$u_k = \begin{cases} u \in \arg\max_{\overline{u}} Q_k(x_k, \overline{u}) & \textit{with probability } 1 - \epsilon_k \\ \textit{a uniformly random action in } U & \textit{with probability } \epsilon_k \end{cases} \tag{3.23}$$

A second option is to use Boltzmann exploration, which at step k selects an action u with the probability expressed by expression (3.24) where the temperature $\tau_k \geq 0$ controls the randomness of the exploration. As τ_k goes to infinity the exploration is more uniform and as τ_k goes down to zero the method is greedier. Usually the algorithms start with a high temperature, i.e., a more explorative behavior, and lower this temperature over time, so that the policy used asymptotically becomes greedy.

$$P(u|x_k) = \frac{e^{Q_k(x_k,u)/\tau_k}}{\sum_{\overline{u}} e^{Q_k(x_k,\overline{u})/\tau_k}} \tag{3.24}$$

Algorithm 3.1 describes the general Q-learning procedure.

A popular variation of Q-learning is SARSA (), an on-policy algorithm proposed by Rummery & Niranjan (1994). The name SARSA is obtained by joining together the initials of every element in the data tuples employed by the algorithm, namely: state, action, reward, (next) state, (next) action $(x_k, u_k, r_{k+1}, x_{k+1}, u_{k+1})$. SARSA starts with an arbitrary Q-function Q_0 and updates it at each step as shown in equation (3.25) where $\alpha \in (0, 1]$ is the learning rate. The term between square brackets is the temporal difference, obtained as the difference between the updated estimate $r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1})$ of the Q-value for (x_k, u_k), and the current estimate of $Q_k(x_k, u_k)$. This is not the same as the temporal difference used in Q-learning (3.22). Notice that while Q-learning uses the maximum of Q-value of the next state, SARSA uses the Q-value

Algorithm 3.1 Q-learning (Buşoniu *et al.*, 2010, Section 2.3).

```
Input: discount factor $\gamma$
       exploration schedule $\{\epsilon_k\}_{k=0}^{\infty}$
       learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$
1: initialize Q-functions, e.g., $Q_0 \leftarrow 0$
2: measure initial state $x_0$
3: for all time step $k = 0, 1, 2, \cdots$ do
4:     choose $u_k$ from $x_k$ using policy derived from $Q$
5:     apply $u_k$
6:     measure next state $x_{k+1}$
7:     measure reward $r_{k+1}$
8:     $Q_{k+1}(x_k, u_k) \leftarrow Q_k(x_k, u_k) + \alpha_k \left[r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1}, u') - Q_k(x_k, u_k)\right]$
9: end for
```

of the action taken in the next sate. This means that while Q-learning performs off-policy, SARSA performs on-policy.

$$Q_{k+1}(x_k, u_k) = Q_k(x_k, u_k) + \alpha_k \left[r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1}) - Q_k(x_k, u_k)\right] \tag{3.25}$$

Algorithm 3.2 formalizes SARSA learning procedure.

Algorithm 3.2 SARSA-learning (Buşoniu *et al.*, 2010, Section 2.4).

```
Input: discount factor $\gamma$
       exploration schedule $\{\epsilon_k\}_{k=0}^{\infty}$
       learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$
1: initialize Q-functions, e.g., $Q_0 \leftarrow 0$
2: measure initial state $x_0$
3: choose $u_0$ from $x_k$ using policy derived from $Q$
4: for all time step $k = 0, 1, 2, \cdots$ do
5:     apply $u_k$
6:     measure next state $x_{k+1}$
7:     measure reward $r_{k+1}$
8:     choose $u_k$ from $x_k$ using policy derived from $Q$
9:     $Q_{k+1}(x_k, u_k) \leftarrow Q_k(x_k, u_k) + \alpha_k \left[r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1}) - Q_k(x_k, u_k)\right]$
10: end for
```

3.4 Policy iteration RL

After discussing value iteration RL we now consider the policy iteration case. Policy iteration algorithms evaluate policies by constructing their value functions, and use these value functions to find new, improved policies (Bertsekas, 2007, Section 1.3). Algorithm 3.3 shows the general procedure in policy iteration.

Algorithm 3.3 Policy iteration.

```
 1: initialize policy $\pi_0$
 2: measure initial state $x_0$
 3: for all time step $k = 0, 1, 2, \cdots$ do
 4:     choose $u_k$ from $x_k$ using policy $\pi_k$
 5:     apply $u_k$
 6:     measure next state $x_{k+1}$
 7:     measure reward $r_{k+1}$
 8:     evaluate policy $\pi_k$ based on $x_k, u_k, r_{k+1}, x_{k+1}$
 9:     improve policy based on its evaluation
10: end for
```

We next discuss actor-critic methods as policy iterators. Actor-critic algorithms are on-policy methods that were introduced by Witten (1977) and have been investigated often since then (Barto *et al.*, 1983; Berenji & Khedkar, 1992; Berenji & Vengerov, 2003; Borkar, 2005; Konda & Tsitsiklis, 2003; Nakamura *et al.*, 2007; Sutton *et al.*, 2000). In an actor critic approach, a critic keeps track of the state value function while the actor represents the policy. The critic is responsible for evaluating the policy while the critic for improving it. Both rely on the temporal difference shown in equation (3.26). This temporal difference is analogous to the one for Q-learning but using the V-function instead of the Q-function.

$$\delta_{TD,k} = r_{k+1} + \gamma V(x_{k+1}) - V(x_k) \tag{3.26}$$

As an example of actor-critic algorithms consider the Interconnected Learning Automata (ILA) algorithm introduced by Wheeler Jr. & Narendra (1986) for the average return and by Witten (1977) for the discounted return. A study of the method and the convergence for multi-agent case is given by Vrancx (2010). The method uses a network of learning automaton (LA) for the actor. Each automaton is responsible for learning the best action at each state. As the process evolve from one state to the other, the control passes from the automaton associated to the current state to the one related to the next state. The automaton is not provided with the immediate reward. Instead, the critic is responsible for calculating the discounted return which is then feeded back to the decision makers.

3.5 Summary

In this chapter, two classes of RL techniques were introduced: value iteration, and policy iteration. The central challenge with these methods is that they are not possible to be implemented for problems with continuous action-state spaces. The main reason is that they need an exact representation of the value function of policies which is only possible for a finite set of actions and states. Most problems in automatic control have continuous state and actions. Even when the states and actions are finite, these sets are too large making the representation of the value functions or policies too expensive. Versions of the classical algorithms that approximately represent value functions and policies must be used for deleting this problem.

Asymptotically converging the optimal policy is not enough in practice. In online RL, the controller interacts with the process to improve its policy. It is important though, that the controller guarantee that it will never destabilize the process in an industrial application. In such a case, the exploration has to take care of this additional issue.

RL was created as a model-free technique. Still, it can also be significantly helped by domain knowledge if available. Encoding prior knowledge in the reward function should be done with care, because doing so incorrectly can lead to unexpected and possibly undesirable behavior (see e.g., Ng *et al.*, 1999).

Chapter 4

Reinforcement Learning in continuous action-state spaces

The Q-learning algorithm 3.1 and SARSA () algorithm 3.2 presented in Chapter 3 were originally designed for efficiently finding optimal control policies in Markov decision processs (MDPs) with finite, discrete state-action spaces. These discrete reinforcement learning (RL) formulations are not able to represent exact solutions of problems with large discrete or continuous action-state spaces, however. This has led to the development of function approximation-based RL methods, which enable RL algorithms to scale up to realistic continuous problem settings. Function approximation techniques have been widely applied in RL to represent policies and value functions in continuous action-state spaces. In online settings, RL methods are often combined with coarse coding approximations such as tile coding, radial basis functions or instance based learning (Degris *et al.*, 2012).

Representation is not the only problem in approximation in RL. Two additional types of approximation are needed. First, sample-based approximation is necessary in any RL algorithm. When the state-action space contains an infinite number of elements, it is impossible, in finite time, for any value iteration algorithm to sufficiently explore all the state-action pairs to estimate the Q-function or for any policy iteration method to evaluate the policy at every state. Second, value iteration and policy iteration must repeatedly solve maximization problems over the action variables, whereas policy search must find optimal policy parameters. In general, these optimization problems can only be solved approximately (Buşoniu *et al.*, 2010, Section 3.2).

In this Chapter we first explain parametric and non-parametric approximators in sections 4.1

and 4.2 respectively. Finally, we present some RL architectures for continuous spaces in section 4.3. The overview is based on the work presented by Buşoniu *et al.* (2010).

4.1 Parametric approximators

A parametric approximator used in an RL approach can be denoted by an approximation mapping $H : \mathbb{R}^n \to \mathcal{Q}$ where $\mathbb{R}^n$ is the parameter space and $\mathcal{Q}$ is the set of possible Q-functions. A compact representation of the Q-function is then obtained for every parameter vector θ (4.1). With this approximation is not sufficient a Q-value for every possible combination, only n parameters are.

$$\hat{Q}(x,u) = [H(\theta)](x,u) \tag{4.1}$$

Let us refer to one example of a linearly parameterized approximator introduced by Bertsekas & Tsitsiklis (1996). The approximator employs n basis functions (BFs) $\phi : X \times U \to \mathbb{R}$ and an n-dimensional parameter vector θ. A basis function is an element of a particular basis for a function space. Every continuous function in the function space can be represented as a linear combination of basis functions. The parameter vector θ refers intuitively to the Q-value associated with a each basis function. In this example the Q-values are computed as shown in equation (4.2).

$$[H(\theta)](x,u) = \phi^T(x,u)\,\theta \tag{4.2}$$

A complete example based on expression (4.2) is given by Buşoniu *et al.* (2010, Section 3.3). First, the action space is discretized into a small number of values $U_d = \{u_1, \cdots, u_M\} \in U$. Second, we need to define N state-dependent BFs $\bar{\phi}_1, \cdots, \bar{\phi}_N : X \to \mathbb{R}$ and replicate them for each discrete action in U_d. Approximate Q-values can be computed for every state-action pair in $X \times U_d$ with (4.3) where, in the state-action BF vector $\phi^T\left(x, u_j\right)$, all the BFs that do not correspond to the current discrete action are taken to be equal to 0 as (4.4).

$$[H(\theta)]\left(x,u_j\right) = \phi^T\left(x,u_j\right)\theta \tag{4.3}$$

$$\phi\left(x,u_j\right) = [\underbrace{0,\cdots,0}_{u_1},\cdots,\underbrace{\bar{\phi}_1(x),\cdots,\bar{\phi}_N(x)}_{u_j},\cdots,\underbrace{0,\cdots,0}_{u_M}]^T \in \mathbb{R}^{NM} \tag{4.4}$$

A second example of an approximator based on a set of BFs is the normalized Gaussian radial basis functions (RBFs). An RBF is a real-valued function whose value depends only on the distance from the origin, or alternatively on the distance from some other point called a center. The Gaussian type of RBF is defined in (4.5) where μ_i refers to the center of each Gaussian.

$$\bar{\phi}_i(x) = \frac{\phi'_i(x)}{\sum_{j=1}^{N} \phi'_j(x)} \quad \textit{where} \quad \phi'_i(x) = \exp\left(-\frac{1}{2}[x-\mu_i]^T B_i^{-1} [x-\mu_i]\right) \tag{4.5}$$

We next refer to tile coding (Watkins, 1989). In this method, the action-state space is divided in a set of N tilings, each covering the whole space. The tilings overlap, so that each point in the space will then be in exactly N tiles, one tile from each tiling. Tilings are displaced relative to each other so that no two tilings have their tile boundaries in the same place. This way, the space is finely partitioned into small regions. Each region corresponds to a set of active tiles, one for every tiling. We can think of the parameters as memory slots containing a scalar value and there is a given memory attached to every tile. Evaluating the Q-function in a given action-state pair is equivalent to calculating the arithmetic sum of the memory slots activated by the tilings. Figure 4.1 illustrate the method.

More formally, let $\psi : X \times U \rightarrow \{0, 1\}^N$ be a set of functions representing the tilings so $\psi_i(x, u)$ returns a binary vector with zeros in all positions but the one assigned to the tile activated by the pair (x, u) in the i-th tiling. The function approximator is then defined by the expression (4.6).

$$[H(\theta)](x, u) = \sum_{i=1}^{N} \psi_i(x, u)^T \theta \tag{4.6}$$

To conclude our description of parametric approximators, we stress that parametric approximators are not limited to linear approximators. Neural networks are probably the most popular non-linear approximators used in RL algorithms, e.g., $TD(\gamma)$ (Sutton, 1988).

4.2 Non-parametric approximators

Nonparametric approximators, despite their name, still have parameters. However, unlike the parametric case, the number of parameters, as well as the form of the nonparametric approximator, are derived from the available data.

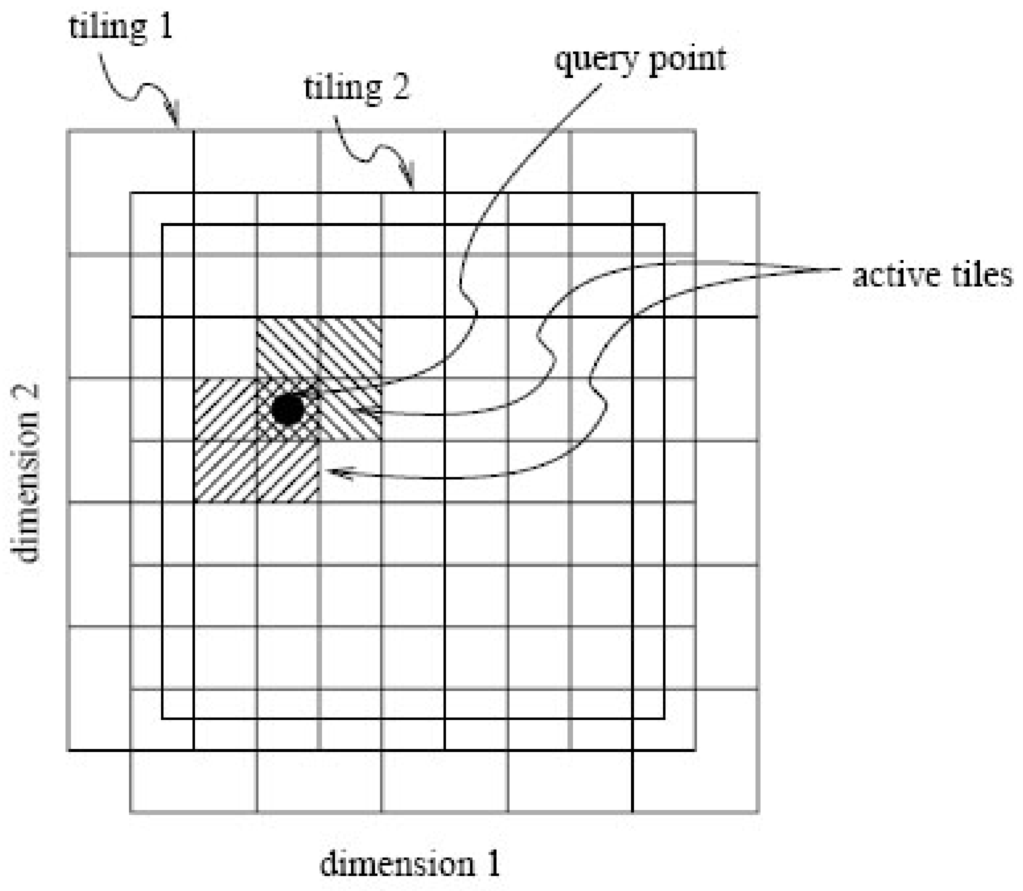


Figure 4.1: Tile coding approximator in 2D - Every state-action pair, represented by the dot, activates one tile in each tiling. The sum of the weights of all the activated tiles represents the vale associated with that state-action pair (Santamaria *et al.*, 1998).

Kernel-based approximators are typical representatives of the nonparametric class (Buşoniu *et al.*, 2010, Section 3.3). The kernel function is a function defined over two state-action pairs $\kappa : X \times U \times X \times U \to \mathbb{R}$ that must also satisfy certain additional conditions (Hofmann *et al.*, 2008).

A widely used type of kernel is the Gaussian kernel given by (4.7) where the kernel width matrix $\Sigma \in \mathbb{R}^{(D+C)\times(D+C)}$ must be symmetric and positive definite. Here, D denotes the number of state variables and C denotes the number of action variables. For instance, a diagonal matrix $\Sigma = diag\,(b_1, \cdots, b_{D+C})$ can be used. Note that the Gaussian kernel has the same shape as a Gaussian state-action RBF centered on (x', u').

$$\kappa\left((x,u),(x',u')\right) = \exp\left(-\frac{1}{2}\left[\begin{array}{c} x-x' \\ u-u' \end{array}\right]^T \Sigma^{-1}\left[\begin{array}{c} x-x' \\ u-u' \end{array}\right]\right) \tag{4.7}$$

Assume that a set of state-action samples $\{(x_{k_s}, u_{k_s})\,|k_s = 1, \cdots, n_s\}$ is available. For this set of samples, the kernel-based approximator takes the form (4.8). Notice that the sole difference between a kernel based approximator and the above linearly parameterized approximators is that in the later the BFs are predefined in advance, while in the former the function is based on the data collected. Using a set of predefined BFs we may construct a perfectly square mesh while by collecting data it will not be perfectly square. No real differences are expected from the interpolation point of view though.

$$[H(\theta)](x,u) = \sum_{k_s=1}^{n_s} \kappa\left((x,u),(x_{k_s},u_{k_s})\right)\theta_{k_s} \tag{4.8}$$

Many types of nonparametric approximators have been designed. Important classes include kernel-based methods (Shawe-Taylor & Cristianini, 2004), among which support vector machines are probably the most popular (Cristianini & Shawe-Taylor, 2000; Schölkopf & Smola, 1999; Smola & Scholkopf, 2004), Gaussian processes, which also employ kernels (Rasmussen & Williams, 2006), and regression trees (Breiman, 2001; Breiman *et al.*, 1984). Most of them have also been used in RL. For instance, kernel-based and related approximators have been applied to value iteration (Deisenroth *et al.*, 2009; Farahmand *et al.*, 2009a; Ormoneit & Sen, 2002) and to policy evaluation and policy iteration (Bethke *et al.*, 2008; Engel *et al.*, 2003, 2005; Jung & Polani, 2007; Lagoudakis & Parr, 2003; Xu *et al.*, 2007). Ensembles of regression trees have been used with value iteration by Ernst (2005); Ernst *et al.* (2006a) and with policy iteration by Jodogne *et al.* (2006).

4.3 RL architectures

Having introduced the function approximators above, we now present some popular architectures of RL. We first refer to approximate value iteration in section 4.3.1, then to approximate policy iteration 4.3.2.

4.3.1 Approximate value iteration

Online algorithms have been studied since the beginning of the nineties, e.g., by Horiuchi *et al.* (1996); Jouffe (1998); Lin (1992); Melo *et al.* (2008); Murphy (2005); Nowé (1992); Sherstov & Stone (2005); Singh *et al.* (1995); Szepesvári & Smart (2004); Tesauro (1992); Tuyls *et al.* (2002). A strong research thread in offline model-free value iteration emerged later (Antos *et al.*, 2008; Ernst, 2005; Ernst *et al.*, 2006b; Farahmand *et al.*, 2009b; Munos & Szepesvári, 2008; Ormoneit & Sen, 2002; Riedmiller, 2005; Szepesvári & Munos, 2005). In this dissertation we will focuss on online learning.

Approximate value iteration methods are extensions to the discrete value iteration methods. Let us center the analysis in the approximate Q-learning case. In the discrete case, the Q-value is updated using a temporal difference update (3.22) repeated here for an easier reading:

$$Q_{k+1}(x_k, u_k) = Q_k(x_k, u_k) + \alpha_k \left[r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1}, u') - Q_k(x_k, u_k) \right]$$

In continuous spaces Q_k cannot be represented exactly so a compact representation is used by means of a parameter vector $\theta_k \in \mathbb{R}^n$, using a suitable approximation mapping $H : \mathbb{R}^n \to \mathcal{Q}$ as shown in equation (4.9). Notice that the dependence on the θ parameter is implicit.

$$\widehat{Q}_k = H(\theta_k) \tag{4.9}$$

So the update (3.22) can be written as:

$$Q_{k+1}(x_k, u_k) = \widehat{Q}_k(x_k, u_k) + \alpha_k \left[r_{k+1} + \gamma \max_{u'} \widehat{Q}_k(x_{k+1}, u') - \widehat{Q}_k(x_k, u_k) \right] \tag{4.10}$$

In (4.10), the term between square brackets gives the difference between the current Q-value and the currently observation, i.e., the current reward plus the the discount factor times the discounted optimal return from the next state on. Since there is no explicit representation of the Q-function, we lack the optimal Q-value as well. Instead, we provide the algorithm with an estimation. Let us call this estimation a Q-iteration mapping of the form:

$$[T(Q)](x_k, u_k) = r_{k+1} + \gamma \max_{u'} \widehat{Q}_k(x_{k+1}, u') \tag{4.11}$$

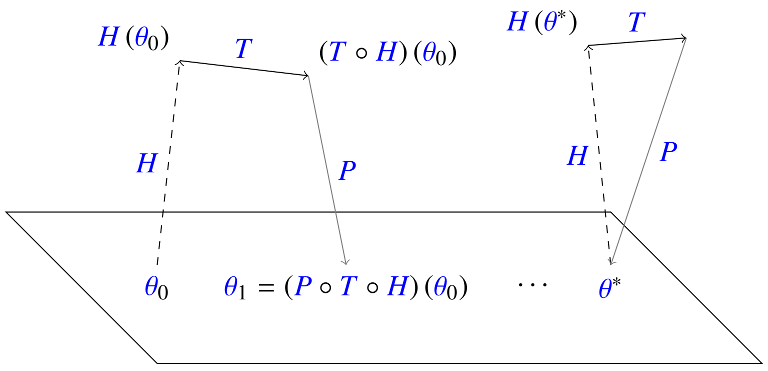


Figure 4.2: A conceptual illustration of approximate value iteration - The approximate Q-function is obtained by applying the approximation mapping H to the current parameter vector at every iteration, which is then passed through the iteration mapping T. Finally, the result of T is projected back by the projection mapping P. The algorithm will asymptotically converge to the optimal θ^*. The Q-function is obtained with $H(\theta^*)$ (Buşoniu *et al.*, 2010, Chapter 3).

In general, the newly found Q-function cannot be explicitly stored, either. Instead, it must also be represented approximately, using a new parameter vector θ_{k+1}. This parameter vector is obtained by a projection mapping $P : \mathcal{Q} \rightarrow \mathbb{R}^n$ which ensures that $\widehat{Q}_{k+1} = H(\theta_{k+1})$ is as close as possible to Q_{k+1}

To summarize, approximate value iteration methods start with an arbitrary (e.g., identically 0) parameter vector θ_0, and updates this vector at every iteration k using the composition of mappings P, T, and H as shows equation (4.12). Figure 4.2 illustrates this process.

$$\theta_{k+1} = (P \circ T \circ H)(\theta_k) \tag{4.12}$$

A straightforward way to obtain a good projection mapping is by using gradient descent. Next we explain how gradient-based Q-learning is obtained (Sutton & Barto, 1998, Chapter 8). The algorithm aims to minimize the squared error between the optimal value $Q^*(x_k, u_k)$ and the current Q-value as shows equation (4.13).

$$\begin{aligned}\theta_{k+1} &= \theta_k - \frac{1}{2}\alpha_k \frac{\partial}{\partial\theta_k}\left[Q^*(x_k, u_k) - \widehat{Q}_k(x_k, u_k)\right]^2 \\ &= \theta_k + \alpha_k\left[Q^*(x_k, u_k) - \widehat{Q}_k(x_k, u_k)\right]\frac{\partial}{\partial\theta_k}\widehat{Q}_k(x_k, u_k)\end{aligned} \tag{4.13}$$

Substituting the optimal Q-value $Q^*(x_k, u_k)$ by the Q-iteration mapping (4.11) we obtain:

$$\theta_{k+1} = \theta_k + \alpha_k\left[r_{k+1} + \gamma \max_{u'} \widehat{Q}_k(x_{k+1}, u') - \widehat{Q}_k(x_k, u_k)\right]\frac{\partial}{\partial\theta_k}\widehat{Q}_k(x_k, u_k) \tag{4.14}$$

Note that the term between brackets is an approximation of the temporal difference. Using the linearly parameterized approximator (4.2) we end up with the simplified approximate Q-learning update shown in expression (4.15). Algorithm 4.1 formalizes approximate Q-learning.

$$\theta_{k+1} = \theta_k + \alpha_k \left[r_{k+1} + \gamma \max_{u'} \left(\phi^T \left(x_{k+1}, u' \right) \theta_k \right) - \phi^T \left(x_k, u_k \right) \theta_k \right] \phi \left(x_k, u_k \right) \tag{4.15}$$

Algorithm 4.1 Approximate Q-learning with a linear parametrization (Buşoniu *et al.*, 2010, Chapter 3).

```
Input: discount factor $\gamma$
       BFs $\phi_1, \cdots, \phi_n : X \times U \to \mathbb{R}$
       exploration schedule $\{\epsilon_k\}_{k=0}^{\infty}$
       learning rate schedule$\{\alpha_k\}_{k=0}^{\infty}$
1: initialize parameter vector, e.g., $\theta_0 \leftarrow 0$
2: measure initial state $x_0$
3: for all iteration $k = 0, 1, 2, \cdots$ do
4:     choose $u_k$ from $x_k$ using policy derived from $\widehat{Q}_k$
5:     apply $u_k$
6:     measure next state $x_{k+1}$
7:     measure reward $r_{k+1}$
8:     $\theta_{k+1} \leftarrow \theta_k + \alpha_k \left[ r_{k+1} + \gamma \max_{u'} \left( \phi^T (x_{k+1}, u') \theta_k \right) - \phi^T (x_k, u_k) \theta_k \right] \phi (x_k, u_k)$
9: end for
```

We finish this section discussing a version of approximate SARSA 3.2. Algorithm 4.2 presents the approximate SARSA procedure. Approximate SARSA has been studied, e.g., by Gordon (2001); Melo *et al.* (2008); Santamaria *et al.* (1998); Sutton (1996). Notice that we are using a generic approximation mapping H. If we are to use a set of BFs we just have to use expression 4.2 so line 9 will finally be written as:

$$\theta_{k+1} \leftarrow \theta_k + \alpha_k \left[r_{k+1} + \gamma \phi^T \left(x_{k+1}, u_{k+1} \right) \theta_k - \phi^T \left(x_k, u_k \right) \theta_k \right] \phi \left(x_k, u_k \right)$$

4.3.2 Approximate policy iteration

Approximate policy search techniques represent the policy approximately, most often using a parametric approximation. Two major categories of approximate policy search might be identify: gradient-based and gradient-free policy search.

Algorithm 4.2 SARSA with a generic approximation mapping (Buşoniu *et al.*, 2010, Chapter 3).

```
Input: discount factor $\gamma$
       Approximation mapping $H : \mathbb{R}^n \rightarrow \mathcal{Q}$
       exploration schedule $\{\epsilon_k\}_{k=0}^{\infty}$
       learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$
 1: initialize parameter vector, e.g., $\theta_0 \leftarrow 0$
 2: measure initial state $x_0$
 3: choose $u_k$ from $x_0$ using policy derived from $[H(\theta_0)]$
 4: for all time step $k = 0, 1, 2, \cdots$ do
 5:     apply $u_k$
 6:     measure next state $x_{k+1}$
 7:     measure reward $r_{k+1}$
 8:     choose $u_k$ from $x_0$ using policy derived from $[H(\theta_k)]$
 9:     $\theta_{k+1} \leftarrow \theta_k + \alpha_k \left[r_{k+1} + \gamma [H(\theta_k)](x_{k+1}, u_{k+1}) - [H(\theta_k)](x_k, u_k)\right] \frac{\partial}{\partial \theta_k} [H(\theta_k)](x_k, u_k)$
10: end for
```

The idea of policy gradient algorithms is to update the policy with gradient ascent on the cumulative expected value V^{π} (Baxter & Bartlett, 2001; Peters *et al.*, 2003; Rückstieß *et al.*, 2010; Sutton *et al.*, 2000; Williams, 1992). The policy is usually represented by a differentiable parametrization and gradient updates are performed to find parameters that lead to maximal returns. The estimate of the gradient can be performed without explicitly using a value function (Marbach & Tsitsiklis, 2003; Munos, 2006; Riedmiller *et al.*, 2007) or approximating this value function.

In an actor critic approach, a critic $\widehat{V} : \mathbb{R}^n \rightarrow \mathcal{V}$ is a parameterized approximation mapping of the V-function where $\mathbb{R}^n$ is the parameter space and $\mathcal{V}$ is the set of all possible V-functions. So a compact representation of the V-function is obtained for every parameter vector θ. The actor $\widehat{\pi} : \mathbb{R}^m \rightarrow U$ outputs actions for each state where $\mathbb{R}^m$ is the parameter space. A compact representation of the policy is also obtained for every parameter vector ϑ. During the learning, the algorithm assumes exploration and uses the gradient of the V-function and the policy for updating the critic and actor parameters respectively. For such an update, an approximate temporal difference is used as shown in equation (4.16). This temporal difference is analogous to the one for Q-functions, used, e.g., in approximate SARSA 4.2.

$$\delta_{TD,k} = r_{k+1} + \gamma \widehat{V}(x_{k+1}; \theta_k) - \widehat{V}(x_k; \theta_k) \tag{4.16}$$

The actor and critic parameters are updated with the gradient formulas (4.17) and (4.18) respectively where $\alpha_{A,k}$ and $\alpha_{C,k}$ are the (possibly time-varying) step sizes for the actor and the critic, respectively.

$$\vartheta_{k+1} = \vartheta_k + \alpha_{A,k} \frac{\partial \widehat{\pi}(x_k; \vartheta)}{\partial \vartheta} \left[u_k - \widehat{\pi}(x_k; \vartheta)\right] \delta_{TD,k} \tag{4.17}$$

$$\theta_{k+1} = \theta_k + \alpha_{C,k} \frac{\partial \widehat{V}(x_k; \theta)}{\partial \theta} \delta_{TD,k} \tag{4.18}$$

The algorithm assumes exploration, meaning that the action predicted by the actor may differ from the current action u_k applied to the system. The temporal difference is interpreted by the actor as a correction in its prediction of the performance meaning that positive values of the temporal difference indicate a better predicted performance than the current one. That is the reason why, in the learning rule (4.17), when the exploratory action u_k leads to a positive temporal difference, the policy is adjusted towards this action or away from it otherwise. Similarly, the critic interprets the temporal difference as a prediction error leading to the update rule (4.18).

Algorithm 4.3 formalizes the actor-critic method using Gaussian exploration.

As a special case of an actor-critic method we consider the continuous actor-critic learning automaton (CACLA) algorithm (van Hasselt & Wiering, 2007, 2009). In contrast with most other actor-critic methods, CACLA uses an error in action space rather than in parameter of policy space and it uses the sign of the temporal-difference error rather than its size (van Hasselt, 2012).

In the CACLA algorithm, an update to the actor only occurs when the temporal-difference error is positive. Most other actor-critic methods use the size of the temporal-difference error and also update in the opposite direction when its sign is negative. However, in the CACLA algorithm that would mean updating towards an action that has not been selected and for which it is not known whether it is better than the current output of the actor. Algorithm 4.4 formalizes the method.

Recently, Vrabie *et al.* (2009) introduced the integral reinforcement learning (IRL), an actor critic algorithm for linear processes with a negative quadratic cost function. Using the

Algorithm 4.3 Actor-critic with Gaussian exploration (Buşoniu *et al.*, 2010, Chapter 3).

```
Input: discount factor $\gamma$
       policy parametrization $\widehat{\pi}$
       V-function parametrization $\widehat{V}$
       exploration schedule $\{\sigma_k\}_{k=0}^{\infty}$
       step size schedules $\{\alpha_{A,k}\}_{k=0}^{\infty}, \{\alpha_{C,k}\}_{k=0}^{\infty}$
 1: initialize parameter vector, e.g., $\vartheta_0 \leftarrow 0, \theta_0 \leftarrow 0$
 2: measure initial state $x_0$
 3: for all time step $k = 0, 1, 2, \cdots$ do
 4:     $u_k \leftarrow \widehat{\pi}(x_k; \vartheta_k) + \bar{u},\ \bar{u} \sim \mathcal{N}(0, \sigma_k)$
 5:     apply $u_k$
 6:     measure next state $x_{k+1}$
 7:     measure reward $r_{k+1}$
 8:     $\delta_{TD,k} \leftarrow r_{k+1} + \gamma \widehat{V}(x_{k+1}; \theta_k) - \widehat{V}(x_k; \theta_k)$
 9:     $\vartheta_{k+1} \leftarrow \vartheta_k + \alpha_{A,k} \frac{\partial \widehat{\pi}(x_k; \vartheta)}{\partial \vartheta} [u_k - \widehat{\pi}(x_k; \vartheta)] \delta_{TD,k}$
10:     $\theta_{k+1} \leftarrow \theta_k + \alpha_{C,k} \frac{\partial \widehat{V}(x_k; \theta)}{\partial \theta} \delta_{TD,k}$
11: end for
```

assumption of linearity, the controller converges fast to a good policy but such an assumption also restricts the application of the method. The algorithm uses an $n \times m$ gain matrix K to define the actor where n is the the dimension of the action space and m is the dimension of the state space. This gain matrix maps states into actions as follows: $u_k = K x_k$. The critic collects enough observations of actions and rewards to optimize the gain matrix by solving a least-square problem. Figure 4.3 shows the structure of the IRL method. The process is defined as a system with linear dynamics and the reward function as a negative quadratic cost. In the figure, T represents the sampling time of the critic and the dashed line the update of the actor. Properly selecting these frequencies is crucial for the performance since the algorithm needs enough data before solving the least-square problem.

The optimal gain matrix K is determined using the Bellman's optimality principle (Lewis & Syrmos, 1995):

$$K = -R^{-1} B^T P$$

where P is the solution of the Algebraic Riccati Equation (ARE):

$$A^T P + PA - PBR^{-1} B^T P + Q = 0$$

Algorithm 4.4 CACLA algorithm (van Hasselt, 2012).

```
Input: discount factor $\gamma$
       policy parametrization $\widehat{\pi}$
       V-function parametrization $\widehat{V}$
       exploration schedule $\{\sigma_k\}_{k=0}^{\infty}$
       step size schedules $\{\alpha_{A,k}\}_{k=0}^{\infty}, \{\alpha_{C,k}\}_{k=0}^{\infty}$
1:  initialize parameter vector, e.g., $\vartheta_0 \leftarrow 0, \theta_0 \leftarrow 0$
2:  measure initial state $x_0$
3:  for all time step $k = 0, 1, 2, \cdots$ do
4:      $u_k \leftarrow \widehat{\pi}(x_k; \vartheta_k) + \bar{u},\ \bar{u} \sim \mathcal{N}(0, \sigma_k)$
5:      apply $u_k$
6:      measure next state $x_{k+1}$
7:      measure reward $r_{k+1}$
8:      $\delta_{TD,k} \leftarrow r_{k+1} + \gamma \widehat{V}(x_{k+1}; \theta_k) - \widehat{V}(x_k; \theta_k)$
9:      if $\delta_{TD,k} > 0$ then
10:         $\vartheta_{k+1} \leftarrow \vartheta_k + \alpha_{A,k} \frac{\partial \widehat{\pi}(x_k;\vartheta)}{\partial \vartheta} [u_k - \widehat{\pi}(x_k; \vartheta)]$
11:     end if
12:     $\theta_{k+1} \leftarrow \theta_k + \alpha_{C,k} \frac{\partial \widehat{V}(x_k;\theta)}{\partial \theta} \delta_{TD,k}$
13: end for
```

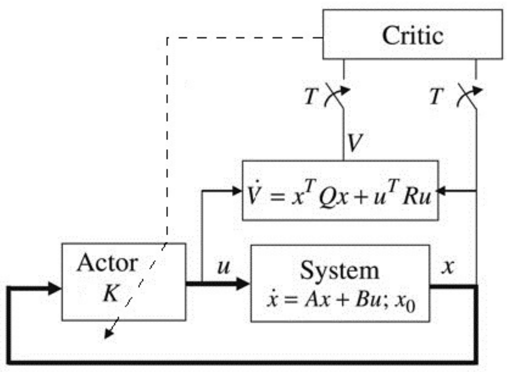


Figure 4.3: Integral Reinforcement Learning - Adaptive critic structure with IRL (Vrabie *et al.*, 2009).

P cannot be obtained without full information of the system. Instead, it is approximated using the critic outputs. Vrabie *et al.* (2009) introduced the algebraic derivations to update P based on two vectors: $\overline{x}$ and $\overline{p}$. Both vectors are calculated from the states and rewards collected over a trajectory. Vector $\overline{x}$ is the Kronecker product quadratic polynomial basis vector with elements $\left\{x_{k,i}x_{k,j}\right\}_{i=1,n;j=i,n}$, where n is the state dimension. The vector $\overline{p}$ is a representation of the matrix P, calculated by stacking the elements of the diagonal and upper triangular part of P in a column vector. Algorithm 4.5 shows the general procedure of the IRL method. Function kron returns $\overline{x}$ from the state transitions while function vec returns $\overline{p}$ from P. Function vec^{-1} represents the inverse procedure of vec (outputs P from $\overline{p}$).

Algorithm 4.5 IRL algorithm (Vrabie *et al.*, 2009).

```
Input: approximated solution of the ARE $P$
       partial information of the model $B$
       cost matrices $Q$ and $R$
       actor update frequency $T$
       target minimum error $\epsilon$
 1: initialize $\overline{p} = \text{vec}(P)$
 2: initialize $\overline{x}_0 = \text{kron}(x_0)$
 3: initialize gain matrix $K = -R^{-1}B^T P$
 4: measure initial state $x_0$
 5: for all time step $k = 0, 1, 2, \cdots$ do
 6:   for all $i = 1, \cdots, T$ do
 7:     $u_k = K x_k$
 8:     apply $u_k$
 9:     measure next state $x_{k+1}$
10:     measure reward $r_{k+1}$
11:     $\overline{x}_i = \text{kronpoly}(x_k)$
12:     store at the i-th column of $X$ the difference $\overline{x}_i - \overline{x}_{i-1}$
13:     store at the i-th element of $Y$ the difference $r_{k+1} - r_k$
14:   end for
15:   $\overline{p} = \left(X\,X^T\right)^{-1} X\,Y^T$
16:   if change($\overline{p}$) > $\epsilon$ then
17:     $P = \text{vec}^{-1}(\overline{p})$
18:     $K = -R^{-1}B^T P$
19:   end if
20: end for
```

The original implementation of IRL was introduced for single agent environments. A generalization to game settings was presented by Vrabie & Lewis (2010). The main idea remains the same but the least square problem involves now two controllers. Figure 4.4 shows the structure of this games settings. Notice that although the controllers may decide on their actions independently from each other, still they rely on global information. The critic corresponding to the first controller is able of observing the first controller actions as well as full state information and a similar situation holds for the second critic. Both actors selects the actions relying on full state information as well. Furthermore, although this method is presented as an alternative to game settings, the interactions between the controller are constrained to be linear restricting the applicability of the method. We will come back to these limitations in section 7.2.1 and also introduce the results obtained with the method presented in this dissertation.

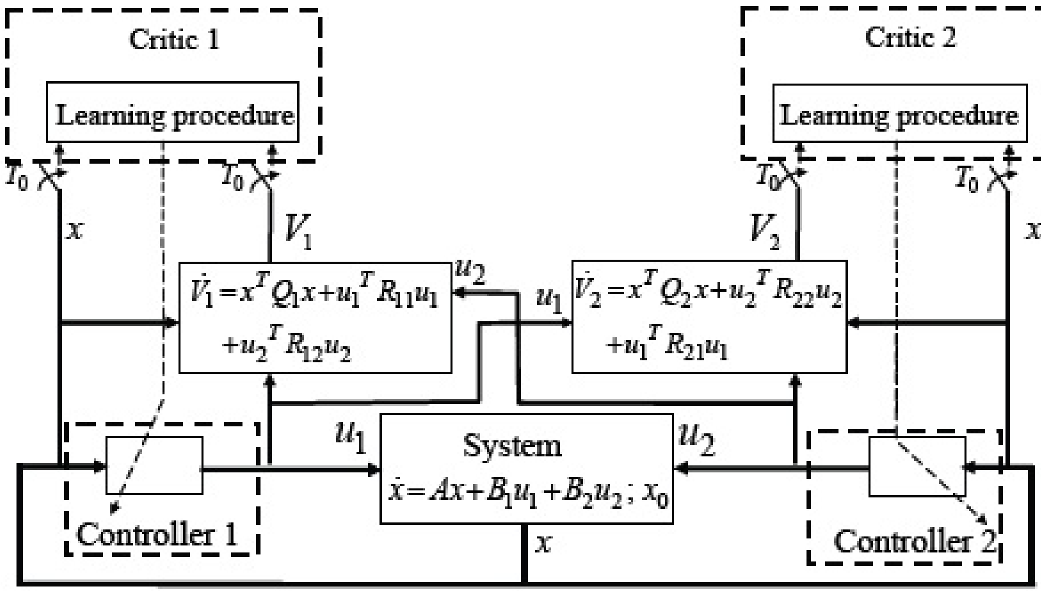

Critic 1
Learning procedure
Critic 2
Learning procedure
T_0
T_0
T_0
T_0
x
V_1
V_2
x
$\dot{V}_1 = x^T Q_1 x + u_1^T R_{11} u_1 + u_2^T R_{12} u_2$
u_2
u_1
$\dot{V}_2 = x^T Q_2 x + u_2^T R_{22} u_2 + u_1^T R_{21} u_1$
u_1
System
$\dot{x} = Ax + B_1 u_1 + B_2 u_2;\ x_0$
u_2
Controller 1
x
Controller 2

Figure 4.4: Multi-agent Integral Reinforcement Learning - Notice that the structure is similar to the one used in the single-agent IRL but now a second controller has been added (Vrabie & Lewis, 2010).

Both policy iteration techniques (CACLA and IRL) discussed in this section are gradient-based techniques. The main restriction of gradient-based policy search is that we have to be careful about the initialization of the parameters if we are to find the global optimum. To accomplish this goal, either prior knowledge about the optimal policy or about the form of the approximator is required. If multiple optima exist and the actor starts exploring far from the global optimum it can get trapped in some local maximum. Furthermore, if the approximator used is not adequate, the policy may be non-differentiable.

Gradient-free policy search avoids the above mentioned problems. The idea behind this type of algorithms is to find the optimal parameter vector ϑ such that the return $R^{\widehat{\pi}(\cdot;\vartheta)}(x)$ is maximized for all $x \in X$. A regular procedure to deal with continuous state spaces is to predefine a finite set X_0 of representative initial states and to estimate the returns only for the states in X_0. Then the score function is the weighted average return over these states (4.19) where $w : X_0 \to (0, 1]$ is the weight function (Buşoniu *et al.*, 2010, Chapter 3).

$$s(\vartheta) = \sum_{x_0 \in X_0} w(x_0)\, R^{\widehat{\pi}(\cdot;\vartheta)}(x_0) \tag{4.19}$$

4.4 Summary

In this chapter, we have introduced approximate RL for large or continuous-space problems. After explaining the need for approximation in such problems, parametric and nonparametric approximation architectures have been presented. Then, approximate versions for the two major categories of algorithms have been described: value iteration, and policy iteration.

Policy iteration methods were presented making the difference between gradient-based and gradient-free methods. The formers are normally faster but rely on knowledge regarding the optimal policy. The latter do not demand special requirements.

In all cases, the solutions expected from the methods are severely affected by the approximator used. In some cases finding a good policy approximation using prior knowledge may be easier than deriving a good approximation for value functions. In such a case, policy iteration methods may be preferred. Gradient-based algorithms have moderate computational demands and are often guaranteed to converge but only to local maxima depending on the approximator and may also converge slow. If no prior knowledge of the process is available, gradient-free policy iteration algorithms may be a better option.

Although approximate RL is active and rapidly expanding we might a face difficult task when considering high-dimensional problems with too few prior knowledge. In such cases it is really difficult to design a good parametrization that does not lead to excessive cost.

Chapter 5

CARLA extensions

Continuous action reinforcement learning automaton (CARLA) was introduced in section 2.3.3. This chapter describes an analysis of CARLA's performance. Based on this analysis, some changes are proposed for speeding up calculations and avoiding numerical integration in section 5.1, speeding up convergence by dynamically tuning the spreading rate in section 5.2 and the standardization of rewards in section 5.3 (Rodríguez *et al.*, 2011; Rodriguez *et al.*, 2011). Later on, the analysis of convergence is extended to games 5.4 and the extension of the exploring selfish reinforcement learning (ESRL) method, introduced in section 2.2.1, is explained in section 5.5 (Rodríguez *et al.*, 2012; Rodriguez *et al.*, 2012). Finally, in order to deal with Markov decision processs (MDPs), function approximators are used to extend the bandit solver CARLA into an reinforcement learning (RL) technique capable of deciding what action to take when sensing state information. This last update of the method is explained in section 5.6.

5.1 Speeding up calculations

We would like to recall the CARLA method. The authors implemented the policy π as a nonparametric probability density function (PDF) $f : U \to [0, +\infty)$. At every time step k the action u_k is drawn stochastically from the distribution f_k so $u_k \sim f_k(\cdot)$. Starting with the uniform distribution over the whole action space U and after exploring action $u_k \in U$ in time step k the PDF is updated as (5.1) shows. Remember that α is the learning rate determining the importance of newly observed rewards, λ is the spreading rate determining how much to spread the reward from one action into the neighboring actions, and η is a standardization factor.

$$f_{k+1}(u) = \begin{cases} \eta_k \left(f_k(u) + \rho(u_k)\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) & u \in U \\ 0 & u \notin U \end{cases} \tag{5.1}$$

This formulation (5.1) controls the strategy for the action selection of the automaton with a nonparametric PDF so it becomes computationally very expensive. The solution is to numerically approximate the function but still, some heavy numerical calculations are necessary for η_k. No convergence proof is given either. We next decrease the computational cost in terms of numerical calculations and provide a convergence proof in single agent environments in order to improve the method.

Let us start the analysis on expression (5.1) in order to look for possible reductions on the computational cost. Let $\min U$ and $\max U$ be the minimum and maximum possible actions. Since f is a PDF the restriction (5.2) has to be met at every time-step.

$$\begin{gathered} \int_{-\infty}^{+\infty} f_k(u)\,\mathrm{d}u = 1 \\ \int_{\min U}^{\max U} f_k(u)\,\mathrm{d}u = 1 \\ \int_{\min U}^{\max U} \eta_k \left(f_k(u) + \rho(u_k)\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) \mathrm{d}u = 1 \end{gathered} \tag{5.2}$$

Hence, the normalization factor η_k can be computed by expression (5.3).

$$\begin{aligned} \int_{\min U}^{\max U} \eta_k \left(f_k(u) + \rho(u_k)\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) \mathrm{d}u = 1 \\ \eta_k \int_{\min U}^{\max U} \left(f_k(u) + \alpha\rho(u_k) e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) \mathrm{d}u = 1 \\ \eta_k = \frac{1}{\int_{\min U}^{\max U} \left(f_k(u) + \rho(u_k)\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) \mathrm{d}u} \end{aligned} \tag{5.3}$$

The original formulation is for the bounded continuous action space U which makes the analytical calculation of the normalization factor η_k unlikely. Let us relax this constraint and work over the unbounded continuous action space $\mathbb{R}$. Then the PDF update rule introduced in (5.1) should be redefined as (5.4) where $f_{\mathcal{N}(u_k,\lambda)} : \mathbb{R} \to [0, +\infty)$ is the normal PDF with mean u_k and standard deviation λ. Analogously, (5.3) is transformed into (5.5). Notice that numerical integration is no longer needed for calculating η_k.

$$\begin{aligned} f_{k+1}(u) &= \eta_k \left(f_k(u) + \rho(u_k)\alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) \\ &= \eta_t \left(f_k(u) + \rho(u_k)\alpha\lambda\sqrt{2\pi} f_{\mathcal{N}(u_k,\lambda)}(u) \right) \end{aligned} \tag{5.4}$$

$$
\begin{aligned}
\eta_k &= \frac{1}{\int_{-\infty}^{+\infty} \left(f_k(u) + \rho(u_k)\,\alpha\lambda\sqrt{2\pi} f_{\mathcal{N}(u_k,\lambda)}\right) \mathrm{d}u} \\
&= \frac{1}{1 + \rho(u_k)\,\alpha\lambda\sqrt{2\pi}}
\end{aligned}
\tag{5.5}
$$

Let δ_k, introduced in (5.6), be the extra area added to the PDF by stretching the curve beyond the interval $[\min U, \max U]$ then (5.5), can be written as (5.7) and (5.4) as (5.8).

$$
\delta_k = \rho(u_k)\,\alpha\lambda\sqrt{2\pi} \tag{5.6}
$$

$$
\eta_k = \frac{1}{1 + \delta_k} \tag{5.7}
$$

$$
f_{k+1}(u) = \eta_k \left(f_k(u) + \delta_k f_{\mathcal{N}(u_k,\lambda)}(u)\right) \tag{5.8}
$$

In order to generate the actions following the policy f_k the cumulative density function (CDF) is needed Parzen (1960) which is introduced in (5.9).

$$
\begin{aligned}
F_{k+1}(u) &= \int_{-\infty}^{u} f_{k+1}(z)\,\mathrm{d}z \\
&= \int_{-\infty}^{u} \eta_k \left(f_k(z) + \delta_k f_{\mathcal{N}(u_k,\lambda)}(z)\right) \mathrm{d}z \\
&= \eta_k \left(F_k(u) + \delta_k F_{\mathcal{N}(u_k,\lambda)}(u)\right) \\
&= \eta_t \left(F_k(u) + \delta_k F_{\mathcal{N}(0,1)}\left(\frac{u - u_k}{\lambda}\right)\right)
\end{aligned}
\tag{5.9}
$$

Although there is no analytical definition for the normal CDF $F_{\mathcal{N}(\mu,\sigma)}$ it can be approximated by means of numerical integration. So still numerical integration is needed however one single CDF is required, being $F_{\mathcal{N}(0,1)}$ which can be calculated at the beginning of learning – only once – and there is no more need for integration during the learning process.

Finally, the original constraint has to be met for practical solutions, that is $\forall k : u_k \in U$ – see (5.1). So η_k and F_{k+1} defined in (5.7) and (5.9) should be transformed as shown in (5.10) and (5.11) where $F_k^{diff}(y_0, y_1) = F_{\mathcal{N}(0,1)}\left(\frac{y_1 - u_k}{\lambda}\right) - F_{\mathcal{N}(0,1)}\left(\frac{y_0 - u_k}{\lambda}\right)$.

$$
\eta_k = \frac{1}{1 + \delta_k F_k^{diff}(\min U, \max U)} \tag{5.10}
$$

$$F_{k+1}(u) = \begin{cases} 0 & u < \min U \\ \eta_k \left(F_k(u) + \delta_k F_k^{diff}(\min U, u_k) \right) & u \in U \\ 1 & u > \max U \end{cases} \tag{5.11}$$

For a practical implementation of this method, equations (5.6), (5.10) and (5.11) are sufficient to avoid numerical integration saving a lot of calculation time during the learning process. We would like to stress that without this reformulation the method was really computationally too heavy to be applied in processes where the controller only has a short period of time to respond, but with this change it turns to be computationally feasible. We perform an analysis of λ used in expression (5.6) in next section which will result in better convergence properties.

Algorithm 5.1 formalizes the method explained above. Notice the main idea of the algorithm remains the same as in algorithm 2.5 but the unnecessary numerical integration is avoided.

Algorithm 5.1 CARLA method with no unnecessary numerical integration.

```
Input: learning rate $\alpha$
       spreading rate $\lambda$
1: initialize $F_0$, e.g., $F_0(u) \leftarrow \begin{cases} \frac{u - \min U}{\max U - \min U} & u \in U \\ 0 & u \notin U \end{cases}$
2: for all iteration $k = 0, 1, 2, \cdots$ do
3:     choose $u_k \sim F_k(\cdot)$
4:     apply $u_k$
5:     measure reward $r_{k+1}$
6:     $\delta_k \leftarrow r_{k+1} \alpha \lambda \sqrt{2\pi}$ {equation (5.6)}
7:     $\eta_k \leftarrow \frac{1}{1 + \delta_k F_k^{diff}(\min U, \max U)}$ {equation (5.10)}
8:     $F_{k+1}(u) \leftarrow \begin{cases} 0 & u < \min U \\ \eta_k \left( F_k(u) + \delta_k F_k^{diff}(\min U, u_k) \right) & u \in U \\ 1 & u > \max U \end{cases}$ {equation (5.11)}
9: end for
```

5.2 Analysis of convergence. Dynamically tuning the spreading rate

The analysis will be performed for normalized reward signals $\rho : U \rightarrow [0, 1]$ – no generality is lost because any closed interval can mapped to this interval by a linear transformation. The final goal of this analysis is to find the necessary restrictions to guarantee convergence to local optima.

The sequence of PDF updates is a Markovian process, where for each time-step k an action $u_k \in U$ is selected and a new f_{k+1} is returned. At each time-step k, f_k will be updated as shown in expression (5.8). The expected value $\bar{f}_{k+1}$ (5.12) of f_{k+1} is computed given the current distribution of the actions.

$$\bar{f}_{k+1}(u) = \int_{-\infty}^{+\infty} f_k(z)\, f_{k+1}(u \mid u_k = z)\, \mathrm{d}z \tag{5.12}$$

Let $\eta_{k_z} = \eta_k | u_k = z$ be the value for η_k if $u_k = z$ and $\bar{\eta}_k$ the expected value of η_k then (5.12) could be rewritten as (5.13).

$$\begin{aligned}
\bar{f}_{k+1}(u) &= \int_{-\infty}^{+\infty} f_k(z)\, \eta_{k_z} \left(f_k(u) + \alpha\rho(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda}\right)^2} \right) \mathrm{d}z \\
&= f_k(u) \int_{-\infty}^{\infty} f_k(z)\, \eta_{k_z} \mathrm{d}z + \alpha \int_{-\infty}^{+\infty} f_k(z)\, \eta_{k_z} \rho(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda}\right)^2} \mathrm{d}z \\
&= f_k(u)\, \bar{\eta}_k + \alpha \int_{-\infty}^{+\infty} f_k(z)\, \eta_{k_z} \rho(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda}\right)^2} \mathrm{d}z
\end{aligned} \tag{5.13}$$

Let us have a look at the right member of the integral. $f_k(z)$ is multiplied by the factor composed by the normalization factor given that $u_k = z$, the feedback signal $\rho(z)$ and the distance measure $e^{-\frac{1}{2}\left(\frac{u-z}{\lambda}\right)^2}$ which can be interpreted as the strength of the relation of actions u and z, the higher the value of this product, the bigger the relation of these actions. Let us call this composed factor $G_k(u, z)$ and $\bar{G}_k(u)$ its expected value at time-step k with respect to z. Then equation (5.13) could be finally formulated as (5.14).

$$\begin{aligned}
\bar{f}_{k+1}(u) &= f_k(u)\, \bar{\eta}_k + \alpha \bar{G}_k(u) \\
&= f_k(u) \left(\bar{\eta}_k + \frac{\alpha \bar{G}_k(u)}{f_k(u)} \right)
\end{aligned} \tag{5.14}$$

The sign of the first derivative of f_k depends on the factor $\bar{\eta}_k + \frac{\alpha \bar{G}_k(u)}{f_k(u)}$ of expression (5.14) so it behaves as shown in (5.15).

$$\frac{\partial f_k}{\partial k}\begin{cases} < 0 & \left(\bar{\eta}_k + \frac{\alpha \bar{G}_k(u)}{f_k(u)}\right) < 1 \\ = 0 & \left(\bar{\eta}_k + \frac{\alpha \bar{G}_k(u)}{f_k(u)}\right) = 1 \\ > 0 & \left(\bar{\eta}_k + \frac{\alpha \bar{G}_k(u)}{f_k(u)}\right) > 1 \end{cases} \tag{5.15}$$

Notice $\bar{\eta}_k$ is a constant for all $u \in U$ and $\int_{-\infty}^{+\infty} f_k(z)\,\mathrm{d}z = 1$ so:

$$\exists_{b_1,b_2 \in U} : \frac{\bar{G}_k(b_1)}{f_k(b_1)} \neq \frac{\bar{G}_k(b_2)}{f_k(b_2)} \Longrightarrow \exists_{U^+,U^- \subset U, U^+ \cap U^- = \emptyset}, \forall_{u^+ \in U^+, u^- \in U^-} : \left(\frac{\partial f_k(u^+)}{\partial k} > 0\right) \wedge \left(\frac{\partial f_k(u^-)}{\partial k} < 0\right) \tag{5.16}$$

From logical implication (5.16) it can be assured that the sign of $\frac{\partial f_k(u)}{\partial k}$ will be determined by the ratio $\frac{\bar{G}_k(u)}{f_k(u)}$. Notice subsets U^+ and U^- are composed by the elements of U that have not reached their value for the probability density function in equilibrium with $\bar{G}_k(u)$. That is, the U^+ subset is composed by all $u \in U$ having a value of probability density function which is too small with respect to $\bar{G}_k(u)$ and vice versa for U^-.

Let $u^* \in U$ be the action that yields the highest value for $\int_{-\infty}^{+\infty} \rho(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda}\right)^2} \mathrm{d}z$ for all time-steps as shown in (5.17). It is important to stress that u^* is not the optimum of ρ but the point yielding the optimal vicinity around it and defined by $e^{-\frac{1}{2}\left(\frac{u^*-z}{\lambda}\right)^2}$ which depends on λ.

$$\forall_{u \in U,k} : \int_{-\infty}^{+\infty} \rho(z)\, e^{-\frac{1}{2}\left(\frac{u^*-z}{\lambda}\right)^2} \mathrm{d}z \geq \int_{-\infty}^{+\infty} \rho(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda}\right)^2} \mathrm{d}z \tag{5.17}$$

It is a fact that $\forall_{u \in U} : \bar{G}_k(u) \leq \bar{G}_k(u^*)$ and since the first derivative depends on $\frac{\bar{G}_k(u)}{f_k(u)}$, the value of $f_k(u^*)$ necessary for keeping $\frac{\partial f_k(u^*)}{\partial k} = 0$ is also higher than any other $f_k(u)$:

$$\forall_{u \in U} : f_k(u) \geq f_k(u^*) \Rightarrow \frac{\partial f_k(u)}{\partial k} < \frac{\partial f_k(u^*)}{\partial k} \tag{5.18}$$

Notice that the maximum update that $f_k(u^*)$ may receive is obtained when $u_k = u^*$ – centering the bell at u^* –, then if $f_k(u^*)$ reaches the value $\frac{1}{\lambda\sqrt{2\pi}}$, its first derivative will not be higher than 0 as shows (5.19) since $\rho : U \to [0, 1]$.

$$\begin{aligned} f_{k+1}(u^*) &= \eta_k \left(f_k(u) + \rho(u_k) \alpha e^{-\frac{1}{2}\left(\frac{u-u_k}{\lambda}\right)^2} \right) \\ &\leq \frac{1}{1+\alpha\lambda\sqrt{2\pi}} \left(\frac{1}{\lambda\sqrt{2\pi}} + \alpha \right) \\ &\leq \frac{1}{1+\alpha\lambda\sqrt{2\pi}} \left(\frac{1+\alpha\lambda\sqrt{2\pi}}{\lambda\sqrt{2\pi}} \right) \\ &\leq \frac{1}{\lambda\sqrt{2\pi}} \end{aligned} \tag{5.19}$$

Then the equilibrium point of $f_k(u^*)$ has the higher bound $\frac{1}{\lambda\sqrt{2\pi}}$. Notice that the closer the $\rho(u^*)$ to 1, the closer the equilibrium point of $f_k(u^*)$ to its higher bound.

$$\frac{\partial f_k(u^*)}{\partial k} \begin{cases} < 0 & f_k(u^*) > \frac{1}{\lambda\sqrt{2\pi}} \\ = 0 & f_k(u^*) = \frac{1}{\lambda\sqrt{2\pi}} \\ > 0 & f_k(u^*) < \frac{1}{\lambda\sqrt{2\pi}} \end{cases} \tag{5.20}$$

We can conclude from (5.18) and (5.37) that the highest value for f will be achieved at u^* as shown in (5.21) which has the higher bound $\frac{1}{\lambda\sqrt{2\pi}}$.

$$\begin{aligned} \forall_{u \in U \setminus \{u^*\}} : lim_{k\to\infty} f_k(u) &< f_k(u^*) \\ lim_{k\to\infty} f_k(u^*) &\leq \frac{1}{\lambda\sqrt{2\pi}} \end{aligned} \tag{5.21}$$

Finally

$$\begin{aligned} lim_{\lambda\downarrow 0} lim_{k\to\infty} f_k(u^*) &= \infty \\ \forall_{u\neq u^*} : lim_{\lambda\downarrow 0} lim_{k\to\infty} f_k(u) &= 0 \end{aligned} \tag{5.22}$$

This analysis has been developed under really restrictive assumptions, such as $k \to \infty$, $\lambda \downarrow 0$, α is small enough – the larger α, the larger the change of the probability law through time allowing a fast convergence but also the bigger the difference between the actual probability law and its expected value – and the reward function is noiseless enough to assure (5.17).

The best solution for the problem stated above about the constrains of λ is to start with a wide enough bell – allowing enough exploration – and make it thinner as it approaches the optimum – to meet (5.39). A good measure of convergence could be the standard deviation of the actions selected lately. When the standard deviation of actions is close to the λ that is been used to update the probability density function then the maximum value for $f_k(u^*)$ has been reached as stated in (5.21).

Since f_0 is the Uniform Density Function, the standard deviation of actions should start at $\sigma_0 = \frac{max(U)-min(U)}{\sqrt{12}}$. We are proposing to use expression (5.23) for the convergence ($conv_k$) value of the method given the standard deviation of actions σ_k. Then, (5.24) could be used as λ necessary in equation (5.6) for each time-step k improving the learning process of the automaton.

$$conv_k = 1 - \frac{\sqrt{12}\sigma_k}{\max(U) - \min(U)} \tag{5.23}$$

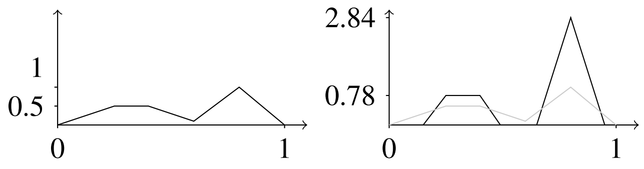
$$\lambda_k = \lambda \left(1 - conv_k\right) \tag{5.24}$$

Algorithm 5.2 formalizes the method explained above. Notice the main idea of the algorithm remains the same as in algorithm 5.1 but the parameter λ is automatically tuned over time.

Algorithm 5.2 CARLA method with automatic tuning of the spreading rate.

```
Input: learning rate $\alpha$
       initial spreading rate $\lambda$
 1: initialize $F_0$, e.g., $F_0(u) \leftarrow \begin{cases} \frac{u-\min U}{\max U - \min U} & u \in U \\ 0 & u \notin U \end{cases}$
 2: for all iteration $k = 0, 1, 2, \cdots$ do
 3:     choose $u_k \sim F_k(\cdot)$
 4:     apply $u_k$
 5:     measure reward $r_{k+1}$
 6:     $conv_k = 1 - \frac{\sqrt{12}\sigma_k}{\max(U)-\min(U)}$
 7:     $\lambda_k \leftarrow \lambda(1 - conv_k)$
 8:     $\delta_k \leftarrow r_{k+1}\alpha\lambda_k\sqrt{2\pi}$
 9:     $\eta_k \leftarrow \frac{1}{1+\delta_k F_k^{diff}(\min U, \max U)}$
10:     $F_{k+1}(u) \leftarrow \begin{cases} 0 & u < \min U \\ \eta_k\left(F_k(u) + \delta_k F_k^{diff}(\min U, u_k)\right) & u \in U \\ 1 & u > \max U \end{cases}$
11: end for
```

Figure 5.1: Standardization of rewards - Both charts plots the reward functions. X axis represents the action and Y axis the value of the function. We can see the original, untransformed reward function on the left. The black curve shows the reward function. On the right we plot both the original and transformed reward function in gray and black, respectively.



5.3 Standardization of rewards

In this section we introduce another modification of the CARLA algorithm. The basic idea is to improve the algorithm's convergence, by re-scaling the reward signal to amplify differences in the feedbacks. From the analysis introduced in section 5.2, it can be seen that larger differences in the feedbacks for different actions, lead to faster convergence to the optimum. If the reward function varies very little around the optimum, many time-steps may be required to detect the difference in feedback, and converge to the optimum.

If a learner memorizes the last few rewards, it can analyze their range, in order to linearly transform newly received rewards, viewing them as high or low signals relative to recently observed rewards. Since exploration is still possible, *z-score standardization* is very useful to deal with this re-scaling, avoiding sensitivity to outlier observations. Let μ_{rwd} and σ_{rwd} be the mean and standard deviation of the rewards collected, then every reward can be transformed as shown in expression (5.25). We stress that μ_{rwd} and σ_{rwd} should be calculated from untransformed rewards.

$$\rho'\left(u_k\right) = \max\left(0, 0.5 + \frac{\rho\left(u_k\right) - \mu_{rwd}}{2\sigma_{rwd}}\right) \tag{5.25}$$

Note that negative values are not allowed, in order to prevent the probability density function from becoming negative. Using this transformation, it is easier for the learner to differentiate rewards, no matter how close to the optimum it is exploring. Also observe that this is not a way to introduce extra information, it just modifies the reward function increasing the amplitude of the difference between the extreme values of this function. Scaling this amplitude does not imply changing the location of the maximum (or minimum) values.

Figure 5.1 shows an illustrative example of our method. On the left, we show the untransformed reward function, defined in equation 5.26. The PDF and the reward function are plotted in gray and black, respectively. These functions are introduced in (5.27) and (5.26) respectively. Note that it is very unlikely that the PDF reaches such a shape but for easy calculations for this example the triangular function was selected as a rough approximation of a Gaussian bell. For those values, the mean μ and standard deviation σ of the reward can be calculated as follows:

$$\mu = E_{u \sim f(\cdot)} \{\rho(u)\} = \int_{-\infty}^{+\infty} \rho(u) f(u) \, du$$

$$\sigma = \sqrt{E_{u \sim f(\cdot)} \left\{(\rho(u))^2\right\} - \left(E_{u \sim f(\cdot)} \{\rho(u)\}\right)^2} = \sqrt{\int_{-\infty}^{+\infty} (\rho(u))^2 f(u) \, du - \left(\int_{-\infty}^{+\infty} \rho(u) f(u) \, du\right)^2}$$

$$\rho(u) = \begin{cases} 2u & u \in [0, 0.25] \\ 0.5 & u \in (0.25, 0.4] \\ 0.5 - 2(u - 0.4) & u \in (0.4, 0.6] \\ 0.1 + 4.5(u - 0.6) & u \in (0.6, 0.8] \\ 1 - 4.5(u - 0.8) & u \in (0.8, 1] \end{cases} \tag{5.26}$$

$$f(u) = \begin{cases} 0 & u \in [0, 0.2] \\ 25(u - 0.2) & u \in (0.2, 0.4] \\ 25(0.6 - u) & u \in (0.4, 0.6] \\ 0 & u \in (0.6, 1] \end{cases} \tag{5.27}$$

A learner exploring around the local maximum on the left will have difficulty detecting the slightly better action to the right. For this example, we take $\mu_{rwd} = 0.43$ and $\sigma_{rwd} = 0.24$ so $\rho'(u) = \max\left(0, 0.5 + \frac{\rho(u) - 0.43}{0.24}\right)$. The figure on the right shows the effect of our transformation. The black curve represents ρ' while the gray curve corresponds to the original function. Note that the only change is in the amplitude of the function, the maxima are still located at the same points. Since the exploration was concentrated on the local maximum to the left, this is the zone that has been "zoomed in". Nonetheless, the other maximum situated to the right has also been amplified. Any exploratory actions to the right will now clearly detect the better feedback. Of course, in a real learning setting we could not simply transform the entire reward function off-line, and we would instead need to calculate the transformation online, based on the current sample mean and standard deviation.

Algorithm 5.3 formalizes the process explained above.

Algorithm 5.3 CARLA method with reward transformation.

```
Input: learning rate $\alpha$
       initial spreading rate $\lambda$
       reward's horizon length $K_r$
 1: initialize $F_0$, e.g., $F_0(u) \leftarrow \begin{cases} \frac{u - \min U}{\max U - \min U} & u \in U \\ 0 & u \notin U \end{cases}$
 2: for all iteration $k = 0, 1, 2, \cdots, K_r - 1$ do
 3:     choose $u_k \sim F_k(\cdot)$
 4:     apply $u_k$
 5:     measure reward $r_{k+1}$
 6:     $\boldsymbol{r}^{store} \leftarrow \boldsymbol{r}^{store} \bigcup r_{k+1}$
 7: end for
 8: $\mu_{rwd} \leftarrow mean(\boldsymbol{r}^{store})$
 9: $\sigma_{rwd} \leftarrow std\_dev(\boldsymbol{r}^{store})$
10: for all iteration $k = K_r, K_r + 1, K_r + 2, \cdots$ do
11:     choose $u_k \sim F_k(\cdot)$
12:     apply $u_k$
13:     measure reward $r_{k+1}$
14:     $\boldsymbol{r}^{store} \leftarrow \boldsymbol{r}^{store}_{[2 \cdot\cdot K_r]} \bigcup r_{k+1}$
15:     update $\mu_{rwd}$ and $\sigma_{rwd}$
16:     $r_{k+1} \leftarrow \max\left(0, 0.5 + \frac{r_{k+1} - \mu_{rwd}}{2\sigma_{rwd}}\right)$
17:     $conv_k \leftarrow 1 - \frac{\sqrt{12}\sigma_k}{\max(U) - \min(U)}$
18:     $\lambda_k \leftarrow \lambda(1 - conv_k)$
19:     $\delta_k \leftarrow r_{k+1}\alpha\lambda\sqrt{2\pi}$
20:     $\eta_k \leftarrow \frac{1}{1 + \delta_k F_k^{diff}(\min U, \max U)}$
21:     $F_{k+1}(u) \leftarrow \begin{cases} 0 & u < \min U \\ \eta_k\left(F_k(u) + \delta_k F_k^{diff}(\min U, u_k)\right) & u \in U \\ 1 & u > \max U \end{cases}$
22: end for
```

5.4 Analysis of convergence in games

The convergence proof for a single agent CARLA was introduced in 5.2. We now analyze the performance of the CARLA method in games. The analysis will be performed for normalized reward signals $\rho : U \to [0, 1]$ – no generality is lost because any closed interval can mapped to this interval by a linear transformation. The final goal of this analysis is to find the necessary restrictions to guarantee convergence to local optima.

Again, the sequence of PDF updates is analyze as a Markovian process, where for each time-step k an action $u_k^{(l)} \in U^{(l)}$ is selected and a new $f_k^{(l)}$ is returned for every learner l. The expected value $\bar{f}_{k+1}^{(l)}$ of $f_{k+1}^{(l)}$ can be computed following equation (5.28).

$$\bar{f}_{k+1}^{(l)}(u) = \int_{-\infty}^{+\infty} f_k^{(l)}(z)\, f_{k+1}^{(l)}\left(u \mid u_k^{(l)} = z\right) \mathrm{d}z \tag{5.28}$$

Let $\bar{\eta}_k^{(l)}(z) = \bar{\eta}_k^{(l)}|u_k = z$ be the expected value for $\eta_k^{(l)}$ if $u_k^{(l)} = z$ taking into account the other agent policies, $\bar{\bar{\eta}}_k^{(l)}$ its expected value and $\bar{\rho}_k^{(l)}(z)$ the expected value for $\rho_k^{(l)}(z)$ as shown in (5.29). Then (5.28) could be rewritten as (5.30).

$$\begin{aligned}
\bar{\eta}_k^{(l)}(z) = & \int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty} f_k^{(0)}\left(z^{(0)}\right) \cdots f_k^{(l-1)}\left(z^{(l-1)}\right) \\
& f_k^{(l+1)}\left(z^{(l+1)}\right) \cdots f_k^{(n)}\left(z^{(n)}\right) \\
& \left(\eta_k^{(l)}|u_k^{(l)} = z, u_k^{(0)} = z^{(0)}, \cdots u_k^{(n)} = z^{(n)}\right) \mathrm{d}z^{(n)} \cdots \mathrm{d}z^{(0)} \\
\bar{\rho}_k^{(l)}(z) = & \int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty} f_k^{(0)}\left(z^{(0)}\right) \cdots f_k^{(l-1)}\left(z^{(l-1)}\right) \\
& f_t^{(l+1)}\left(z^{(l+1)}\right) \cdots f_k^{(n)}\left(z^{(n)}\right) \\
& \rho_k^{(l)}\left(z^{(0)}, \cdots, z^{(l-1)}, z, z^{(l+1)}, \cdots, z^{(n)}\right) \mathrm{d}z^{(n)} \cdots \mathrm{d}z^{(0)} \\
\bar{\bar{\eta}}_k^{(l)} = & \int_{-\infty}^{+\infty} f_k^{(l)}(z)\, \bar{\eta}_k^{(l)}(z)\, \mathrm{d}z
\end{aligned} \tag{5.29}$$

$$\begin{aligned}\bar{f}_{k+1}^{(l)}(a) &= \int_{-\infty}^{+\infty} f_k^{(l)}(z)\,\bar{\eta}_k^{(l)}(z) \\ &\left(f_k^{(l)}(u) + \alpha^{(l)}\bar{\rho}_k^{(l)}(z)\,e^{-\frac{1}{2}\left(\frac{u-z}{\lambda_k^{(l)}}\right)^2}\right)\mathrm{d}z \\ &= f_k^{(l)}(u)\,\bar{\bar{\eta}}_k^{(l)} + \\ &\alpha^{(l)}\int_{-\infty}^{+\infty} f_k^{(l)}(z)\,\bar{\eta}_k^{(l)}(z)\,\bar{\rho}_k^{(l)}(z)\,e^{-\frac{1}{2}\left(\frac{u-z}{\lambda_k^{(l)}}\right)^2}\mathrm{d}z\end{aligned} \tag{5.30}$$

Let us have a look at the right member of the integral. $f_k^{(l)}(z)$ is multiplied by the factor composed by the normalization factor given that $u_k^{(l)} = z$, the feedback signal $\bar{\rho}_k^{(l)}(z)$ and the distance measure $e^{-\frac{1}{2}\left(\frac{u-z}{\lambda_k^{(l)}}\right)^2}$ which can be interpreted as the strength of the relation of actions u and z, the higher the value of this product, the bigger the relation of these actions. Let us call this composed factor $G_k^{(l)}(u,z)$ and $\bar{G}_k^{(l)}(u)$ its expected value at time-step k with respect to z. Then equation (5.30) could be finally formulated as (5.31).

$$\begin{aligned}\bar{f}_{k+1}^{(l)}(u) &= f_k^{(l)}(u)\,\bar{\bar{\eta}}_k^{(l)} + \alpha^{(l)}\bar{G}_k^{(l)}(u) \\ &= f_k^{(l)}(u)\left(\bar{\bar{\eta}}_k^{(l)} + \frac{\alpha^{(l)}\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)}\right)\end{aligned} \tag{5.31}$$

The sign of the first derivative of $f_k^{(l)}$ depends on the factor $\bar{\bar{\eta}}_k^{(l)} + \frac{\alpha^{(l)}\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)}$ of expression (5.31) so it behaves as shown in (5.32).

$$\frac{\partial f_k^{(l)}}{\partial k}\begin{cases} < 0 & \bar{\bar{\eta}}_k^{(l)} + \frac{\alpha^{(l)}\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)} < 1 \\ = 0 & \bar{\bar{\eta}}_k^{(l)} + \frac{\alpha^{(l)}\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)} = 1 \\ > 0 & \bar{\bar{\eta}}_k^{(l)} + \frac{\alpha^{(l)}\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)} > 1 \end{cases} \tag{5.32}$$

Notice $\bar{\bar{\eta}}_k^{(l)}$ is a constant for all $u \in U^{(l)}$ and $\int_{-\infty}^{+\infty} f_k^{(l)}(z)\,\mathrm{d}z = 1$ so:

$$\begin{gathered}\exists_{b_1^{(l)},b_2^{(l)}\in U^{(l)}} : \frac{\bar{G}_k^{(l)}\left(b_1^{(l)}\right)}{f_k^{(l)}\left(b_1^{(l)}\right)} \neq \frac{\bar{G}_k^{(l)}\left(b_2^{(l)}\right)}{f_k^{(l)}\left(b_2^{(l)}\right)} \Longrightarrow \\ \exists_{U_+^{(l)},U_-^{(l)}\subset U^{(l)},U_+^{(l)}\cap U_-^{(l)}=\emptyset}, \forall_{u_+^{(l)}\in U_+^{(l)},u_-^{(l)}\in U_-^{(l)}} : \\ \left(\frac{\partial f_k^{(l)}\left(u_+^{(l)}\right)}{\partial k} > 0\right) \wedge \left(\frac{\partial f_k^{(l)}\left(u_-^{(l)}\right)}{\partial k} < 0\right)\end{gathered} \tag{5.33}$$

From logical implication (5.33) it can be assured that the sign of $\frac{\partial f_k^{(l)}(u)}{\partial k}$ will be determined by the ratio $\frac{\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)}$. Notice subsets $U_+^{(l)}$ and $U_-^{(l)}$ are composed by the elements of $U^{(l)}$ that have not reached their value for the probability density function in equilibrium with $\bar{G}_k^{(l)}(u)$. That is, the $U_+^{(l)}$ subset is composed by all $u \in U^{(l)}$ having a value of probability density function which is too small with respect to $\bar{G}_k^{(l)}(u)$ and vice versa for $U_-^{(l)}$.

Let $u_*^{(l)} \in U^{(l)}$ be the actions yielding the highest value for every agent l at expression $\int_{-\infty}^{+\infty} \bar{\rho}_k^{(l)}(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda_k^{(l)}}\right)^2} dz$ as shown in (5.34). It is important to stress that $u_*^{(l)}$ are not the optimum of $\bar{\rho}_k^{(l)}$ given the set $f_k^{(j)}$ but the points yielding the optimal vicinity around it and defined by $e^{-\frac{1}{2}\left(\frac{u_*^{(l)}-z}{\lambda_k^{(l)}}\right)^2}$ which depends on $\lambda_k^{(l)}$.

$$\forall_{u \in U^{(l)}} : \int_{-\infty}^{+\infty} \bar{\rho}_k^{(l)}(z)\, e^{-\frac{1}{2}\left(\frac{u_*^{(l)}-z}{\lambda_k^{(l)}}\right)^2} dz \geq \int_{-\infty}^{+\infty} \bar{\rho}_k^{(l)}(z)\, e^{-\frac{1}{2}\left(\frac{u-z}{\lambda_k^{(l)}}\right)^2} dz \tag{5.34}$$

For common interest games ($\forall_{l,j} : \rho_k^{(l)} = \rho_k^{(j)}$), all $u_*^{(l)}$ will correspond initially to the same maximum in the joint-action space, if all learner start with the same parameters and an uniform PDF each one. Then (5.35) holds.

$$\begin{aligned} \forall_k : f_k^{(l)}\left(u_*^{(l)}\right) \geq f_k^{(l)}\left(u^{(l)}\right) &\Rightarrow f_{k+1}^{(l)}\left(u_*^{(l)}\right) \geq f_{k+1}^{(l)}\left(u^{(l)}\right) \\ \forall_k : \bar{\rho}_k^{(l)}\left(u_*^{(l)}\right) \geq \bar{\rho}_k^{(l)}\left(u^{(l)}\right) &\Rightarrow \bar{\rho}_{k+1}^{(l)}\left(u_*^{(l)}\right) \geq \bar{\rho}_{k+1}^{(l)}\left(u^{(l)}\right) \\ \forall_k : \bar{G}_k^{(l)}\left(u_*^{(l)}\right) \geq \bar{G}_k^{(l)}\left(u^{(l)}\right) &\Rightarrow \bar{G}_{k+1}^{(l)}\left(u_*^{(l)}\right) \geq \bar{G}_{k+1}^{(l)}\left(u^{(l)}\right) \end{aligned} \tag{5.35}$$

Since the first derivative of the PDF depends on $\frac{\bar{G}_k^{(l)}(u)}{f_k^{(l)}(u)}$ the value necessary for keeping $\frac{\partial f_k^{(l)}(u)}{\partial k} = 0$ is also the highest for every $u_*^{(l)}$.

Notice that the maximum update that $f_k^{(l)}\left(u_*^{(l)}\right)$ may receive is obtained when $u_k^{(l)} = u_*^{(l)}$ – centering the bell at $u_*^{(l)}$ –, then if $f_k^{(l)}\left(u_*^{(l)}\right)$ reaches the value $\frac{1}{\lambda_t^{(l)}\sqrt{2\pi}}$, its first derivative will not be higher than 0 as shows (5.36) since $\rho : U \rightarrow [0, 1]$.

$$\begin{aligned} f^{(l)}_{k+1}\left(u^{(l)}_*\right) = \bar{\eta}^{(l)}_k\left(u^{(l)}_k\right) \\ \left(f^{(l)}_*\left(u^{(l)}_k\right) + \rho^{(l)}_k\left(u^{(l)}_k\right)\alpha^{(l)}e^{-\frac{1}{2}\left(\frac{u^{(l)}_*-u^{(l)}_k}{\lambda^{(l)}_k}\right)^2}\right) \\ \leq \frac{1}{1+\alpha^{(l)}\lambda^{(l)}_k\sqrt{2\pi}}\left(\frac{1}{\lambda^{(l)}_k\sqrt{2\pi}} + \alpha^{(l)}\right) \\ \leq \frac{1}{1+\alpha^{(l)}\lambda^{(l)}_k\sqrt{2\pi}}\left(\frac{1+\alpha^{(l)}\lambda^{(l)}_k\sqrt{2\pi}}{\lambda^{(l)}_k\sqrt{2\pi}}\right) \\ \leq \frac{1}{\lambda^{(l)}_k\sqrt{2\pi}} \end{aligned} \tag{5.36}$$

Then the equilibrium point of $f^{(l)}_k\left(u^{(l)}_*\right)$ has the higher bound $\frac{1}{\lambda^{(l)}_k\sqrt{2\pi}}$. Notice that the closer the $\rho^{(l)}_k\left(u^{(l)}_*\right)$ to 1, the closer the equilibrium point of $f^{(l)}_k\left(u^{(l)}_*\right)$ to its higher bound.

$$\frac{\partial f^{(l)}_k\left(u^{(l)}_*\right)}{\partial k}\begin{cases} < 0 & f^{(l)}_k\left(u^{(l)}_*\right) > \frac{1}{\lambda^{(l)}_k\sqrt{2\pi}} \\ = 0 & f^{(l)}_k\left(u^{(l)}_*\right) = \frac{1}{\lambda^{(l)}_k\sqrt{2\pi}} \\ > 0 & f^{(l)}_k\left(u^{(l)}_*\right) < \frac{1}{\lambda^{(l)}_k\sqrt{2\pi}} \end{cases} \tag{5.37}$$

We can conclude from (5.37) that the highest value for $f^{(l)}$ will be achieved at $u^{(l)}_*$ as shown in (5.38) which has the higher bound $\frac{1}{\lambda^{(l)}_k\sqrt{2\pi}}$.

$$\begin{aligned} \forall_{u\in U^{(l)}\setminus\{u^{(l)}_*\}} : lim_{k\to\infty}f^{(l)}_k(u) < f^{(l)}_k\left(u^{(l)}_*\right) \\ lim_{k\to\infty}f^{(l)}_k\left(u^{(l)}_*\right) \leq \frac{1}{\lambda^{(l)}_k\sqrt{2\pi}} \end{aligned} \tag{5.38}$$

Finally

$$\begin{aligned} lim_{\lambda^{(l)}\downarrow 0}lim_{k\to\infty}f^{(l)}_k\left(u^{(l)}_*\right) = \infty \\ \forall_{u\neq u^{(l)}_*} : lim_{\lambda^{(l)}\downarrow 0}lim_{k\to\infty}f^{(l)}_k(u) = 0 \end{aligned} \tag{5.39}$$

We can safety conclude that the set of CARLA will converge to an equilibrium in the joint-action space as long as the parameter α is low enough for the learners to match their expected policies through time. Which of the multiple equilibria is selected will depend on the initialization of the parameter λ.

Let us analyze the following example to better illustrate the above described procedure. The analytical expression of the reward signal is shown in (5.40) and its contour representation in figure 5.2. There are three attractors in this example. The two local maxima located in the top left and bottom right corners have larger basins of attraction while the global maximum at the center has a narrower basin of attraction.

$$\begin{aligned}\rho\left(u^{(1)}, u^{(2)}\right) = {} & \operatorname{Bell}\left(u^{(1)}, u^{(2)}, 0.5, 0.5, 0.084\right) \\ & \bigcup 0.63 \operatorname{Bell}\left(u^{(1)}, u^{(2)}, 0.1, 0.9, 0.187\right) \\ & \bigcup 0.63 \operatorname{Bell}\left(u^{(1)}, u^{(2)}, 0.9, 0.1, 0.187\right)\end{aligned} \tag{5.40}$$

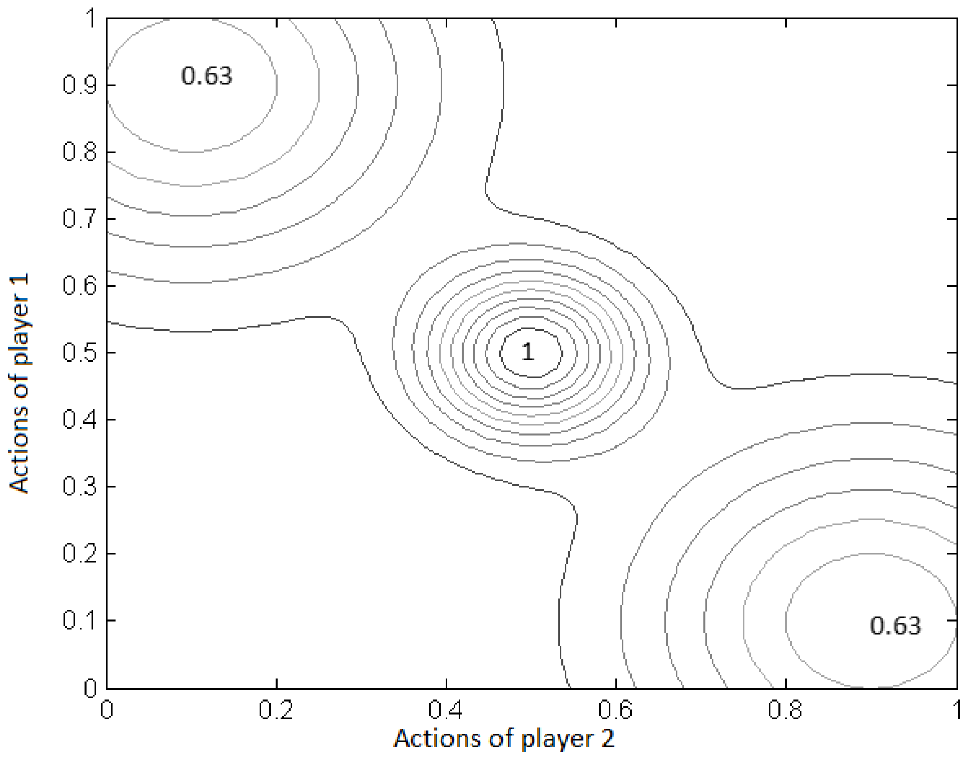


Figure 5.2: A two players game with three local optima - The contours represents combination of actions with the same reward. Notice that the global optimum is located in a narrower region, this region does not overlap with the other maxima from both agents point of view.

If both learners start with the uniform distribution we expect them to converge to the attractor that fits expression (5.34) the best.

Since we are only interested in the effect of λ we will use a fixed value over all time-steps. Two λ values will be explored: 0.04 and 0.007. The learners are expected to converge to either one of the attractors to the corners (either (0.1,0.9) or (0.9,0.1)) when using $\lambda = 0.04$ and to the global maximum (0.5,0.5) when $\lambda = 0.007$. In order to prevent the learners from deviating too much from their expected strategies, a very low α is used in this case 0.005. In a real setup we will never use such a low value. Because of the low learning rate we run 100 experiments for 30000 time-steps. When using $\lambda = 0.04$, half of the times the learners converged to the attractor (0.1,0.9) while the other half they converged to (0.9,0.1). In the second case, when using $\lambda = 0.007$, both agents converged to the global maximum.

We can conclude that in order to make the learners converge to the best attractor we either need information about the reward function or a better exploration of the joint-action space, as we will introduce in the next section.

5.5 ESCARLA

In order to introduce exploring selfish continuous action reinforcement learning automaton (ESCARLA) let us first recall ESRL, introduced for improving global performance in discrete action games (section 2.2.1). ESRL is an exploration method for a set of independent learning automatons (LAs), playing a repeated discrete action game. The basic idea of this method is that the set of independent LAs will converge to one of the Nash equilibriums (NEs) of the game, but not necessarily one from the Pareto front. ESRL proposes that once the agents converge to a NE, at least 2 agents should delete the action which is involved in the selected NE from their individual action spaces and restart learning. Since the agents are now using a reduced action set, they will converge to a different equilibrium point. Repeating this procedure allows the agents to find all dominant equilibria and agree on the best one, using only limited communication. The learning process is independent, each agent applies the LA learning algorithm, more precisely the reward in action update rule for discrete actions, using its individual reward and without knowing which action have been selected by the other agents. This allows to perform the learning in the individual actions spaces. Only after convergence, a communication protocol takes care of the exclusion of the appropriate actions. As the more interesting NEs are often also stronger attractors, the agents can quite efficiently reach Pareto optimal Nash equilibria, without the need to exhaustively explore all NE. The problem with applying this approach in continuous action games, is that it makes no sense for the agents to delete the action to which they converged. Rather, a vicinity around the action should be identified and excluded.

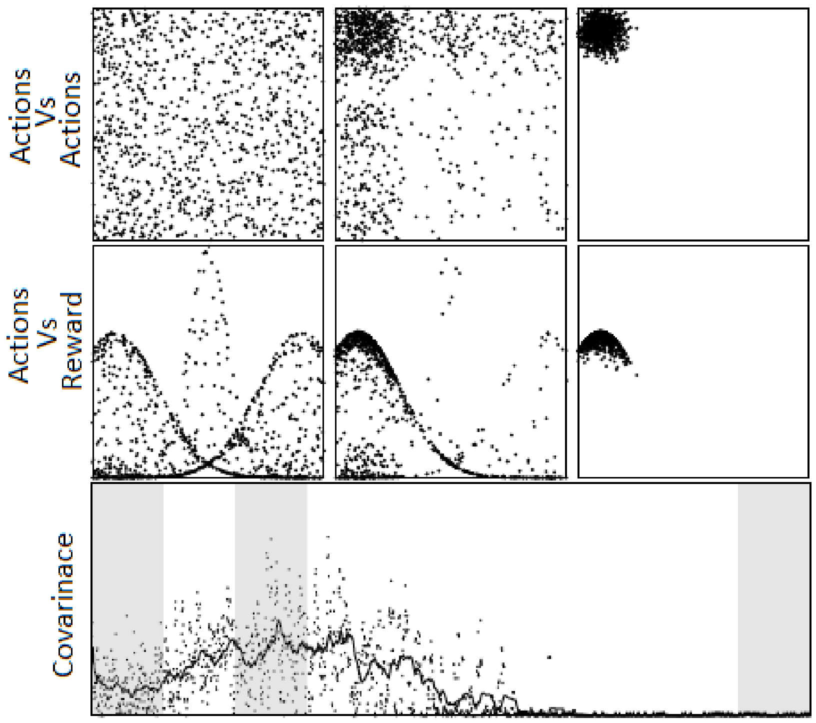


Figure 5.3: Relation between covariance and exploration - The row on top represents the joint action of the agents. The second row represents the reward collected for each action by the first agent. The left-most column collects samples from time-step 0 to 1000. The column in the center collects samples from time-step 2000 to 3000. The right-most column collects samples from time-step 9000 to 10000. The last row shows the absolute value of the covariance between actions and rewards for the first agent through the 10000 time-steps. The intervals corresponding to 0 to 1000, 2000 to 3000 and 9000 to 10000 are shaded

In order to identify the basin of attraction, we propose to use the absolute value of the covariance between the actions and rewards as a metric. Figure 5.3 shows the relation between the exploration and the covariance between actions and rewards from a single agent point of view. The parameter setting used for this example is the same as the one selected in section 5.4. The first row shows a global view of the exploration. Three time intervals are shown. The first interval is the beginning of learning (time-steps from 0 to 1000). The second interval is when the learners are leaving global exploration (time-steps from 2000 to 3000). The last interval selected is when agents have converged to the local attractor (time-steps from 9000 to 10000).

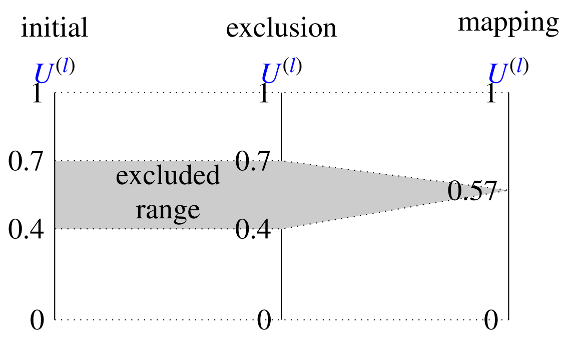
In the middle top figure, one can notice that the learners are converging to a locally superior strategy and are trapped within its basin of attraction. The dark left top cluster shows that the actions near to the locally superior strategy (0.1,0.9) are now sampled more frequently. The challenge now is to identify this using only information local to the agents. The second row shows the local information the agents have at their disposal. The same time intervals are represented in each respective column. The plots depict the selected actions on the horizontal axis with the corresponding reward on the vertical axis. The bottom row shows the absolute value of the covariance between actions and rewards over the whole learning process. Additionally, in order to have a better idea of how this covariance is evolving, the solid curve represents its average. The three intervals considered above are shaded in gray. This covariance has a low value at the beginning of learning since a lot of exploration is performed by both agents. When the agents start to be influenced by the basin of attraction of a local attractor then the noise in the rewards significantly drops, so the covariance reaches its maximum. As agents converge to a locally superior strategy, less exploration is performed and therefore the covariance value drops down to zero. The appropriate region to exclude after the agent's actions have converged to the local optimum, can therefore be estimated at the moment when the absolute value of the covariance reaches its maximum value.

A good way of estimating this region is by using the percentiles of the probability density function of the actions. For a given confidence value c we can define a region as shown in expression (5.41) where percentile(p, f) represents the value where the probability density function f accumulates the probability p. Notice that the lower c, the wider the region defined by expression (5.41), but also the lower the probability for the actions in the region to have been properly explored.

$$\left[\text{percentile}\left(\frac{1-c}{2}, f\right), \text{percentile}\left(1 - \frac{1-c}{2}, f\right)\right] \tag{5.41}$$

Our proposal is to let the agents learn until they all reach a threshold value of convergence $conv_{stp}$. Then each agent should delete the region defined by (5.41) from its action space – examples in the experimental results section will use $conv_{stp} = 0.95$ and $c = 0.9$. Deleting the action range implies modifying the action space so we need to map the remaining action space onto a compact set again as shown in Figure 5.4. In the example given, the region to exclude is $(0.4, 0.7)$. The new action space will be composed by the two remaining intervals $[0, 0.4]$ and $[0.7, 1]$. This new action space will be linearly mapped into $[0, 1]$. Both of them have a subset

Figure 5.4: Mapping process - The new action space will stay as the previous one but actions will be mapped in order to avoid the deleted range (0.4,0.7). Initially, U^l runs continuously in $[0,1]$. The region to exclude is $[0.4,0.7]$. Then the remaining intervals been $[0,0.4]$ and $[0.7,1]$ are expanded to $[0,0.57]$ and $[0.57,1]$ respectively to cover the initial action space U.



on it proportional to their extension so the remaining actions are equally eligible as they were before the mapping.

This method can introduce some noise in the reward function, however since CARLA learners do not require continuity in the reward function, this does not cause problems. After the exclusion of the appropriate action range, all agents must restart the learning process in order to converge to another attractor. After sufficient exploration the agents should compare the reward obtained for the different joint actions to which they converged and agree on the strategy that yielded the highest reward. It is important to remember that we are assuming a common interest game, so therefore the agents have no problem to agree on the best combination of actions. The general algorithm is given next.

Algorithm 5.4 shows the main process of the ESCARLA method. It iterates between phases of exploration and synchronization. When sufficient exploration has occurred, the agents agreed on the best strategy. The exploration and synchronization phases are detailed in algorithms 5.5 and 5.6 respectively.

Algorithm 5.4 Exploring Selfish CARLA

```
1: repeat
2:     explore
3:     synchronize
4: until enough exploration
5: select best strategy
```

Algorithm 5.5 Exploration phase (ESCARLA)

```
1: initialize parameters
2: repeat
3:     sample action
4:     update strategy
5:     if maximum covariance then
6:         interval ← $\left[\text{percentile}\left(\frac{1-c}{2}, f\right), \text{percentile}\left(1 - \frac{1-c}{2}, f\right)\right]$
7:     end if
8: until convergence
```

Algorithm 5.6 Synchronization phase (ESCARLA)

```
1: exclude marked interval
```

5.6 MSCARLA

By combining discrete RL techniques and function approximators we allow RL to deal with large or continuous action-state spaces. Chapter 4 described some techniques. In the case of value iteration methods, the actions selected by the methods are still discrete however. If the optimal action to be selected has a smooth continuous dependence on the state, there is no way a discrete value iteration RL with a function approximator will find it. The quality of the approximation of the optimal actions depends on the approximation resolution, but finer resolutions will make the learning process longer and heavier. The CARLA method is able to deal with continuous action problems but does not incorporate state information since it is a bandit solver. We next present an approximate gradient-free policy iteration method based on a set of CARLAs taking advantage of the fact that this bandit problem can find optimal solutions on continuous action sets.

5.6.1 Policy representation

As introduced in section 2.3.3, CARLA is a continuous action bandit solver. This means that the algorithm does not incorporate state information in the learning process. Our proposal is to combine CARLA with function approximators to extend it to a multi-state method. Notice that this is a similar idea of what Wheeler Jr. & Narendra (1986) and Witten (1977) did by associating discrete LAs with the finite set of states in MDPs. Multi-state continuous action reinforcement learning automaton (MSCARLA) will use the function approximators for the policy representation, only as an approximation mapping over the state space as the original CARLA method can already deal with continuous actions. A set of CARLA learners will be used to represent the policy parameters. Every single CARLA will keep its own stateless policy and then the state-informed policy will be obtained from them by means of the function approximator. The ϑ parameter will refer now to the set of independent policies π_i of the stateless CARLAs. For optimality reasons we do not use, as the parameter vector, the PDF themselves but the set of actions $u_i \in U$ chosen by the policies corresponding to every stateless CARLA. In an actor-critic approach, this module of the algorithm would be the actor. The MSCARLA policy can be calculated as shown in expression (5.42).

$$\widehat{\pi}(x; \vartheta) = H(x; \vartheta) \tag{5.42}$$

In order to use a normalized aggregation of the policies of the base learners, let us define a function $W : X \to \mathbb{R}^n$ representing the weight of every CARLA in the decision of the action for a given state $x \in X$ where n is the size of the state space X. We may now redefine H in terms of W as (5.43). Notice that the function is now normalized so each stateless CARLA may select an action $u_i \in U$ and the final aggregation of all actions will still be in the action space U.

$$H(x; \vartheta) = \frac{W(x)^T \vartheta}{\sum_j W_j(x)} \tag{5.43}$$

Two examples are given below to clarify this procedure.

The first example is using tile coding as the function approximator for the state. Expression (4.6) referred to the original tile coding, repeated here for easier reading:

$$[H(\theta)](x, u) = \sum_{i=1}^{N} \psi_i(x, u)^T \vartheta$$

Notice that the original formulation takes as parameter the state x and the control action u as well. Now we are only interested in approximating over the state space so we use the method as shown in 5.44. Notice that here ϑ stands for a vector with the actions generated by all stateless CARLAs in the system. For optimality issues when coding the method, we will not ask every base learner for its action. Instead, we may only take into account the CARLAs where $\psi_i(x)$ is not zero.

$$H(x; \vartheta) = \sum_{i=1}^{N} \psi_i(x)^T \vartheta \tag{5.44}$$

In this case W would be written as shown in equation (5.45).

$$W(x) = \sum_{i=1}^{N} \psi_i(x) \tag{5.45}$$

The normalized approximator (5.43) would finally be written as (5.46).

$$\begin{aligned} H(x; \vartheta) &= \frac{W(x)^T \; \vartheta}{\sum_j W_j(x)} \\ &= \frac{\sum_{i=1}^{N} \psi_i(x)^T \; \vartheta}{\sum_j \sum_{i=1}^{N} \psi_{i,j}(x)} \end{aligned} \tag{5.46}$$

The second example is the kernel based version. Kernel based function approximators were introduced in (4.8) also repeated here for easier reading:

$$[H(\theta)](x, u) = \sum_{k_s=1}^{n_s} \kappa((x, u), (x_{k_s}, u_{k_s})) \vartheta_{k_s}$$

Expression (5.47) introduces the MSCARLA version. Again, ϑ stands for a vector with the actions generated by all stateless CARLAs in the system and x_{k_s} their positions in the state space. Now this vector's size varies with time as this is a nonparametric approximator.

$$H(x; \vartheta) = \sum_{k_s=1}^{n_s} \kappa(x, x_{k_s}) \vartheta_{k_s} \tag{5.47}$$

Equation (5.48) shows the same formulation but in vectorial form where x_k is the vector of positions of the base learners at time-step k.

$$H(x; \vartheta) = \kappa(x, x_k)^T \; \vartheta \tag{5.48}$$

From the previous formulation we define W as follows:

$$W(x) = \kappa(x, x_k) \tag{5.49}$$

Notice that this function produces a vector of variable size over time. Finally, the normalized approximator is defined in equation (5.50).

$$\begin{aligned} H(x; \vartheta) &= \frac{W(x)^T \; \vartheta}{\sum_j W_j(x)} \\ &= \frac{\kappa(x, x_k)^T \; \vartheta}{\sum_j \kappa_j(x, x_k)} \end{aligned} \tag{5.50}$$

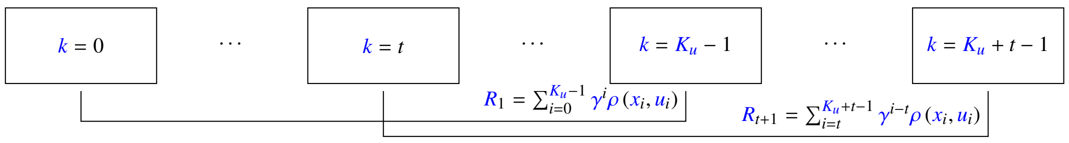

$k = 0$
$\cdots$
$k = t$
$\cdots$
$k = K_u - 1$
$\cdots$
$k = K_u + t - 1$
$R_1 = \sum_{i=0}^{K_u-1} \gamma^i \rho(x_i, u_i)$
$R_{t+1} = \sum_{i=t}^{K_u+t-1} \gamma^{i-t} \rho(x_i, u_i)$

Figure 5.5: Discounted sum over the trajectory - For calculating the reward signal at time step 0, the learner needs to collect K_u samples first and then aggregate them, e.g., with a discounted sum.

5.6.2 Policy evaluation

In order to complete the procedure, we still need to formalize the policy evaluation. We refer now to the critic in an actor-critic approach. Two important steps are necessary: evaluating the policy of the global learner and evaluating the policy of each base learner. The first step could be performed with any sample-based policy evaluation method reported in the literature. We use the truncated discounted return $R^{\widehat{\pi(\cdot;\vartheta)}}$. Figure 5.5 summarizes this procedure for a horizon length K_u. For the second step, we need to measure the extension of the influence of every base learner in the final decision and estimate the feedback of each base learner as explained next.

The reward received by every stateless CARLA has to be proportional to its contribution to the decision over the action. We propose to decompose the return R_k so the ith CARLA observes the reward $r_{i,k}$ defined by expression (5.51).

$$r_{i,k} = \frac{R_k\, W_i(x)}{\sum_j W_j(x)} \tag{5.51}$$

The rewards for each base learner in both examples introduced above are given next. The first example used tile coding (5.46) to approximate over the state space. The reward that every CARLA learner i will receive is calculated as equation (5.52).

$$\begin{aligned} r_{i,k} &= \frac{R_k\, W_i(x)}{\sum_j W_j(x)} \\ &= \frac{R_k\, \sum_{k=1}^{N} \psi_{k,i}(x)}{\sum_j \sum_{k=1}^{N} \psi_{k,j}(x)} \end{aligned} \tag{5.52}$$

Similarly, the reward that every CARLA learner i will receive in the second example, the kernel based approximator, is calculated as equation (5.53).

$$\begin{aligned} r_{i,k} &= \frac{R_k\, W_i(x)}{\sum_j W_j(x)} \\ &= \frac{R_k\, \kappa_i(x, x_k)}{\sum_j \kappa_j(x, x_k)} \end{aligned} \tag{5.53}$$

The policy update is a process that is naturally done by the stateless CARLAs. The function approximator will only modify the feedback signal that is perceived by them. Algorithm 5.7 formalizes the MSCARLA procedure.

Algorithm 5.7 represents the general procedure. The two examples described above can be obtained by setting W to (5.45) and (5.49) respectively.

5.7 Summary

The CARLA method is an LA based technique which is well suited for practical applications. It belongs to the category of policy iteration and it performs a global search of the action space. While other LA techniques search locally around the policy used at every time-step, CARLA may search the whole action space at a time if starts using the proper policy. Unfortunately, the method is not practical from a computational point of view, due to the excessive numerical calculations. In this chapter, an analysis of its performance was carried out, aimed at reducing this computational cost. The new derivations obtained, allow us to avoid the numerical integration. Section 5.1 introduced algorithm 5.1 which is capable of exactly the same performance in terms of exploration of the action space, but at a much lower computational cost.

The same algorithm was analyzed in section 5.2, now aiming to reduce the cost in terms of exploration. The convergence analysis showed, that by using a fixed spreading rate λ over time, the policy of the method will be limited so it cannot approach its local maximum more than this limit imposed by the latter parameter. In order to avoid this problem, the spreading rate was tuned during learning based on the distribution of the last actions taken. The technique is introduced in algorithm 5.2.

A final proposed optimization to the single agent CARLA, is a reshape of the reward function, introduced in section 5.3. The reshaped feedback signal enables the learner to keep learning at a high rate when approaching the maxima. The technique is introduced in algorithm 5.3.

Section 5.4 demonstrated the need of a better exploration technique of the joint-action space in games. Based on this hypothesis, section 5.5 introduced the extension of ESRL (see section 2.2.1) to continuous action spaces in algorithm 5.4.

Finally, in order to deal with learning in MDP settings we assemble a set of stateless CARLA learners with a function approximation over the state space and develop the MSCARLA algorithm 5.7.

All these theoretical results are practically tested in next chapters.

Algorithm 5.7 MSCARLA's general procedure.

```
Input: discount factor $\gamma$
    a function $W : X \to \mathbb{R}^n$ defining the function approximator
    discounted horizon length $K_u$
    base learner's parameters: learning rate $\alpha$, initial spreading rate $\lambda$ and reward's horizon
    length $K_r$
1:  initialize base learners with parameters $\alpha, \lambda$ and $K_r$
2:  for time step $k = 0$ to $K_u - 1$ do
3:      for all learner in base learners do
4:          if $W_i(x_k) \neq 0$ then
5:              let learner chooses action $\vartheta_{k,learner}$ from its policy
6:          else
7:              $\vartheta_{k,learner} \leftarrow 0$
8:          end if
9:      end for
10:     $u_k \leftarrow \frac{W(x_k)^T \vartheta_k}{\sum_j W_j(x_k)}$ {function approximator (5.43)}
11:     measure next state $x_{k+1}$
12:     measure reward $r_{k+1}$
13: end for
14: for all time step $k = K_u, K_u + 1, K_u + 2, \cdots$ do
15:     for all learner in base learners do
16:         $rew \leftarrow \sum_{i=k-K_u}^{k-1} \gamma^{i-k+K_u} r_{i+1}$ {discounted sum, see figure 5.5}
17:         if $W_i(x_{k-K_u}) \neq 0$ then
18:             let learner updates its policy with reward $\frac{rew\, W_{learner}(x_{k-K_u})}{\sum_j W_j(x_{k-K_u})}$ at action $\vartheta_{k,learner}$
19:         end if
20:         if $W_i(x_k) \neq 0$ then
21:             let learner chooses action $\vartheta_{k,learner}$ from its policy
22:         else
23:             $\vartheta_{k,learner} \leftarrow 0$
24:         end if
25:     end for
26:     $u_k \leftarrow \frac{W(x_k)^T \vartheta_k}{\sum_i W_i(x_k)}$
27:     measure next state $x_{k+1}$
28:     measure reward $r_{k+1}$
29: end for
```

Chapter 6

State-less applications

This chapter demonstrate the use the continuous action reinforcement learning automaton (CARLA) method in different real world applications. First, we show a comparison of the different implementations of CARLA and other bandit solvers. After, some experiments in real production machines show that it is quite simple to use the method without excessive parameters tuning. No strong restrictions exist on the reward function and, unlike the hierarchical optimistic optimization (HOO) method, for example, that has large memory requirements, the CARLA learner can solve problems without high computational nor memory requirements.

First, in section 6.1, we show the advantages of the improved methods when tested in some test cases. After this comparison, four real world control applications are tackled: controlling a linear motor in section 6.2.1, learning the parameters of a linear controller for the autonomous driving of a tractor in section 6.2.2, controlling a wire winding machine in section 6.2.3, and learning the set points in a hydrologic system in section 6.2.4.

6.1 Test cases

We perform the comparison in two scenarios: single agent settings in section 6.1.1, and in a game setting in section 6.1.2.

6.1.1 Single agent

The first set of test cases concerns single agent problems. Different scenarios are introduced below.

Six example cases are first tested. Their analytical expressions are given next in equations (6.1), (6.2), (6.3), (6.4), (6.5) and (6.6).

$$\rho_1(u) = B_{0.5,0.04}(u) \tag{6.1}$$

$$\rho_2(u) = 0.9B_{0.2,1}(u) \tag{6.2}$$

$$\rho_3(u) = (0.9B_{0.2,0.16}(u)) \bigcup B_{0.9,0.09}(u) \tag{6.3}$$

$$\rho_4(u) = 0.9\rho_1(u) + rand(0.1) \tag{6.4}$$

$$\rho_5(u) = 0.8\rho_2(u) + rand(0.1) \tag{6.5}$$

$$\rho_6(u) = 0.9\rho_3(u) + rand(0.1) \tag{6.6}$$

The test cases are described by the use of the binary operator $\bigcup$ where $x \bigcup y = x+y-xy$, the function $rand : \mathbb{R} \rightarrow \mathbb{R}$ that produces a random number normally distributed around 0 and with standard deviation specify by the one parameter of the function, and the function $B_{\mu,\Sigma} : U \rightarrow \mathbb{R}$ been a Gaussian bell centered at $\boldsymbol{\mu}$ and with covariance matrix $\boldsymbol{\Sigma}$ as shown in equation (6.7).

$$B_{\boldsymbol{\mu},\boldsymbol{\Sigma}^{-1}}(\boldsymbol{u}) = e^{-\frac{1}{2}(\boldsymbol{u}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{u}-\boldsymbol{\mu})} \tag{6.7}$$

Their graphical representations are shown in figure 6.1.

Functions ρ_1, ρ_2, and ρ_3 are deterministic functions Function ρ_1 represents an easy learning problem as it has an unique well defined maximum. Function ρ_2 also has only one maximum but the difference in the reward around it is very low. Finally, function ρ_3 represents the more challenging problem with two maxima. Functions ρ_4, ρ_5, and ρ_6 are their stochastic versions adding the additional difficulty of noisy observations.

Using these functions, we compare the performance of the original CARLA method with the performance obtained with the different transformations obtained in the previous chapter. We will also compare the methods reported in this dissertation with the other bandit solvers introduced in section 2.3. For these experiments, we use a low value for the learning rate of the CARLA method in order to achieve a more stable behavior. In this case we use $\alpha = 0.1$. The

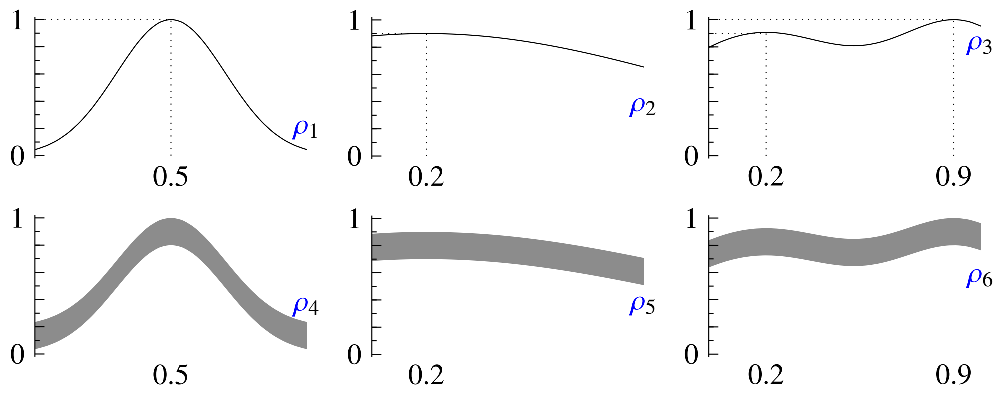


Figure 6.1: Graphic representation of the reward functions - The first row represents the deterministic functions (6.1), (6.2), and (6.3). The second row shows the stochastic functions (6.4), (6.5), and (6.6)

spreading rate parameter is set to 0.2. In the case of the continuous action learning automaton (CALA) method, we are use a low value for the step size, in this case $\alpha = 0.01$. The lower bound for the standard deviation is set to $\sigma_L = 0.03$ and the constant $K_+ = 5$. The initial distribution's parameters are set to $\mu_0 = 0$ and $\sigma_0 = 1$. These values were experimentally verified to exhibit stable convergence behavior. Two parameters have to be tuned for the HOO method been ϱ and v_1. These two parameters characterize the reward function and determine how much to explore suboptimal actions. The explanation on how to set these parameters is given by Bubeck *et al.* (2011). Notice that when no information about the reward function is available we would need to use generic values which will lower the convergence rate of the method. The parameter values for each function are given in table 6.1.Figure 6.2 reports the average reward. The red and green curves represents the CALA and HOO results respectively while the gray curves the results when using CARLA. The dotted gray curve reports the original (unchanged) CARLA method, the solid light gray reports the method including the automatically tuning of the spreading rate and the solid dark gray curve reports the method including the automatically tuning of the spreading rate and the standardization of the rewards as well. As it may be seen in the figure, the CALA or HOO learning technique always obtain a better performance than the original CARLA technique. Still, these methods have some drawbacks. The CALA method relies on the assumption that the reward function is smooth. If we have more noise in the reward function the lower bound for the σ parameter needs to increase affecting its performance. The HOO method also presents drawbacks regarding the initial knowledge of the reward function. Notice that at the beginning it starts learning very fast and then the performance dramatically drops. This happens because the ϱ and v_1 parameters controlling the exploration are very suited for the overall function but for a smaller region these parameters allow to much exploration. This problem could be avoided by constantly updating these parameters but this will require the availability of sufficient information about the reward function which is unlikely. Furthermore, if we have that much information about the reward function, why to use an reinforcement learning (RL) technique? By providing the CARLA technique with the automatic tuning of the spreading rate, we increase its performance. Moreover, when combining this upgrade of the method with the standardization of rewards we obtain better results than with the CALA and HOO methods. Notice that we could think of including standardization of rewards to the latter methods as well. In such a case, we would be constantly changing the reward. For the CARLA method this is not a problem but we cannot expect the same from the other techniques. HOO bases its performance in learning the reward function. If we constantly change it, then the convergence will be seriously affected. CALA does not estimate the reward function but it is a gradient based method that uses the size of the difference between observations to update its parameters. If we constantly change the reward, which implies changing its gradient, then the method will also be severely affected.

	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6
ϱ	0.5359	0.2872	0.4061	0.5359	0.2872	0.4061
v_1	3.9187	1.0447	2.4623	3.9187	1.0447	2.4623

Table 6.1: Parameter setting of HOO for continuous action functions - Every row shows the values of the ϱ and v_1 parameters for each of the functions (represented by every column) considered.

The six examples given above are stationary (i.e., the reward does not change over time). We consider next a non-stationary problem. The reward function is similar to ρ_1 but the mean of the Gaussian is shifted starting at time-step 800. The size of the shifts increases over time until it stops changing at time-step 3200. Equation (6.8) shows the expression for the reward where k is the time-step.

$$\rho_1'(u) = B_{0.5+\Delta,0.04}(u) \tag{6.8}$$

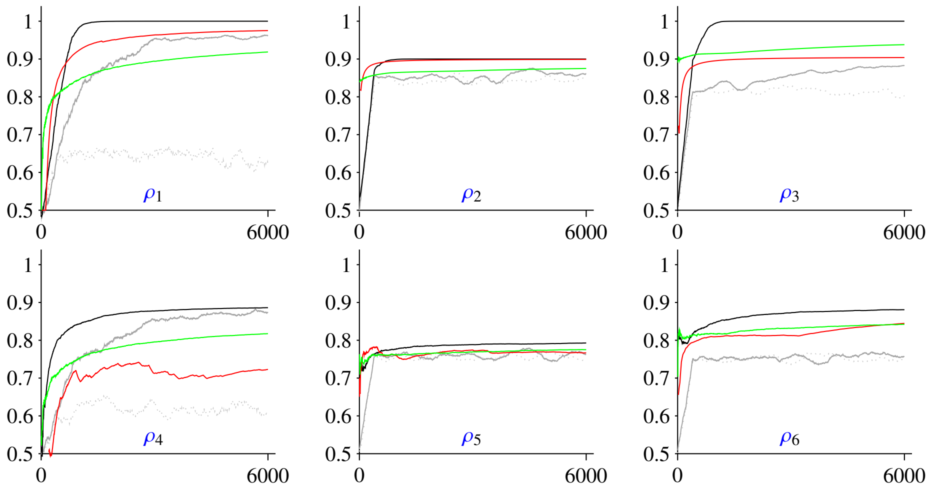


Figure 6.2: Comparison on continuous action bandit problems - The horizontal axis represent the time-step while the vertical axis represent the average reward. The six functions are tested and reported. The top row shows the deterministic functions ρ_1, ρ_2, and ρ_3 while the stochastic functions ρ_4, ρ_5, and ρ_6 are shown in the second row. The red curve represents the CALA results, the green the results when using the HOO method and the gray curves the results when using CARLA. The dotted gray curve reports the original (unchanged) CARLA method, the solid light gray reports the method including the automatically tuning of the spreading rate and the solid dark gray curve reports the method including the automatically tuning of the spreading rate and the standardization of the rewards as well.

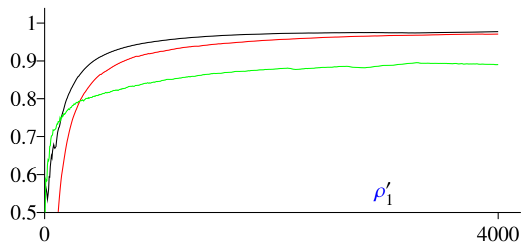


Figure 6.3: Comparison on a non-stationary bandit problem - The horizontal axis represent the time-step while the vertical axis represent the average reward. The non-stationary function is tested and reported. The red curve represents the CALA results, the green the results when using the HOO method and the black curve the results when using CARLA.

where

$$\Delta = \begin{cases} 0 & k < 800 \\ \left(\frac{k-800}{3600}\right)^2 & 800 \leq k < 3200 \\ \frac{4}{9} & k \geq 3200 \end{cases}$$

Notice that CARLA needs now to constantly explore for changes in the reward function. To this end, we lower bound the spreading rate so that the method can detect the changes in the environment. The lower bound for the spreading rate is 0.02. Figure 6.3 reports the results in the non-stationary function. Observe that while the CALA and the CARLA methods are able to adapt to the changes of the function over time, the HOO method needs a longer period for such an adaptation, lowering the average reward considerably.

6.1.2 Games

Some games are presented next. These two games are introduced as complex examples where the basins of attractions are overlapping from both agent point of views. The analytical functions are shown in equations (6.9) and (6.10) where I is the identity matrix. Figure 6.4 shows the contour representation of both functions.

$$\rho_7(\boldsymbol{u}) = 0.3125\,(1 + \sin(9u_1u_2))^2\,\frac{u_1 + u_2}{2} \tag{6.9}$$

$$\begin{aligned}\rho_8(\boldsymbol{u}) = B_{(0.5,0.5)^T,0.002I}&\left(\left(\begin{array}{c} u_1^2 \\ u_2 \end{array}\right)\right)\bigcup\\ 0.7B_{(0.1,0.1)^T,0.2I}&\left(\left(\begin{array}{c} u_1^2 \\ u_2 \end{array}\right)\right)\bigcup\\ 0.7B_{(0.1,0.9)^T,0.2I}&\left(\left(\begin{array}{c} u_1^2 \\ u_2 \end{array}\right)\right)\bigcup\\ 0.7B_{(0.9,0.1)^T,0.2I}&\left(\left(\begin{array}{c} u_1^2 \\ u_2 \end{array}\right)\right)\bigcup\\ 0.7B_{(0.9,0.9)^T,0.2I}&\left(\left(\begin{array}{c} u_1^2 \\ u_2 \end{array}\right)\right)\end{aligned} \tag{6.10}$$

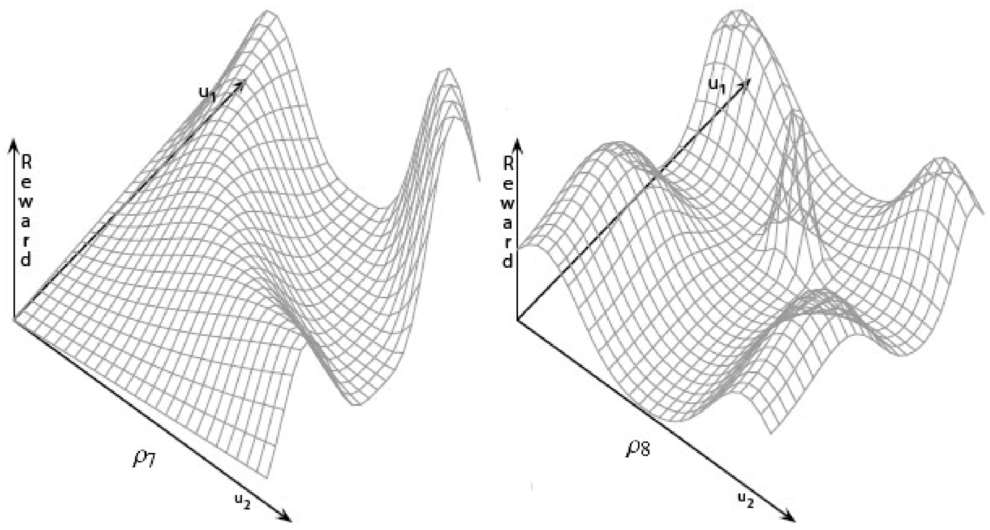
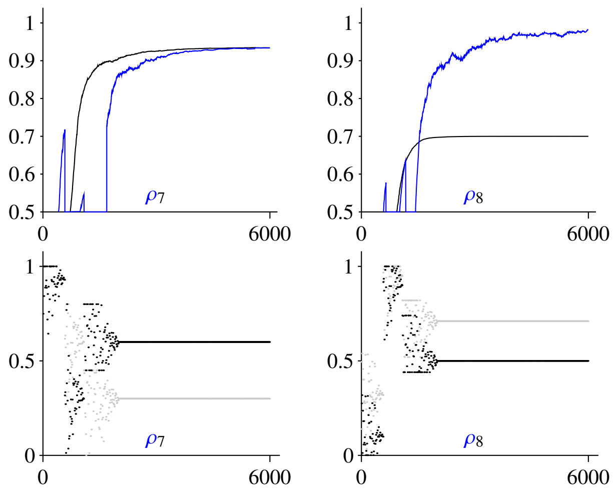


Figure 6.4: Reward functions of Two 2-player games - Notice in the first game to the left that the global optimum is located in a very narrow region, but this region overlaps with the other maximum in both axis (first and second agents axis). The second game to the right is even more complex since it also has a very narrow optimum region but it does not even overlap with the other maxima.

Figure 6.5 shows the learning performance of the agents. More specifically the average rewards collected over time are given in the figure on top and the actions selected (only for exploring selfish continuous action reinforcement learning automaton (ESCARLA)) every time-step are shown in the picture at the bottom. The parameters setting is the same for the CARLA learner (with spreading rate tuning and standardization of rewards) shown by the black curve.

Figure 6.5: Comparison on continuous action games - Average rewards for ρ_7 and ρ_8. For the first game, CARLA converges to the global optimum, sousing ESCARLA just delayed the convergence, but still the learners converged to the global optimum. On the other hand, for the second game, the improvement clearly led the agents to the global optimum in the third exploration restart. Average reward is shown on top while the actions are shown on bottom.



Notice that in the first game (left chart), the CARLAs are able to converge to the global optimum. In the second game (right chart) the learners get stuck in local optima.

The blue curve shows the results when using the ESCARLA. Notice that every peak in the rewards for ESCARLA implies a restart in the learning. Although a bit slower, the ESCARLA obtains more accurate convergence to the global optimum and does not get stuck in a suboptimal solution as CARLA on this second game. Observe in the second game (right chart) that after a first phase, further exploration is performed allowing the learners to scape the local optima.

Figure 6.6 shows the regions elimination process for the second game. In the phase one the agents converged to the joint action $(0.32, 0.10)$ and deleted the region $([0.04, 0.52], [0.01, 0.44])$ which is shown in black. In the second phase they converged to $(0.95, 0.9)$ and deleted the region $([0.82, 0.99], [0.74, 0.99])$. Finally – phase three –, they easily converged to $(0.71, 0.5)$.

We would like to stress that the basins of attraction of the local attractor are not symmetrical, in other words the roles of agents are not interchangeable. This can be seen by inspecting equation (6.10), the function is not symmetrical due to taking the square of the action of the first agent. As a consequence the range of actions excluded by each agent has not the same size.

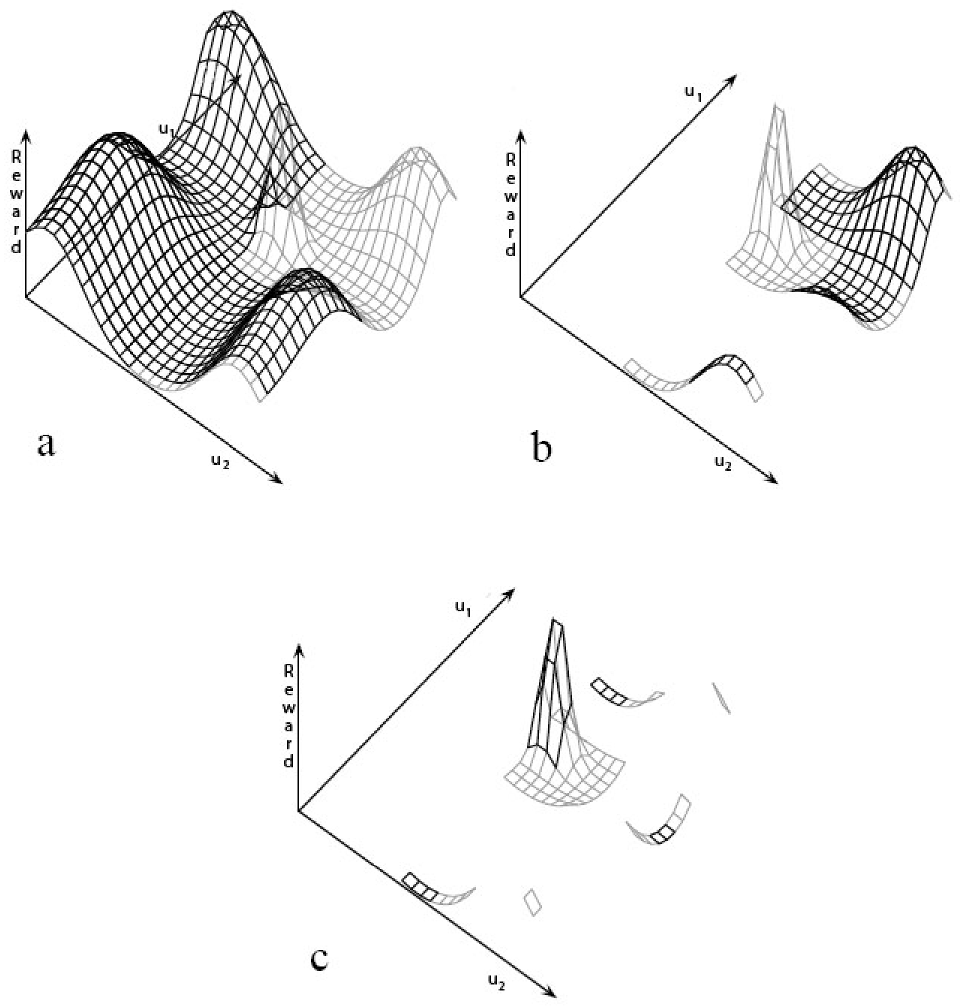


Figure 6.6: Region elimination process - Notice it is very unlikely to find the global optimum because it is a very narrow region. After ending the first exploration phase and restarting the learning, it is not that unlikely anymore. After the second restart, it is inevitable to find it.

The learners performed well in the above games but, to know whether the conclusion can be generalized we evaluate the new approach on randomly generated games. Random functions are generated for the reward function of the games. The functions are generated by the union of Gaussian bells. We consider both, single agent bandit problems, as well as 2-players games. Single agent functions use 1 or 2 bells while multi-agent functions will use between 1 and 6 of these bells. The number of bells is generated at random. The parameters of the bells $\mu \in [0, 1]$ and $\sigma \in [0.01, 1]$ are generated randomly too, as well as the factor $f \in [0.5, 1]$ multiplying the factor for the bells to make them different in amplitude. An extra parameter $r \in [0.75, 0.95]$ is generated for introducing noise in single agent settings. Two ways for introducing noise in single agent setting are explored with these functions, adding noisy bells or randomly selecting between them. The analytical expressions are introduced in (6.11). Notice that ρ_9, ρ_{10} and ρ_{11} are single agent problems while ρ_{12} is the only one representing a game of two players.

$$\begin{array}{l} \rho_9(u) = \bigcup_{i=1}^{n} f_i \, B_{\mu_i,\sigma_i I}(u) \\ \rho_{10}(u) = \bigcup_{i=1}^{n} \left(r_i \, f_i \, B_{\mu_i,\sigma_i I}(u) + \mathrm{rand}\,(1 - r_i) \right) \\ \rho_{11}(u) = pick \left\{ \frac{r_i}{\sum_1^n r_i}, f_i \, B_{\mu_i,\sigma_i I}(u) \right\} \\ \rho_{12}(\boldsymbol{u}) = \bigcup_{i=1}^{n} f_i B_{\mu_i,\sigma_i I}(\boldsymbol{u}) \end{array} \tag{6.11}$$

50 games are generated from each type of function. The average difference between the global maximum of the function and the final average reward collected by agents over 10000 time-steps are shown in table 6.2. The same parameter settings reported for the previous tests are used. A Wilcoxon rank test shows that in single-agent scenarios the automatic tuning of the spreading rate and the standardization of the rewards causes the CARLA method to perform better while in games the ESCARLA outperforms them both. The results of this statistical test can be seen in the last two columns of the table 6.2.

6.2 Real world applications

The cases discussed in this section are cases which have been selected within the Lecopro project (ref) (except from the "reservoir" case). Most cases have been tested on real setups, for the autonomous driving of the tractor we rely of realistic simulations which have been developed by the Lecopro members and are sufficiently realistic to be able to draw conclusions for the performance we can expect on the real setups.

	CARLA	CARLA$_+$	ESCARLA	CARLA Vs CARLA$_+$	CARLA$_+$ Vs ESCARLA
ρ_9	0.0876	0.0049	0.0146	0.000	0.352
ρ_{10}	0.0885	0.0092	0.0100	0.000	0.746
ρ_{11}	0.0786	0.0178	0.0171	0.000	0.515
ρ_{12}	0.0943	0.0851	0.0365	0.347	0.001

Table 6.2: Comparison between the different implementations of CARLA - Every row shows the average difference between the long-run reward and the current maximum reward. First column for the standard CARLA (labeled as CARLA), second using the automatically tuning of the spreading rate and the standardization of the rewards with CARLA (labeled as CARLA$_+$), and third column represents the ESCARLA method. The automatic tuning of the spreading rate in addition with the standardization of the rewards considerably reduces this difference in single-agent scenarios and the ESCARLA considerably reduces this difference for multi-agent games while it makes no harm in single-agent scenarios. Last two column shows the Wilcoxon rank test significance of the comparison of the CARLA algorithms.

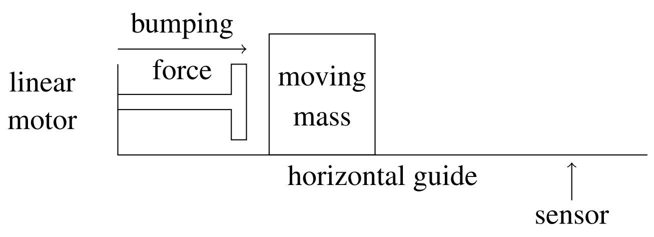
6.2.1 Linear motor

In many industrial applications, it is either not possible or too expensive to install sensors which measure the system's output over the complete stroke: instead, the motion can only be detected at certain discrete positions. The control objective in these systems is often not to track a complete trajectory accurately, but rather to achieve a given state at the sensor locations (e.g. to pass by the sensor at a given time, or with a given speed). Model-based control strategies are not suited for the control of these systems, due to the lack of sensor data.

6.2.1.1 Description

In this section, we present experiments with a simple abstraction of such systems, an electro-mechanical setup consisting of a linear (in the sense that the resultant force is in a linear direction but it does not imply linear dynamics) motor and a moving mass mounted on a straight horizontal guide (Figure 6.7). The position of the moving mass is monitored using a single discrete sensor, set along the guide, which fires at the passage of a small (1 cm) element attached to the mass. When the motor is activated, the mass is punched forward, and slides up to a certain position, depending on the duration and speed of the motor's stroke, and on the unknown friction among the mass and its guide. Two tasks are defined on this setup, with different objectives: a) let the mass pass the sensor at a predefined time (time task); b) let the

Figure 6.7: Sketch of the linear motor set-up - A linear motor bumps a mass through a horizontal guide. The position measurement can only be taken at one location with a discrete sensor.



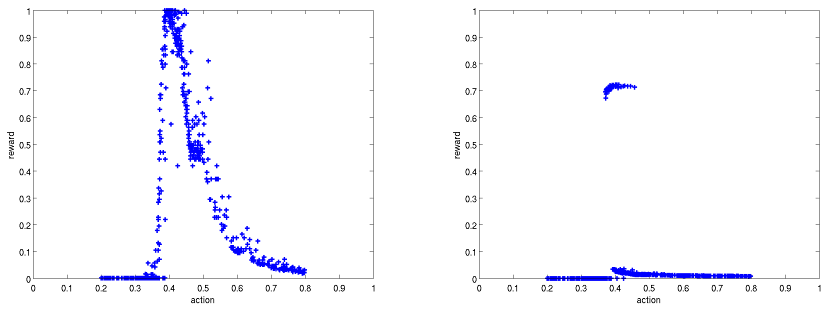


Figure 6.8: Reward functions for the linear motor set-up - Two control tasks are considered, sampled for 500 randomly chosen actions on a subset of the unit interval, including the optimal action. Left: time task. Right: position task.

mass stop exactly in front of the sensor (position task).

The motor is activated by a electric signal. We only consider constant amplitude signals, with a length varying on a closed interval, such that an action consists of a single scalar, which we normalize in $[0, 1]$, every single action is followed by a scalar reward, and no state information is used. For each task, a corresponding reward function is implemented, favoring the desired behavior. For the time task, given a target time t_0, the reward is given as $r_k = \exp\left(-c\,(t_k - t_0)^2\right)$, where c is a constant ($c = 1000$), and t_k is the time at which the sensor starts firing at trial k, or ∞ if the mass does not reach it. For the position task, the reward is given as the portion of time during which the sensor fires, over a fixed time interval ($4s$), measured from the beginning of the motor's movement. Figure 6.8 reports samples of the two reward functions taken during different days. Note that the system is highly stochastic, and repeating the same action may result in different rewards.

6.2.1.2 Results

The results of the modified CARLA algorithm are shown in figure 6.9 on both tasks. We plot single experiments with the actual setup, each lasting for 300 time-steps. CARLA was run with a learning rate of 0.8, and a spreading rate 0.25. The lower bar plot reports the returns observed at each time-step, while the upper plot reports the actions performed (+), along with the evolution of statistics describing the policy: mean and standard deviation of the CARLA's nonparametric distribution. Observe how the learner is able in a few trials to find a good policy. After 50 trials the decision maker is mainly receiving high rewards (bottom-left figure), and after 100 trials the policy has already converge around the optimal action (top-left figure).

The position task turns out to be more difficult: this can easily be explained comparing the reward samples (Fig. 6.8). The best action for the position task, around 0.4, is a needle in a haystack compared to the time task, where the reward function changes more gradually around the optimal action. Nonetheless, the CARLA algorithm is able to converge to a good action for both tasks. Notice that in this case, after 100 trials the learner is mainly exploring around the optimal action often receiving good rewards (bottom-right figure). Afterwards, it converges gradually closer to the optimal strategy (top-right figure).

6.2.2 Learning PID parameters to control an autonomous vehicle

We next discuss how to use CARLA for learning unknown parameters of a model based controller for autonomous driving. Our ultimate goal is to control a tractor to follow a reference trajectory.

6.2.2.1 Description

For controlling the tractor we have an actuator controlling the direction. This actuator consists of a hydraulic valve that makes the front wheels turn to target degree in both possible directions. We are interested in learning to control the actuator with the appropriate degree value for any state of the system so the tractor can match its reference trajectory. The tractor is provided with a GPS so we can track its position. Since we are interested in the tractor tracking any target trajectory the system state is not based on the absolute position of the tractor but on its relative position with respect to the target trajectory. More specifically, the system state is defined as a vector consisting of three components: the error y (i.e., the distance from the current position

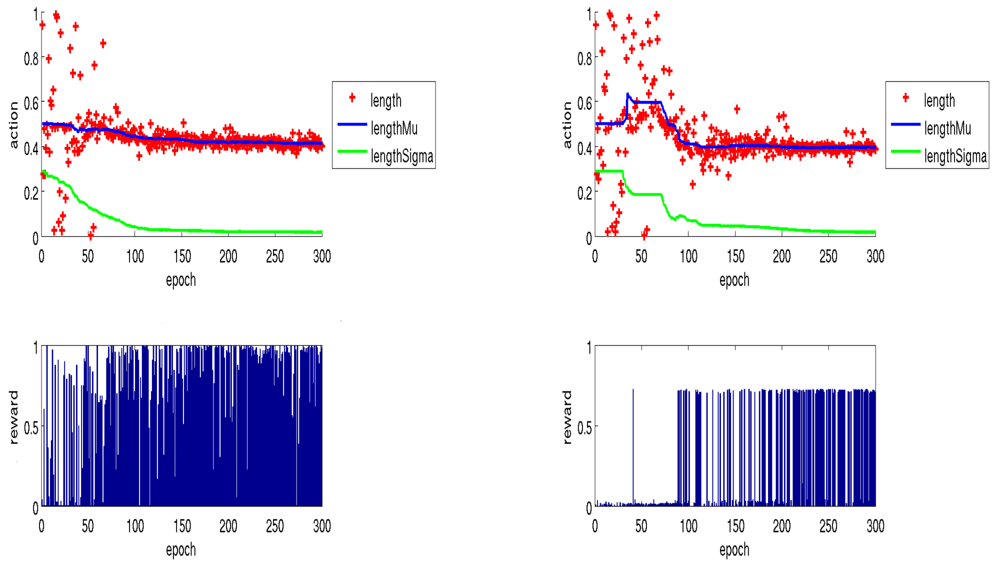


Figure 6.9: Results on the linear motor control task - The top row shows sampled actions together with the mean and standard deviation of the action probabilities. Bottom row shows obtained rewards. Left: time task. Right: position task.

to the trajectory), the first derivative of the error $\dot{y}$ (i.e., how fast the tractor is getting away from the trajectory), and the integral of the error y_{int} (i.e., the cumulative error).

The CARLA method has successfully been used for learning linear controllers (Howell & Best, 2000; Howell *et al.*, 1997). The easiest way of doing so is to learn a gain matrix K so the control action u can be expressed depending on the state x as:

$$u = K^T x$$

The goal of the learning is to find the value of K that maximizes the performance index (or minimizes the cost) of the system. We may learn this gain matrix from scratch and the learning procedure will eventually find the appropriate values. In a practical situation this is not feasible as it would require too much exploration, so some model information can be used. Figure 6.10 depicts this problem. The reference trajectory is drawn in red. In the figure, the angle of the tractor relative to the trajectory is called ϕ, the front wheels steering angle is $\theta \in [-35°, 35°]$, and the distance from the rear wheels to the front wheels is LT.

When the tractor drives at a constant speed v the error (meaning the distance to the trajectory) and the orientation (of the tractor relative to the trajectory) should change following the expressions shown in (6.12) where kT is the slip factor between the tractor wheels and the ground.

$$\begin{aligned} \dot{y} &= v \sin(\phi) \\ \dot{\phi} &= -\frac{kT}{LT} v \tan(\theta) \end{aligned} \tag{6.12}$$

For controlling this system we may use a combination of a feed-forward controller taking care of the trajectory that the tractor is facing and a feed-back controller correcting the error observed so far. Let Rt be the radius of the tangent circle to the trajectory at the current position. The feed-forward controller uses the control rule shown in (6.13). Remember that in this case the control action u is the steering angle θ. For the feed-back controller we use a PID controller where the gain matrix K is composed by the proportional gain K_p, the derivative gain K_d, and the integral gain K_i as shown in equation (6.14).

$$u = \operatorname{atan}\left(\frac{LT}{kTR}\right) \tag{6.13}$$

$$u = K^T x = \begin{pmatrix} K_p \\ K_d \\ K_i \end{pmatrix}^T \times \begin{pmatrix} y \\ \dot{y} \\ y_{int} \end{pmatrix} = K_p y + K_d \dot{y} + K_i y_{int} \tag{6.14}$$

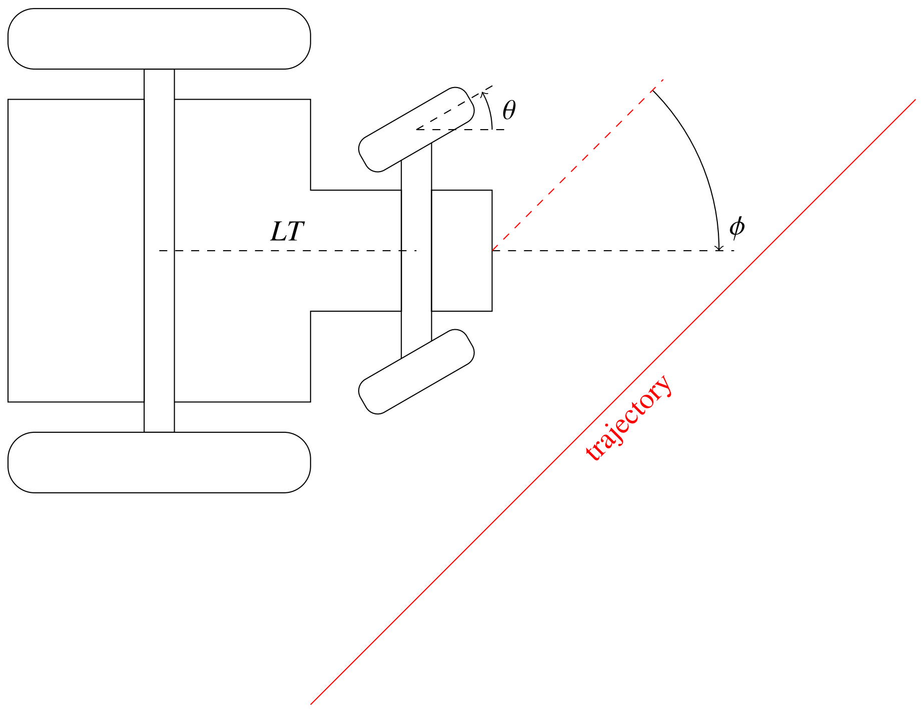


Figure 6.10: Controlling an autonomous vehicle - The tractor is expected to follow a reference trajectory. In the figure an example trajectory is plotted in red. The state of the system is described by the distance from the current position of the tractor to the trajectory namely the error, the first derivative of this error, and its integral. The control action is the steering angle θ.

where:

$$K_d = \frac{Kd_{eq}}{v^2}$$
$$K_p = kTv^2\frac{K_d^2}{3LT}$$
$$K_i = kT^2v^4\frac{K_d^3}{27LT^2}$$

and the parameter Kd_{eq} is the derivative gain of the PID at a constant speed $1m/s$ (the higher this parameter, the more aggressive the controller for reducing error).

Although this is a model based controller (the gain matrix is obtained based on the model), still there are some parameters such as the slip factor kT or the aggressiveness Kd_{eq} that cannot be easily predefined in advance. Learning the optimal gain matrix from scratch with an RL technique will take a very long time. However, if we use this model based controller and learn the one parameter kT by means of an RL technique, the learning time will be considerably lowered. Notice that unlike the aggressiveness, the slip factor is a parameter that depends on the environmental conditions and cannot be set to a predefined value since it will change from one field to the other or even from a day to the other in the same field. For the experiments we use an eight shaped trajectory as shown in figure 6.11.

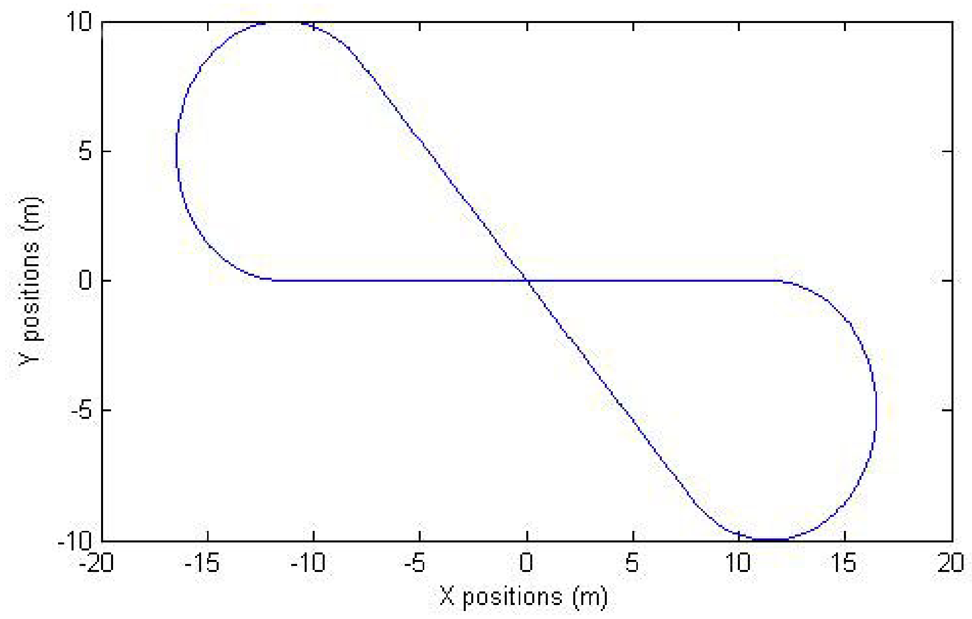


Figure 6.11: A reference trajectory for the tractor - This trajectory makes possible an infinitum repetition of trials.

The difficult part during driving is on the turns. Every learning time-step is defined by starting in the x-axis origin, turning, and crossing the x-axis origin again. During a learning time-step a CARLA learner selects an action, namely the slip factor. This slip factor is used to calculate the gain matrix and then the tractor is controlled during the whole time-step. At the ending of the time-step the learner is provided with a pay-off value and the process is repeated again. Selecting a wrong slip factor produces the controller to drive the tractor in an incorrect way. So the better the estimation of the slip factor, the lower the mean error during the trajectory. The mean error during the trajectory is not enough to measure the quality of the estimation of the slip factor as it does not only rely on the estimation itself but also in the initial position of the tractor when starting the learning time-step since the tractor can be off the reference trajectory. Notice that after a wrong estimation the tractor will most likely end up really far from the reference trajectory so even if we perfectly estimate the slip factor in the next time-step and the tractor drives rapidly approaching the reference trajectory, the mean error will still be high due to the accumulated error during the correction at the beginning. For this reason it is necessary to take into account in the pay-off function the initial position of the tractor. Instead of using only the accumulated error during one learning time-step, we propose to use the improvement made to the error during such a time-step. The pay-off function is defined in equation (6.15) where t_{k0} and T_k are the current simulation time when the learning time-step k begins and ends respectively, y_{k0} is the error at t_{k0} and y_{kt} is the error at time t during time-step k. Notice that this pay-off is not measuring how big or small is the accumulated error but how much the error reduces over time. If at t_{k0}, the tractor is perfectly situated on the trajectory, then the best possible outcome will be not to deviate again from the trajectory, any deviation will be punished in this case. If there was an initial error the best possible outcome is to go as fast as possible to the reference trajectory and not deviate from it. The slower the tractor goes to the trajectory, the lower the pay-off. If the tractor keeps deviating from the trajectory, then the reward will be even lower.

$$\rho_k(u_k) = \int_{t_{k0}}^{T_k} \left(\frac{y_{k0} - |y_{kt}|}{T_K - t_{k0}} \right) dt \tag{6.15}$$

6.2.2.2 Results

Figure 6.12 shows the error observed while learning the correct slip factor. All results are obtained in simulation. Notice that even with the correct slip factor the reference trajectory cannot be followed perfectly. This a normal behavior for a PID controller, the error is only corrected after it occurs. The dashed lines represent the maximum and minimum errors observed when using manually defined slip factors. The best results were obtained using a fixed slip factor of 0.6 (red dashed line). When using the extreme values of 0 or 1 the error range is wider (green and black dashed lines respectively). The solid blue curve shows the error when learning the slip factor with the CARLA procedure. Notice that some large errors exist during learning but the method rapidly converges to a region where the errors are smaller and it actually manages to keep the absolute value of the extreme error close to its minimum (about 0.3 meters).

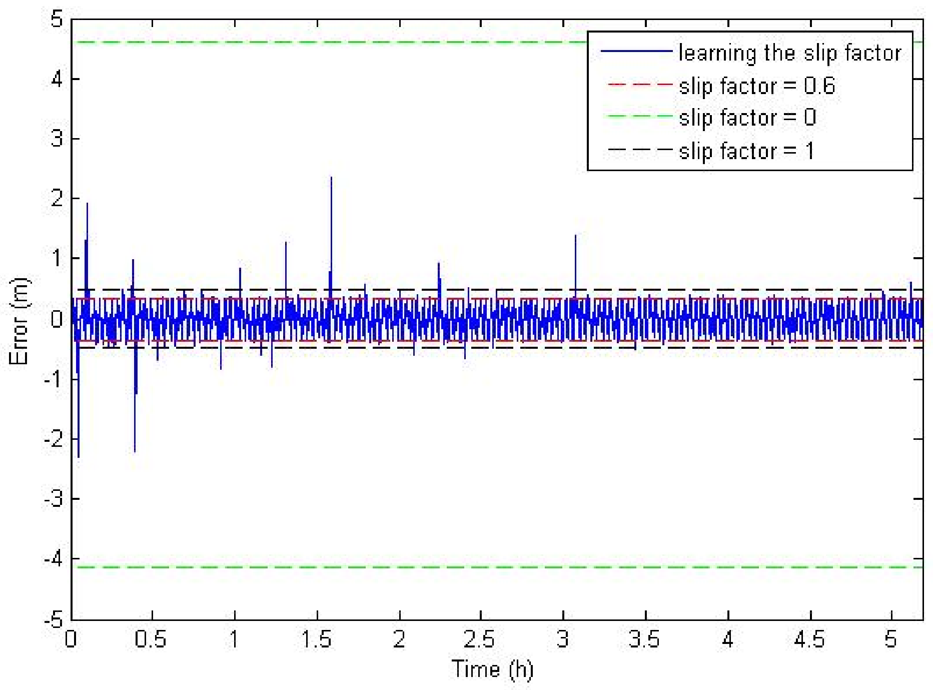


Figure 6.12: Error during the learning process of the slip factor - The discontinuous lines represent the maximum and minimum errors observed when using manually defined slip factors. The optimal performance is achieved by using a slip factor of 0.6. The learner is able to explore for this optimal value at a relatively low cost.

Figure 6.13 depicts how fast the controller corrects the error. In the figure we plot in the figure the average over time of the accumulated error, i.e., the average from the first iteration to the current one of the total accumulated area existing between the reference trajectory and the current one per turn. Again the same nomenclature is used.

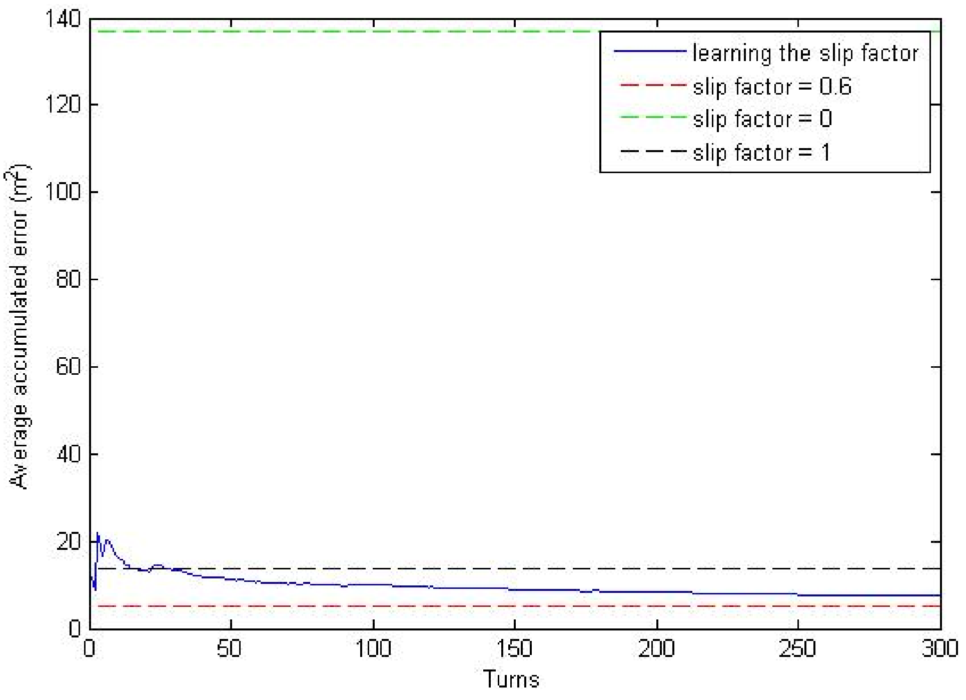


Figure 6.13: Average accumulated error during the learning process of the slip factor - Average total area existing between the reference trajectory and the current one at every turn.

Figure 6.14 shows how fast the learner corrects the controller in order to lower the error faster. We plot in the figure the the accumulated error over each turn, i.e., the total area existing between the reference trajectory and the current one at every turn. Again the same nomenclature is used.

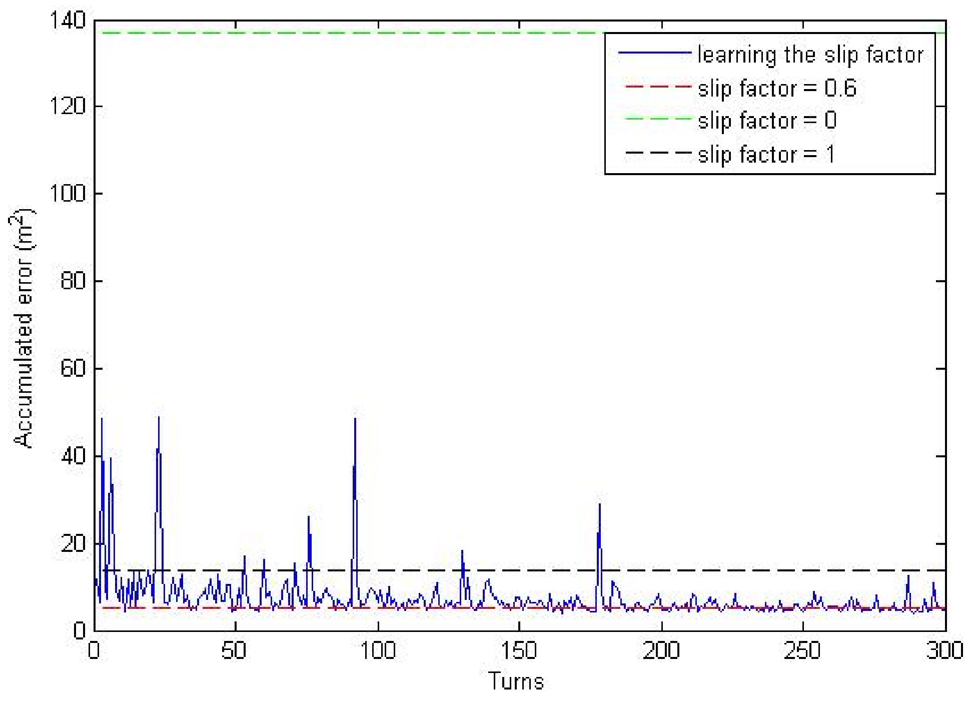


Figure 6.14: Accumulated error during the learning process of the slip factor - Total area existing between the reference trajectory and the current one at every turn.

Figure 6.15 shows the same results as figure 6.14 but in this case we only show the error obtained when using the learning procedure and the error obtained with the predefined slip factor 0.6. As can be noticed, the learner converges to an even better solution than the one found manually.

6.2.3 Wire winding machine

The wire winding problem is a very challenging case. In this application, a wire should be wound on a bobbin automatically meaning that the turning point should be decided upon by the control algorithm. This is highly non-deterministic problem and hard to model sufficiently accurate. As a consequence, traditional model-based techniques fail to properly control this

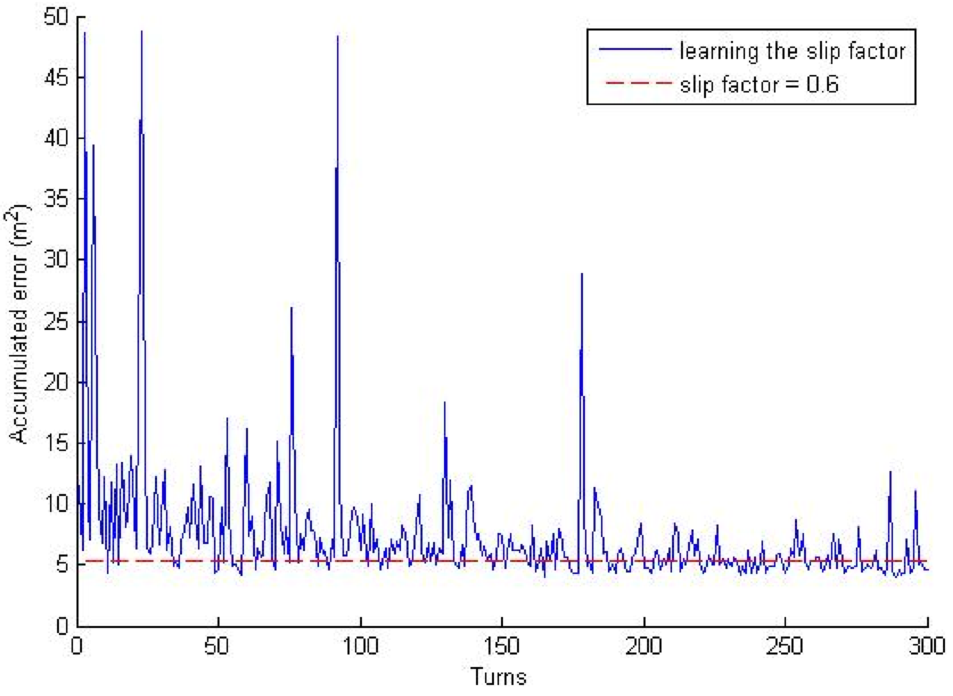
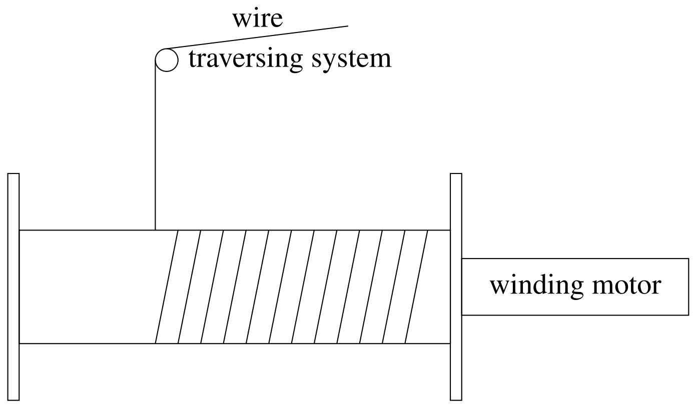


Figure 6.15: Accumulated error during the learning process of the slip factor - Same results as figure 6.14 but in this case we only show the error obtained using the learning procedure and the error obtained with the defined slip factor 0.6.

Figure 6.16: Wire winding machine – The bobbin rotates by the effect of the winding motor while the traversing system spreads the wire over the surface. The controller must decide on the turning point of the traversing system for an adequate winding.



process. Failing to determine the proper turning point during winding leads to accumulation or tangle of thread, which leads in turn to problems when unwinding the bobbin afterwards. As an additional challenge, in bobbins with bending flanges, the process will perform non-stationary, i.e., the optimal action will change over time.

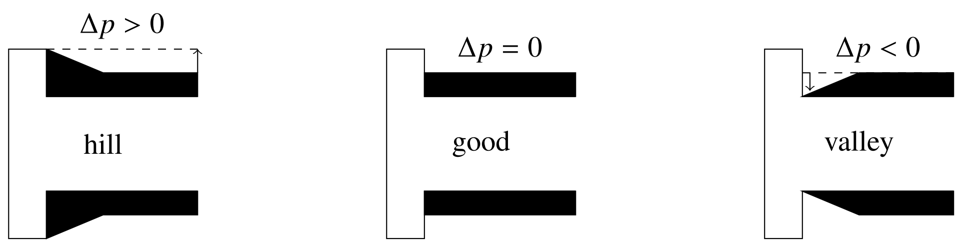
6.2.3.1 Description

Figure 6.16 shows an sketch of the general setup. The bobbin rotates by the effect of the winding motor while the traversing system spreads the wire over the surface. The controller must decide on the turning point of the traversing system for an adequate winding. This implies learning the proper control of the turning points of the back-and-forth moving traversing system, avoiding hills or valleys of the wire along the edges of the bobbin.

A sensor is attached to the traversing system for getting information about the winding process. Collecting information with this sensor produces highly noisy observations. If the traversing system turns too fast, problems close to the flange position will not be detected. Turning too slow will produce accumulation of the wire at the edge of the bobbin. We use the flatness of the surface Δp of the bobbin to reward the controller as figure 6.17 shows.

The straightforward way to address this problem is by using a fixed turning point for the traversing system. However due to uncertainty of the systems, this will not perform optimal. Furthermore, in bobbins with a bending flange the fixed turning point for the traversing system will absolutely fail since the action needs to adapt to the changing process.

Figure 6.17: Flatness of the surface measured as Δp – The black region corresponds to the wire. A positive Δp implies that the radius of the volume of wound wire close to the flange is larger than in the center of the bobbin, meaning a hill. A negative Δp implies that the radius of the volume of wound wire close to the flange is shorter than in the center of the bobbin, indicating a valley. If Δp is zero, we have a good profile, which needs no correction.



6.2.3.2 Results

The problem is tackled by using two CARLA controllers. One learning the proper turning point at the left flange and another at the right. The action is expressed as the distance from the center of the bobbin to the turning point. The reward function is based on the observed Δp:

$$r_k = -\Delta p$$

At a first look at this reward it may seem like we are positively rewarding the creation of valleys, but this is not correct. Let us analyze the different situations separately.

If a positive Δp is sensed, it means that there is a hill. The current action is not responsible for creating the hill, it already exists and the actions that generated it were rewarded already. Still, if with this action we are detecting the hill, it means that such an action is reinforcing even more this hill so it is a bad action and it has to be penalized. If a negative Δp is sensed, it means that there is a valley. Again, the valley was not created by the current action so we cannot penalize it for such an observation. Detecting the valley implies that the action is correcting the problem though. The controller has to be rewarded if it keeps detecting and correcting such a problem. If a nearly zero Δp is observed, maybe there is a valley or a hill but possibly it was not detected because the action made the winder turn before the hill or valley could be detected, but there is no evidence to categorize the action as a bad one.

The creation of hills is easily avoided with this reward function. The problem is how to cope with not detecting valleys. To tackle this problem we need a proper exploration strategy. We never want the controllers to explore for shorter actions since the problems are only detected with larger actions. If a hill is detected the rewards will make the controllers to select a shorter action. In case no problem is sensed the controller needs to keep exploring for larger actions in order to detect possible hills. We need to bias the exploration in the direction of larger actions if the controller is settled. At the beginning, the controllers must act as regular CARLAs learners. Once they converge to good actions, they have to keep slightly exploring around the optimal action. For this reason, the spreading rate is lower bounded. The exploration has to be directed in the positive direction. This is achieved by changing the action selection procedure. The standard procedure is to generate a random number *rand* uniformly distributed in $[0, 1]$ and then generate the action as $F^{-1}(rand)$ where F is the cumulative density function. We still use a random number *rand* in $[0, 1]$ but not uniformly distributed now. Instead, we use a linear density function with a positive slope.

It is important to stress that the exploration bias is only active when the controller is settled and not if it detected any problem. During the learning (i.e., the spreading rate has not reached its lower bound) the exploration bias is inactive. If the controller starts detecting a hill, the exploration bias is also deactivated.

Figure 6.18 shows the results obtained in a the flat-flange bobbin. Left plots correspond to the left controller and right plots to the right controller. Top down, the figure shows the turn point, the observed Δp, and the reward. Notice, in the top chart, how the action varies over time adapting to the detected problems. the acceptable boundaries for good performance and we see that the controller manages to keep the system within these boundaries.

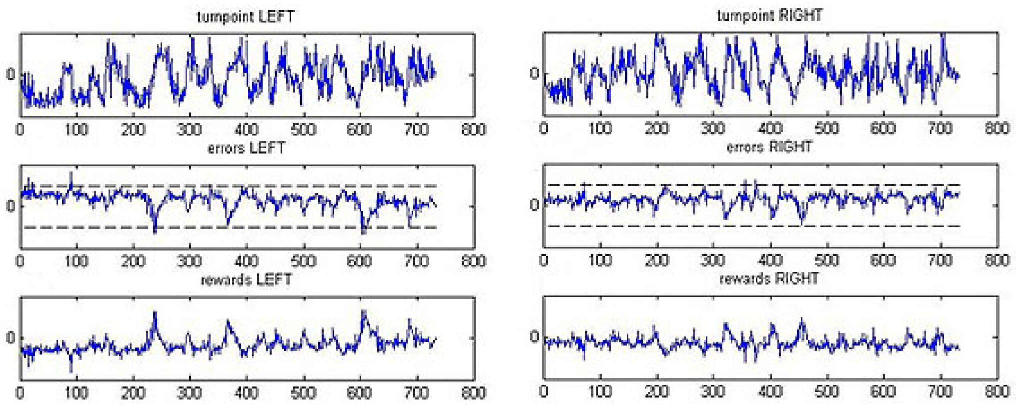


Figure 6.18: Wire winding results in a flat bobbin - Left plots correspond to the left controller and right plots to the right controller. Top down, the figure shows the turn point, the observed Δp, and the reward.

Figure 6.19 shows the results obtained in a the round-flange bobbin. These bobbins bend during the winding process resulting in a non-stationary problem. Left plots correspond to the left controller and right plots to the right controller. Top down, the figure shows the turn point, the observed Δp, and the reward. This is a more challenging problem since now the controllers face a non-stationary process. Notice, in the top chart, how the action adapts to the bending flange over time. We also show a boundary for good performance in the error plot by the hyphened line. The controllers keep the error within good boundaries.

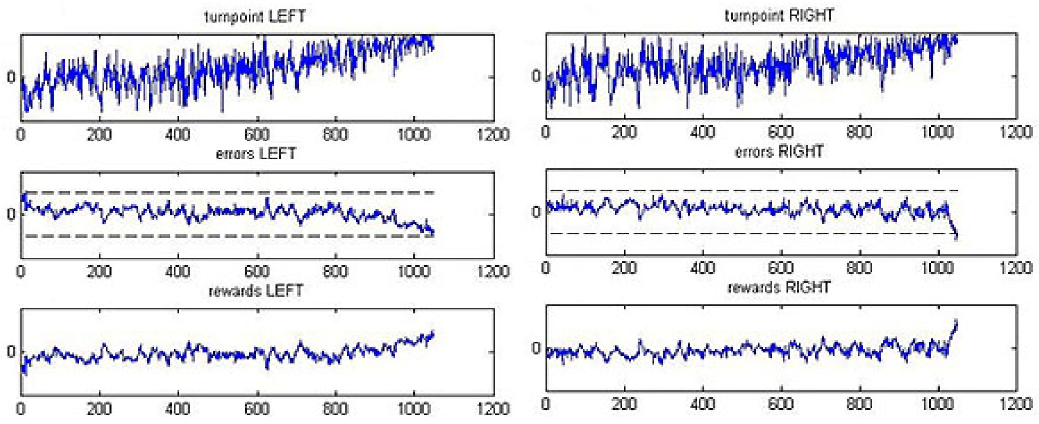


Figure 6.19: Wire winding results in a round bobbin - Left plots correspond to the left controller and right plots to the right controller. Top down, the figure shows the turn point, the observed Δp, and the reward.

6.2.4 A hydrologic problem

Below, we report a control problem which is a simplified yet representative problem of controlling the water level of a network of connected reservoirs or locks. Typically such a network is spread over different countries or authorities who want to decide independently on the level of the reservoirs belonging to their territory, yet it is not allowed to get the other ones in situations that would lead to flooding or water scarcity. An example is the Belgian lock systems. Water enters from France and Germany, and within Belgium the locks located in the South are controlled by Wallonia and the locks in the North by Flanders. Controlling reservoirs by RL is not entirely new. In Castelletti *et al.* (2012) an RL approach is described for controlling a single reservoir. In this paper the setting is much more complicated as a network of locks is considered.

6.2.4.1 Description

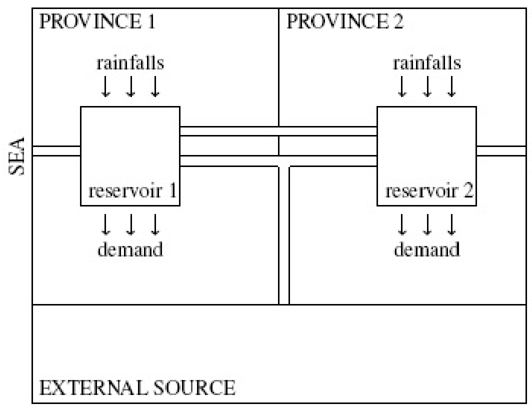
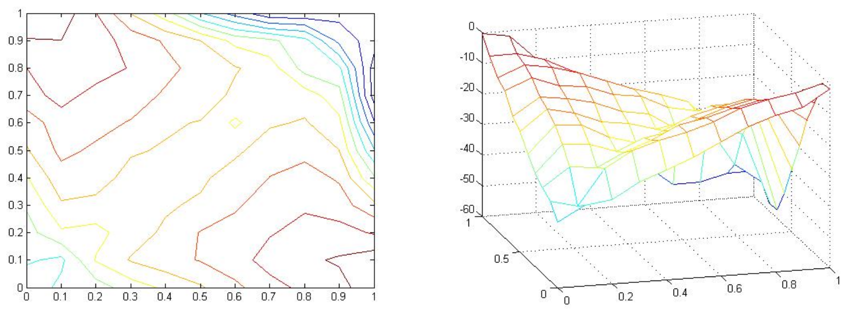


Figure 6.20: Water reservoirs problem - Two water reservoirs are controlled in two different provinces. They may flow water from one to the other or discharge it to the sea. The objective is to find the optimal target levels to use in order to avoid both: flooding and water scarcity.

Consider two water reservoirs that belong to different provinces. Each reservoir covers the demand of the region. The reservoirs are normally filled by rainfall. Both reservoirs may discharge to the sea at a limited flow by means of a valve. The reservoirs are connected so they can transport water from one to the other. This flow comes at a price that linearly depends on the levels of water of each reservoir, flowing from a high level to a lower one is much cheaper than the other way around. In case of emergency they may also use an external source for feeding or discharging. This external resource can only be used when both reservoirs are full or empty to avoid flooding or water scarcity. There is only one connection to the external source that becomes more expensive when used by both reservoirs at the same time. Figure 6.20 illustrates this problem.

A more detailed description of the system is given bellow. Rain fall is normally distributed with mean 0.5 and standard deviation 0.1 in both provinces. The demand is fixed in both provinces at 0.5. Valves controlling water to flow to the sea of both reservoirs allow a maximum flow of 0.35 per time unit. The cost of transporting a unit of water from or to the external source is 0.5 if it is used by only one province and 4 when used by the both of them. The cost for transporting water from one province to the other depends on the difference of water levels as shown in equation (6.16) where $level_1$ is the level of the water in the reservoir of the province which is giving the water and $level_2$ is the level of the water of the reservoir of the province

Figure 6.21: Reward function of the water reservoirs problem - To the left the contour representation of the reward while to the right its landscape.



which is asking for the water. The objective of this problem is to find the adequate levels of water that the reservoirs should keep to meet the demand at the lowest cost. If a reservoir's level is higher than the target level, it discharges the water to the other reservoir if the other needs it to reach its target level or to the sea as fast as the valve allows it. The cost is calculated after 50 time units starting from random levels of water at each reservoir. Figure 6.21 shows a contour representation of the average reward (calculated over a hundred simulations) that the learners may expect when facing this problem (a contour representation to the left and the landscape representation to the right). The expected cost is plotted over the z-axis and the actions runs from completely empty (0) to completely full (1).

$$cost = 50\,(level_2 - level_1 + 1) \tag{6.16}$$

6.2.4.2 Results

Notice there are two Nash equilibria in this problem. The total amount of water in both provinces has to be sufficient to avoid the need for feeding the reservoirs from the external source but not too high in order to avoid discharging to the external source at a high cost. Notice this is necessary due to the stochasticity of the rainfall. Because it is much more expensive to simultaneously use the external resource for both reservoirs and it is much cheaper to flow water from one reservoir to the other if there is a big difference in their levels it is preferable to store most of the water in one reservoir and keep the other level lower.

Table 6.3 shows the percentage of the times that each optimum was found with the ESCARLA method after every synchronization phase using the same settings as before. Each row

Table 6.3: Results of ESCARLA on the water reservoirs problem - Convergence to each optimum per exploration phase

	1st phase	2nd phase	total percent
$(0,1)$	26	24	50
$(1,0)$	24	26	50
	50	50	100

represents the percentage of times that the top-left or bottom-right maximum was found. The last row shows that ESCARLA was always successful in finding one of the Nash equilibria in the first phase and the other one in the second phase. Notice they both found both Nash equilibria with the same probability, which is to be expected as the problem is symmetric.

6.3 Summary

This chapter demonstrates how to use the CARLA learner for controlling state-less real life control problems. First, some toy functions are used for showing a comparison of the different existing techniques and how the changes proposed in this dissertation ensure a faster convergence. The first application is a rather simple problem: controlling a linear motor. The term linear comes from the direction of the force produced by the motor, not from its dynamics. This stationary continuous action bandit example is representative of several production machines where we need to control a motor with information obtained from limited sensors. A much more complex task is demonstrated by using a linear controller for driving an autonomous tractor. The RL technique is used here for learning the optimal parameters of the linear feedback controller. We also show how to control a wire winding machine as a non-stationary example. The last state-less application is in a game setting. It relates to a distributed hydrologic system where local controllers exist and we want to decide on their set-point for an optimal performance.

Chapter 7

Multi-state applications

The previous chapter explains in detail the state-less applications covered by the research exposed in this dissertation. Next we proceed with the explanation of problems represented by a Markov decision process (MDP) from which we can obtain information about the state of the system. First, we try some test cases in section 7.1. Afterwards, the autonomous tractor problem is addressed in section 7.2.1 and a game setting example is provided in section 7.2.2.

7.1 Test cases

Two abstract problems are addressed in this section: the double integrator problem and controlling a DC motor. The double integrator problem was presented by Santamaria *et al.* (1998) as an example solved using SARSA () with tile coding and kernel based approximation. The DC motor problem was introduced by Buşoniu *et al.* (2010) as an example solved using several approximated reinforcement learning (RL) techniques involving different basis function approximators.

7.1.1 Double integrator problem

The double integrator problem was presented by Santamaria *et al.* (1998). This is a challenging problem where the action affects indirectly the state variable taken into account in the reward function. As the goal state is approached by the process, the action has to be chosen more carefully making really necessary to use a continuous state-action technique.

7.1.1.1 Description

The double integrator is a system with linear dynamics and bidimensional state. A car of unit mass moving moves in a flat terrain subject to the application of a force. The state is described by the current position p and velocity v. In a vectorial representation we have $x_k = \begin{bmatrix} v & p \end{bmatrix}^T$. The controller decides on the acceleration a applied to the car $u_k = [a]$. The objective is to move the car from the starting state $x_0 = \begin{bmatrix} v_0 & p_0 \end{bmatrix}^T$ to the goal state $x_g = \begin{bmatrix} v_g & p_g \end{bmatrix}^T$ such that the return is maximized. The reward function is defined as a negative quadratic function of the difference between the current and desired position and the acceleration applied (7.2), where $Q = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ and $R = [1]$. Negative quadratic functions are widely used in robotics which specify an objective to drive. The tradeoff between time to reach the goal and effort is specified by matrices Q and R. The acceleration is bounded to the range $a \in [-1, 1]$. Additionally, the values of p and v must be kept in the same range as well. For this reason we use a penalty of -50 any time their values go out-bounded and restart the simulation. The system's dynamics are defined as shown in expression (7.1).

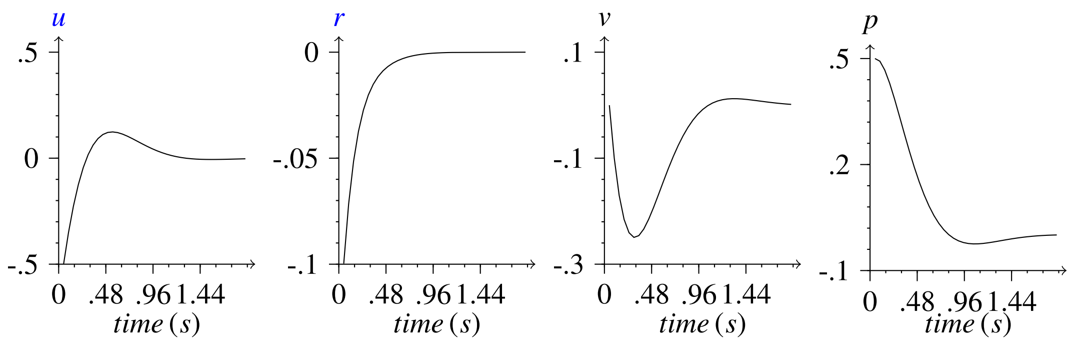
$$x_{k+1} = \begin{bmatrix} v_{k+1} \\ p_{k+1} \end{bmatrix} = \begin{bmatrix} v_k \\ p_k \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_k \\ p_k \end{bmatrix} \Delta t + \begin{bmatrix} 1 \\ 0 \end{bmatrix} [a] \, \Delta t = x_k + A x_k \Delta t + B u_k \Delta t \tag{7.1}$$

$$r_{k+1} = -\left(\left(x_k - x_g \right)^T Q \left(x_k - x_g \right) + u_k^T R u_k \right) \tag{7.2}$$

Results are obtained in simulation using a time step of $\Delta t = 0.05s$. The results are reported over 35 trials. Every trial consists of starting the system at state $x_k = \begin{bmatrix} 0 & 1 \end{bmatrix}$ and running it for 200 time-steps (i.e., 10 simulated seconds), unless the state gets out of bounds (i.e., when $|p| > 1$ or $|v| > 1$) or the goal state is reached (i.e., $\left| x_k - x_g \right| < \epsilon = 0.001$) in which cases the trial is finished.

This is a problem with linear dynamics and a quadratic cost function. In such a case, the optimal solution is known as the Linear-Quadratic Regulator (LQR). The derivation follows from the solution to the Hamilton-Bellman-Jacobi partial differential equation (Bertsekas, 1995; Stengel, 1994). The optimal action at every time-step k will be defined in the form $u_k^* = -K \left(x_k - x_g \right)$ where x_k is the current state of the system and x_g is the goal (or target) state. In this case, $K = \begin{bmatrix} \sqrt{2} & 1 \end{bmatrix}$ so the optimal policy is $u^* = -K \left(x_k - x_g \right) = -\begin{bmatrix} \sqrt{2} & 1 \end{bmatrix} x_k$. Figure 7.1 shows the trajectory from x_0 to x_g using the optimal policy.

Figure 7.1: Optimal solution for the double integrator problem - Trajectory when using optimal policy. From left to right: control action, reward, velocity and position.

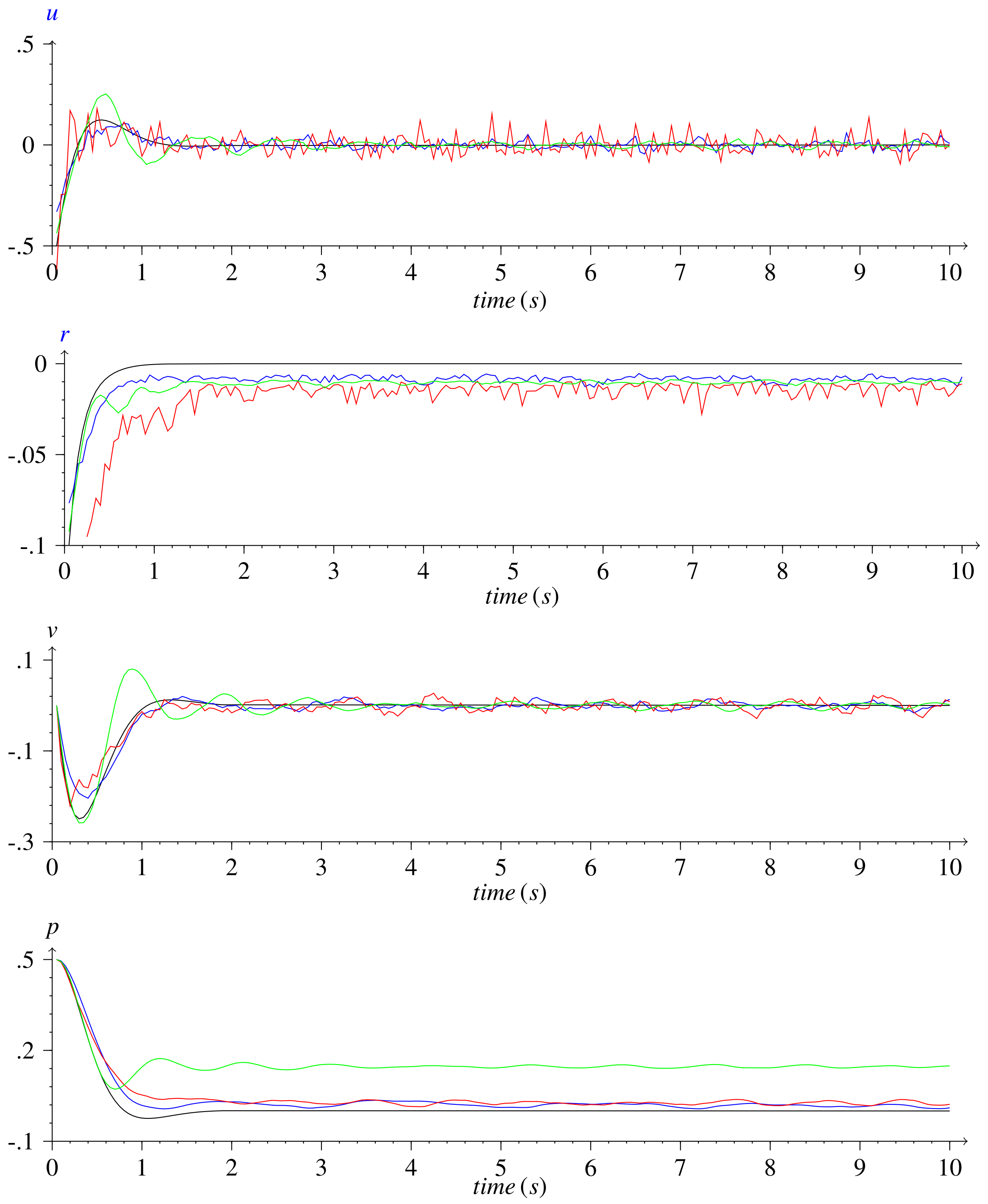


7.1.1.2 Results

The double integrator problem is tackled with the multi-state continuous action reinforcement learning automaton (MSCARLA) algorithm. The state space is normalized. Two function approximators are used as examples: tile coding and kernel based approximators. The state space is composed by two components in both examples so the tiles are defined the same for both applications. The parameter settings for the tile coding approximator is taken equal to the one reported by Sutton & Barto (1998). The tile coding approximator has 24 tiles, 12 taking both components in consideration, 6 taking only first component and 6 taking only the second component. All tiles are uniformly shifted over the state space. The same problem is also solved using the continuous actor-critic learning automaton (CACLA) algorithm and approximate SARSA with the same approximator's parameters. Notice that since CACLA is an actor-critic method, it uses exactly the same approximation mapping. Unlike CACLA, approximate SARSA include the action in the approximation mapping as well so instead of 24 tiles, 36 are necessary (including 12 tiles for the action).

Figure 7.2 reports the results on the double integrator problem using a tile coding approximator. The solid black curve represents the optimal solution. The red, green and blue lines report the results when using approximate SARSA, CACLA, MSCARLA respectively. All charts plot the performance using the policy learned after 200 episodes. In the case of CACLA, we report the trajectory using the greedy policy (without the Gaussian exploration) after learning. Top down, the charts represents the control action, the cost and both components of the state space: velocity and position. Observe that all methods behave comparably good. The higher reward is obtained by MSCARLA.

Figure 7.2: Learned policies to control the double integrator problem a using tile coding - Trajectory when using the policy learned by MSCARLA (blue line), approximate SARSA (red line), and CACLA (green line) using tile coding approximator. The black curve represents the optimal trajectory. Top down, the charts plots control action, cost, velocity and position.



In the case of the kernel based approximator, it uses the parameter setting reported by Buşoniu *et al.* (2010). The Gaussian kernel uses a width matrix $B^{-1} = \begin{bmatrix} 0.041 & 0 \\ 0 & 0.041 \end{bmatrix}$. It adds a new parameter to the approximation, i.e. a new stateless continuous action reinforcement learning automaton (CARLA), if the new observation is in a point such that no other observation has been taken before in a range of $1/35$ times the range of the state. The learning rate is 0.5 and the spreading rate 0.25. The horizon length is 50.

Figure 7.3 reports the results on the double integrator problem using a Gaussian kernel based approximator. The solid black curve represents the optimal solution. The red, and blue lines report the results when using approximate SARSA, and MSCARLA respectively. All charts plot the performance using the policy learned after 200 episodes. Top down, the charts represents the control action, the cost and both components of the state space: velocity and position. In this case, the SARSA method converged closer to the optimal policy than the MSCARLA learner.

Finally, figure 7.8 shows the accumulated cost (simple sum of all rewards from the initial state to the end of the episode) over learning. The black line represents the optimal cost while the red, green, and blue curves plot the accumulated cost when using approximate SARSA, CACLA, and MSCARLA respectively. Top down the charts show the double integrator problem using a tile coding, and a Gaussian kernel based approximator. This figure gives us a better idea of the learning process. Notice that when using a Gaussian kernel based approximator, the MSCARLA method is still learning after the 200 episodes. Although in the previous figure we observed a better performance from the SARSA learner than from the MSCARLA, eventually the MSCARLA can still do better. At this point of learning their behaviors are comparable at least. The learning is much faster using SARSA than using MSCARLA in both cases. A possible cause for this difference is that the MSCARLA method uses a reward-inaction rule for updating its policy so it is not as fast as value iteration methods when dealing with with high penalties.

7.1.2 DC motor

As a second example we provide a problem involving a DC motor control. This is a challenging problem as well with faster dynamics introduced by Buşoniu *et al.* (2010). In this case we can also find the optimal policy with a LQR making possible to know how close the RL methods behave to the optimal.

Figure 7.3: Learned policies to control the double integrator problem using a Gaussian kernel - Trajectory when using the policy learned by MSCARLA (blue line), approximate SARSA (red line), and CACLA (green line) using tile coding approximator. The black curve represents the optimal trajectory. Top down, the charts plots control action, cost, velocity and position.

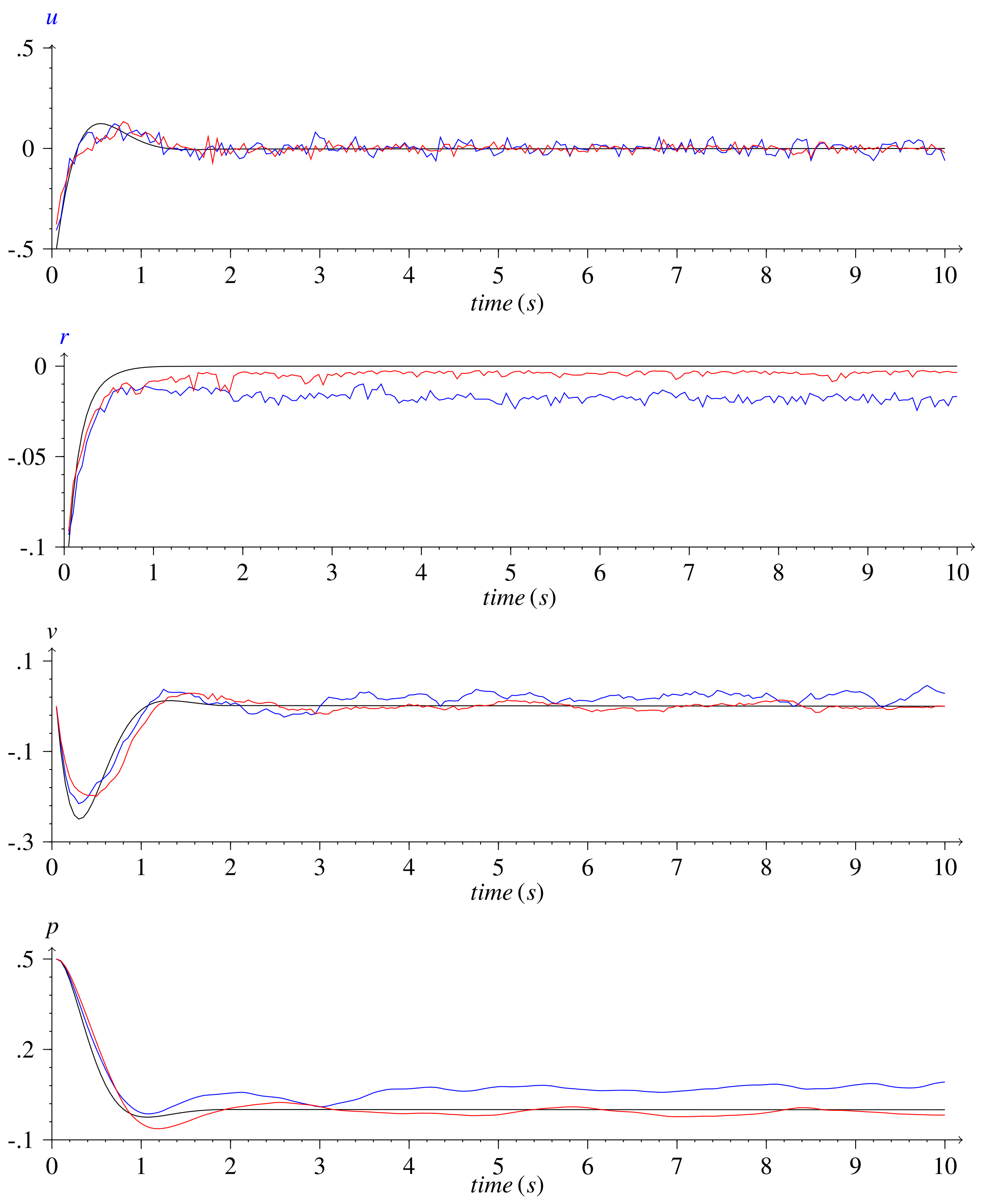
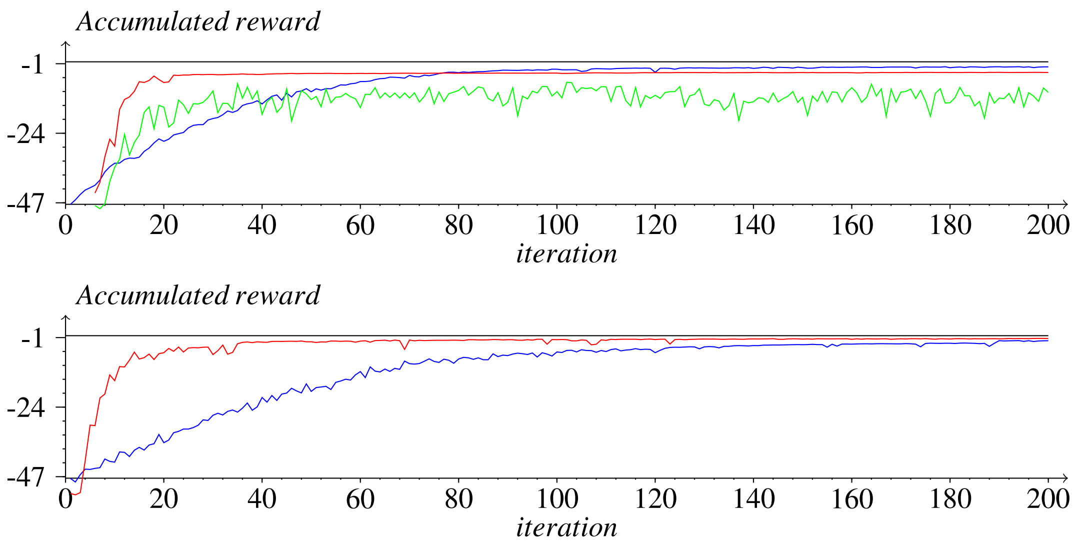


Figure 7.4: Accumulated cost during learning process - The black line represents the optimal cost while the red, and blue curves plot the accumulated cost when using approximate SARSA, and MSCARLA respectively. Top down the charts show the double integrator problem using tile coding, and Gaussian kernel based approximators.



7.1.2.1 Description

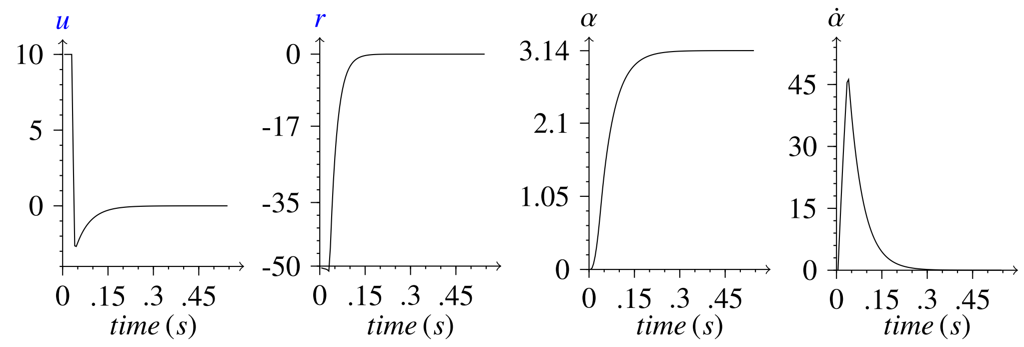
The model is a discrete-time abstraction of the electrical direct current (DC) motor introduced in (7.3). The discretization was performed using a sampling time of $\Delta t = 0.005s$. The state of the system is defined by the shaft angle and its angular velocity $x = \begin{bmatrix} \alpha & \dot{\alpha} \end{bmatrix}^T$. The shaft angle is measured in *rad* and it is bounded to $[0, 2\pi]$. The angular velocity is measured in rad/s an it is bounded to $[-16\pi, 16\pi]$. The controller must decide on the voltage feeded to the motor in an interval $[-10V, 10V]$. The goal is to stabilize the system at $x_g = \begin{bmatrix} \pi & 0 \end{bmatrix}^T$. Again, a negative quadratic reward function (7.2) is chosen to express this goal. The selected matrices are $Q = \begin{bmatrix} 5 & 0 \\ 0 & 0.01 \end{bmatrix}$ and $R = [0.01]$.

$$x_{k+1} = Ax_k + Bu_k$$
$$A = \begin{bmatrix} 1 & 0.0049 \\ 0 & 0.9540 \end{bmatrix}, B = \begin{bmatrix} 0.0021 \\ 0.8505 \end{bmatrix} \tag{7.3}$$

The simulation uses a time step of $\Delta t = 0.005s$. Every trial consists of starting the system at state $x_k = \begin{bmatrix} 0 & 0 \end{bmatrix}$ and running the system until either 160 decision steps has elapsed (i.e., 0.8 simulated seconds), or the goal state is reached (i.e., $|x_k - x_g| < \epsilon = 0.001$).

Again, the optimal policy is represented by the LQR. In this case, $K = \begin{bmatrix} 22.3607 & 1.1203 \end{bmatrix}$

Figure 7.5: Optimal solution for the DC motor problem - Trajectory when using optimal policy. From left to right: control action, reward, shaft angle and angular velocity.



so the optimal policy is $u^* = -K\left(x_k - x_g\right) = -\left[\begin{array}{cc} 22.3607 & 1.1203 \end{array}\right]\left(x_k - \left[\begin{array}{cc} \pi & 0 \end{array}\right]\right)$. Figure 7.5 shows the trajectory from x_0 to x_g using the optimal policy.

7.1.2.2 Results

The same parameter setting (as in the case of the double integrator problem) is used for the learners in this problem Figure 7.6 reports the results on the DC motor problem using a tile coding approximator. The solid black curve represents the optimal solution. The red, green and blue lines report the results when using approximate SARSA, CACLA, MSCARLA respectively. All charts plot the performance using the policy learned after 200 episodes. The trajectory sown for the CACLA is obtained with the greedy policy after learning. Top down, the charts represents the control action, the cost and both components of the state space: shaft angle and angular velocity. Observe that again the results are comparably good.

Figure 7.7 reports the results on the DC motor problem using a Gaussian kernel based approximator. The solid black curve represents the optimal solution. The red, and blue lines report the results when using approximate SARSA, and MSCARLA respectively. All charts plots the performance using the policy learned after 200 episodes. Top down, the charts represents the control action, the cost and both components of the state space: shaft angle and angular velocity. Again, both methods achieved a comparable performance.

Finally, figure 7.8 shows the accumulated cost (simple sum of all rewards from the initial state to the end of the episode) over learning. The black line represents the optimal cost while the red, green, and blue curves plot the accumulated cost when using approximate SARSA, CACLA, and MSCARLA respectively. Top down the charts show the DC motor problem

Figure 7.6: Learned policies to control the DC motor problem using a tile coding - Trajectory when using the policy learned by MSCARLA (blue line), approximate SARSA (red line), and CA-CLA (green line) using tile coding approximator. The black curve represents the optimal trajectory. Top down, the charts plots control action, cost, shaft angle and angular velocity.

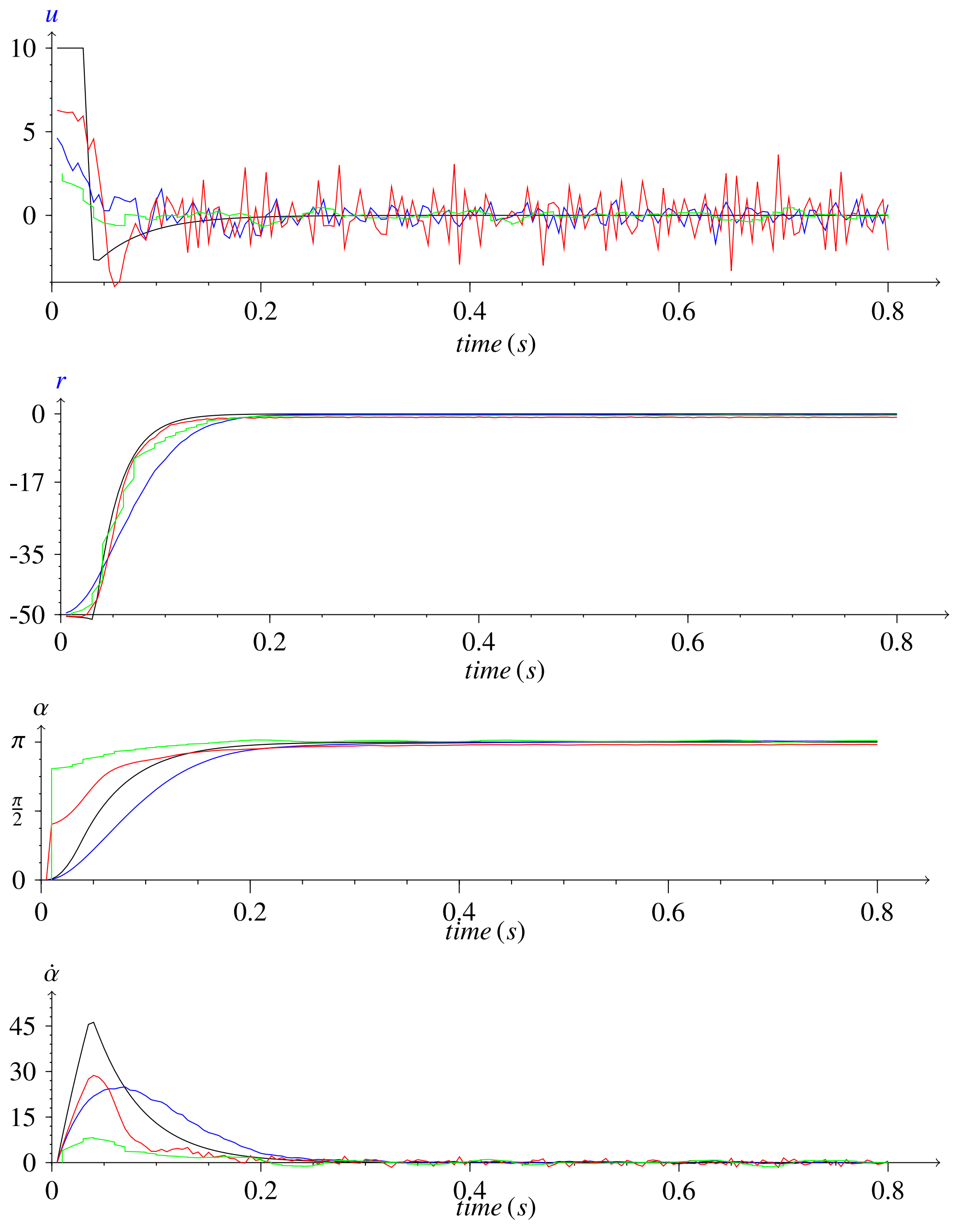


Figure 7.7: Learned policies to control the double integrator problem a using Gaussian kernel - Trajectory when using the policy learned by MSCARLA (blue line), and approximate SARSA (red line) using tile coding approximator. The black curve represents the optimal trajectory. Top down, the charts plots control action, cost, shaft angle and angular velocity.

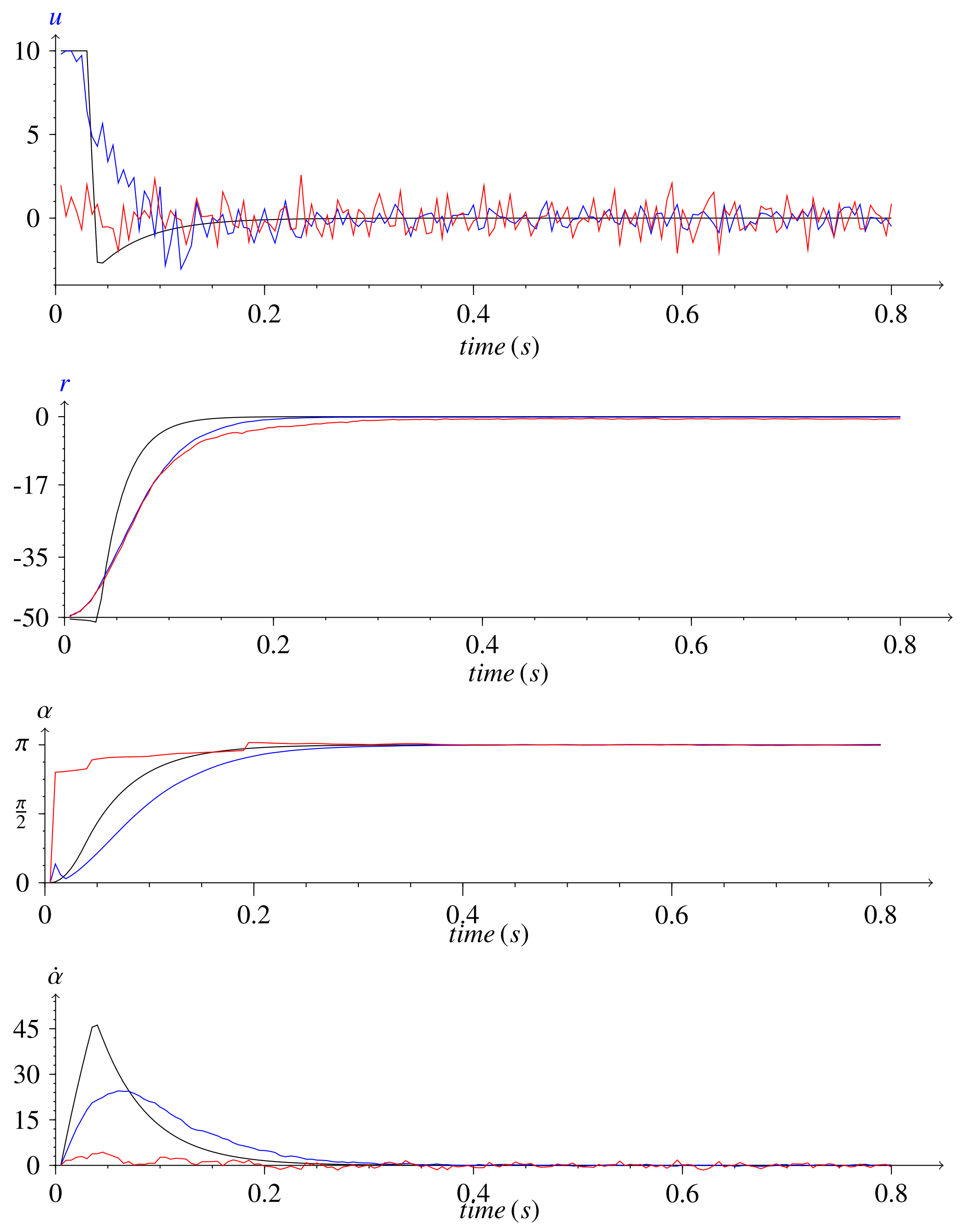
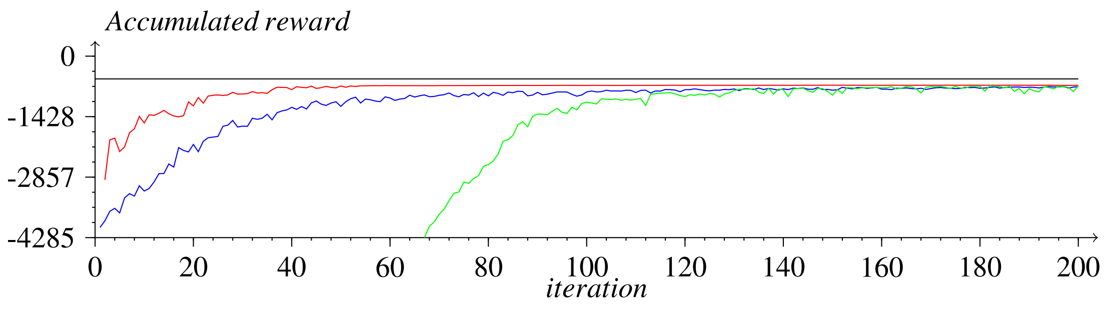
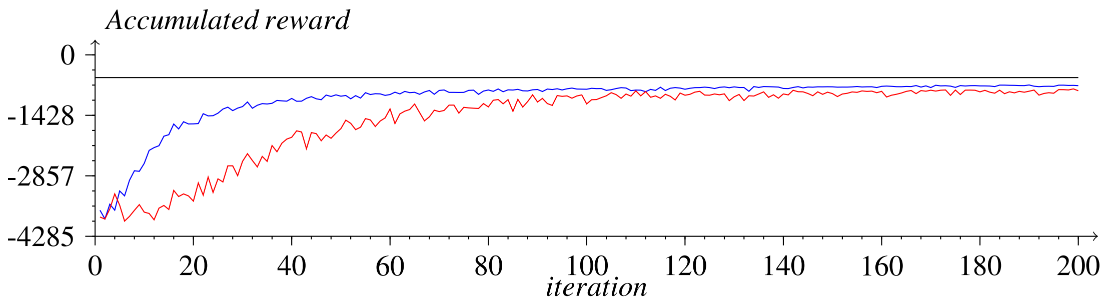


Figure 7.8: Accumulated cost during learning process - The black line represents the optimal cost while the red, green, and blue curves plot the accumulated cost when using approximate Q-learning, CACLA, and MSCARLA respectively. Top down the charts show the DC motor problem using tile coding, and Gaussian kernel based approximators.





using tile coding, and Gaussian kernel based approximators. Notice that the difference in the learning time is smaller than in the previous problem since there are no penalties involved in this process.

7.2 Real world applications

7.2.1 Autonomous tractor

The autonomous-tractor driving problem was introduced in section 6.2.2. In this section we come back to the same application. This process has non-linear dynamics so an algorithm such as the integral reinforcement learning (IRL) introduced in section 4.3.2 is not suited. The algorithm can still work if the dynamics are nearly linear. We tried the IRL technique with different parameter settings however the method was unfeasible. After the first update of the policy the controller became unstable driving the tractor away from the trajectory. As written before, this was not unexpected. The possible solution to this problem is to reduce the sampling time. After reducing this sampling time significantly (from 0.2 to 0.005 seconds) in simulation, we started obtaining a stable performance of the controller. Unfortunately, this is a change we cannot afford in the real set-up. The minimum possible sampling time that we can use is 0.2 seconds. Instead, we use a more flexible technique such as the MSCARLA.

Before, the state information was used to construct a linear controller by means of a gain matrix. That was a first step in using CARLA for control in a MDP. The control action was given as the sum of the actions predicted by the feed-forward and feed-back controllers. The feedback controller that was designed for correcting the error over time is now substituted by an MSCARLA learner. By doing so, we eliminate a long process of model identification in order to create the model-based gain matrix. Still, we are keeping some model information (the feed-forward controller) which is much simpler to obtain and that will shorten the learning process significantly. We could have substituted both and let the learner to take over on its own and it would also have converged to a good policy but the learning time for this approach would have been unpractical.

7.2.1.1 Description

The state space of the learner is defined using most of the variables used before for the controllers: error y, its first derivative $\dot{y}$, and the curvature of the trajectory Rt. The control action is the correction to be added to the action selected by the feed-forward controller so the steering angle $\theta = u_{ff} + u_{MSCARLA}$ where u_{ff} is the action selected by the feed-forward controller and $u_{MSCARLA}$ is the action selected by the learner. The learner takes an action at every time-step during the driving process leading to a new state and a reward. The one-step reward function is a negative quadratic function of the current state:

$$r_{k+1} = -\left(x_k^T Q x_k + u_k^T R u_k\right)$$

where:

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad R = [0]$$

Every time that the tractor deviates from the reference trajectory in more than 2 meters the trajectory is restarted from the origin and the learner is punished with -20.

7.2.1.2 Results

Again, all results are obtained in simulation. Figure 7.9 reports the error made while driving along the trajectory. The dashed lines represents the maximum error when using the PID controller with the optimal slip factor (0.6). The solid blue line shows the error when using the MSCARLA technique. The results are obtained using tile coding as the function approximator with the same parameter settings as in the previous section. Notice that the learner clearly outperforms the PID. Also notice that although the learning method needs to drive for more than one hour before it starts performing nearly optimal, all the time and effort to obtain a model (for the PID) is completely unnecessary now. If we compare this approach with the one used previously in section 6.2.2, we can see that the results obtained are more accurate and the learning process is not considerably lengthened.

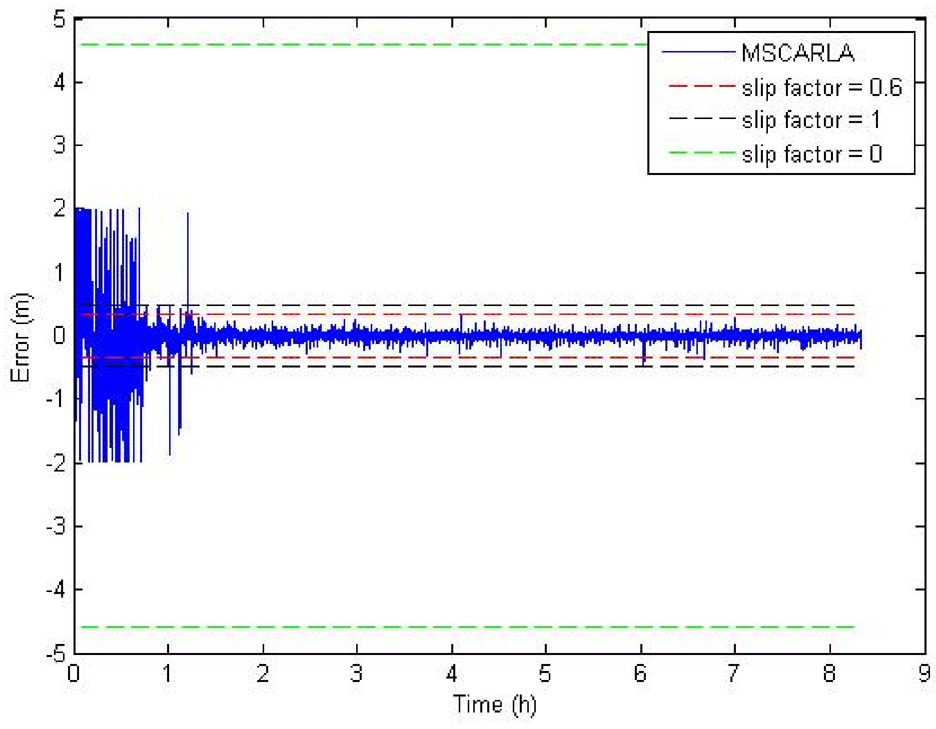


Figure 7.9: Error during the learning process using MSCARLA - The discontinuous lines represent the maximum and minimum errors observed when using manually predefined slip factors for the PID controller. The optimal performance is achieved by using a slip factor of 0.6. When using MSCARLA, the tractor has a smaller deviation from the target trajectory.

Figure 7.10 shows the accumulated error during every turn (half of the eight shaped trajectory). Notice that during the learning process a higher error is accumulated since we are not using any model information for learning the correction. Still, the accumulated error after 50 turns becomes lower than the one obtained by the best PID controller.

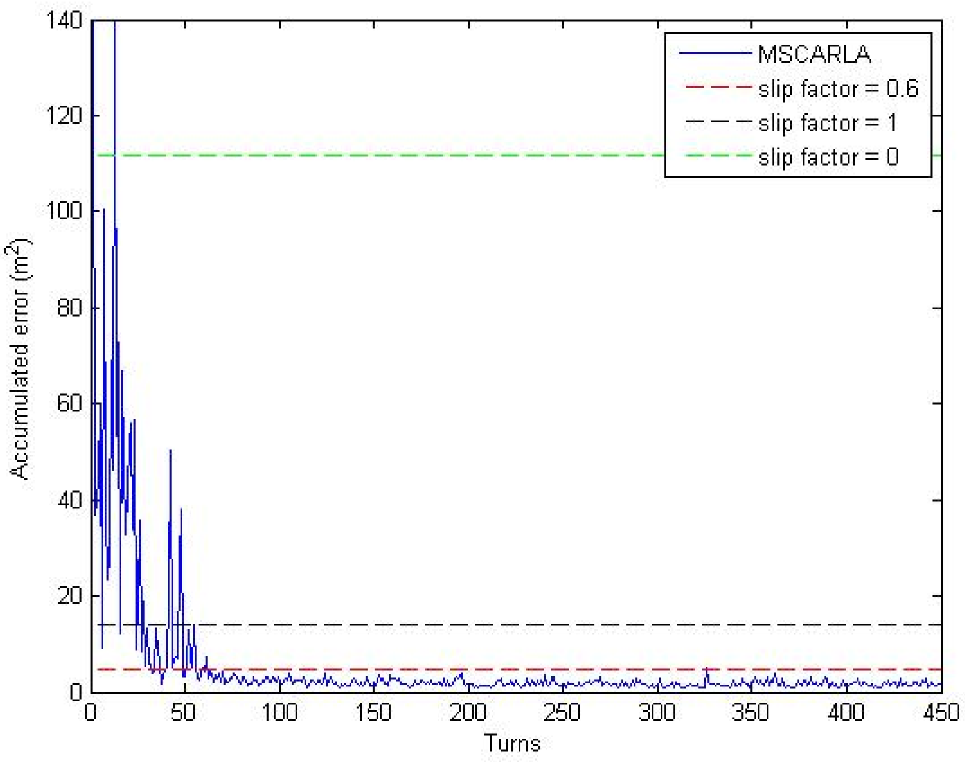


Figure 7.10: Accumulated error during the learning process using MSCARLA - The discontinuous lines represent the maximum and minimum errors observed when using some manually predefined slip factors for the PID controller. The optimal performance is achieved by using a slip factor of 0.6. When using MSCARLA, a considerably lower accumulated error is obtained at every turn after the learning process.

We close the discussion on this experiment by showing the learned policy. The idea of RL is to finely estimate the policy around the optimal trajectory and more loosely as the state gets farer from the optimum. For this reason you may expect the policy to be well defined around the steady-state (at zero error and derivative of the error) and poorer defined as the state is farer from this point. The state space has three dimensions. The curvature is fixed during the straight paths, during the turn to the right and during the turn to the left though.

Figures 7.11, 7.12, and 7.13 show a contour representation of the average action decided by the MSCARLA controller after learning when turning right, going straight, and turning left respectively.

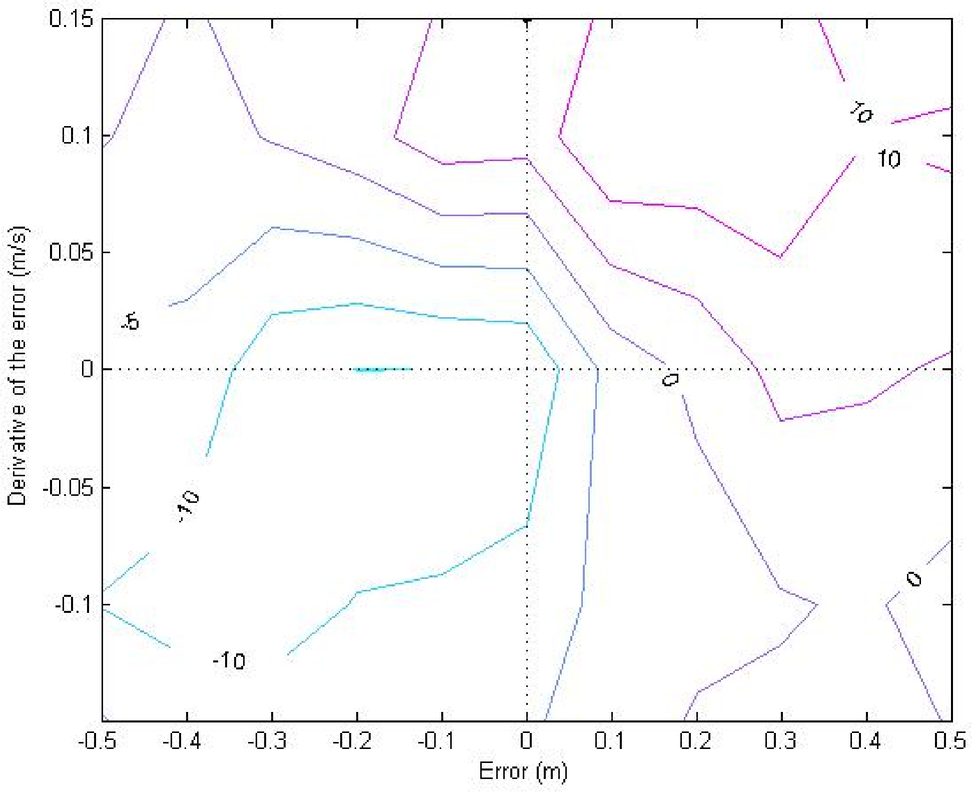


Figure 7.11: Policy for driving the tractor when turning right - When there is a negative error and a negative derivative of the error (meaning the error is becoming even greater in the negative direction), a correction of $-10°$ is necessary. As the error and its derivative grows positive the correction grows up till $10°$. Notice that when both, the error and its derivative, are inverse in terms of sign no correction is necessary since the absolute value of the error is decreasing already.

7.2.2 Autonomous tractor plus an implement

The final goal when controlling the tractor is to perform some agricultural task. For this, we also add an implement that is supposed to perform the task, e.g., harvesting crops or plowing a field. The trajectory followed by the implement is mostly governed by the tractor but it still has some movement freedom. It has been provided with an actuator for controlling its positioning

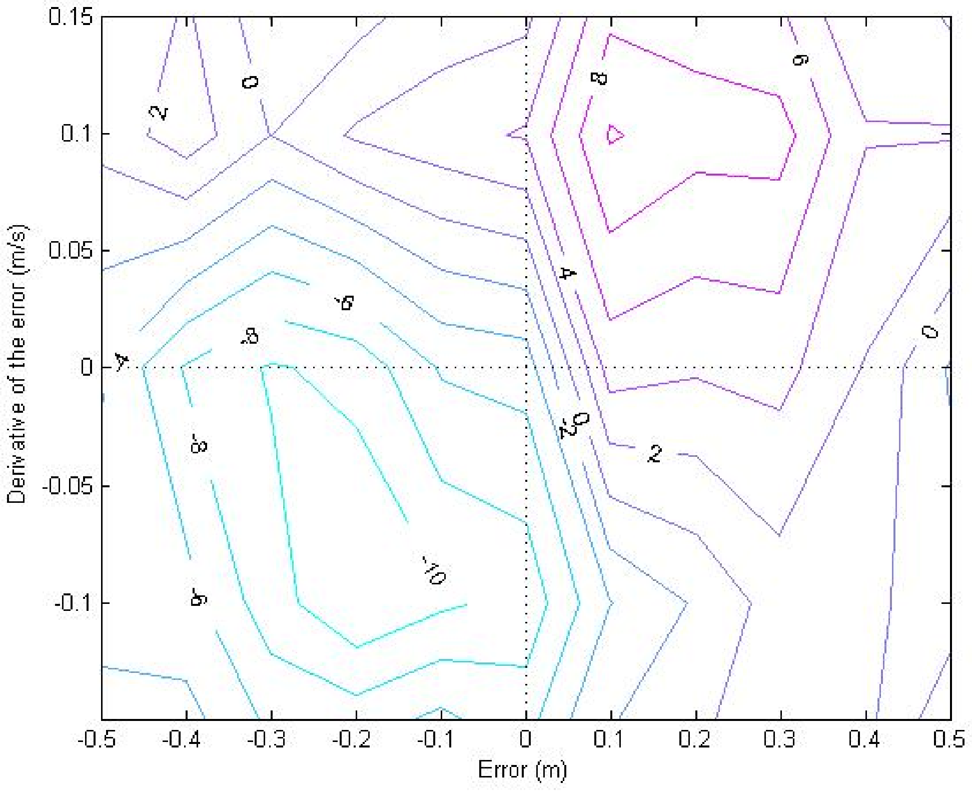


Figure 7.12: Policy for driving the tractor when going straight - When there is a negative error and a negative derivative of the error (meaning the error is becoming even greater in the negative direction), a correction of $-10°$ is necessary. As the error and its derivative grows positive the correction grows up. Notice that when both, the error and its derivative, are inverse in terms of sign no correction is necessary since the absolute value of the error is decreasing already.

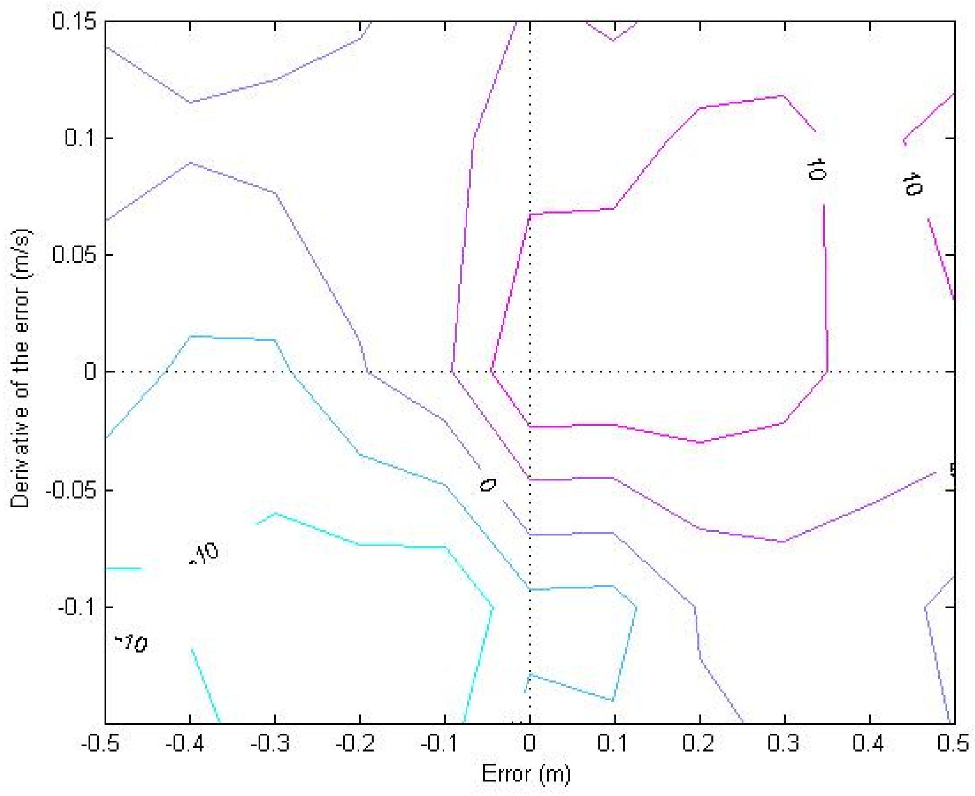


Figure 7.13: Policy for driving the tractor when turning left - When there is a negative error and a negative derivative of the error (meaning the error is becoming even greater in the negative direction), a correction of $-10°$ is necessary. As the error and its derivative grows positive the correction grows up till $10°$. Notice that when both, the error and its derivative, are inverse in terms of sign no correction is necessary since the absolute value of the error is decreasing already.

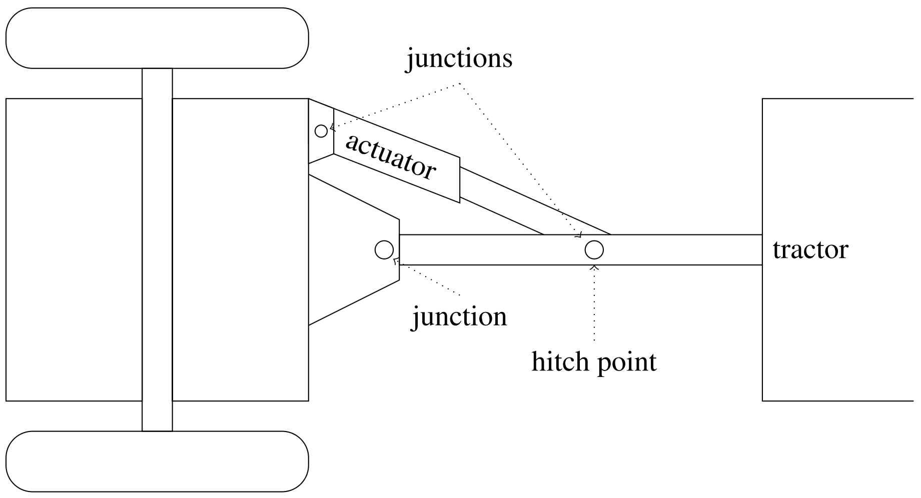


Figure 7.14: Controlling an implement - The implement is attached to the tractor and by means of the actuator it can control its positioning with respect to the tractor.

with respect to the tractor.

7.2.2.1 Description

Figure 7.14 shows an sketch of the set-up. The actuator can change the length of the side joint making the implement to slide at a given angle with respect to the tractor.

In this new set-up, on the straight paths, it is perfectly possible for both, the tractor and the implement, to follow the same trajectory. It is impossible for the implement to follow exactly the same trajectory that the tractor is following during the turns though. For this reason, if we are interested on the implement to track a reference trajectory we need to modify the one being the reference for the tractor. Figure 7.15 shows the trajectories selected for the experiments.

7.2.2.2 Results

We now have two subsystems to control: the tractor and the implement. Each one has its own state information based on its own position. We use two exploring selfish continuous action reinforcement learning automatons (ESCARLAs) for controlling them. The reward function and state description remains unchanged with respect to the previous setup (without the implement). Keep in mind that the state of the tractor and the implement depend on their own positions which will most of the time have a different relative position to the reference signal. The action space for the tractor remains the same while the implement states range from the

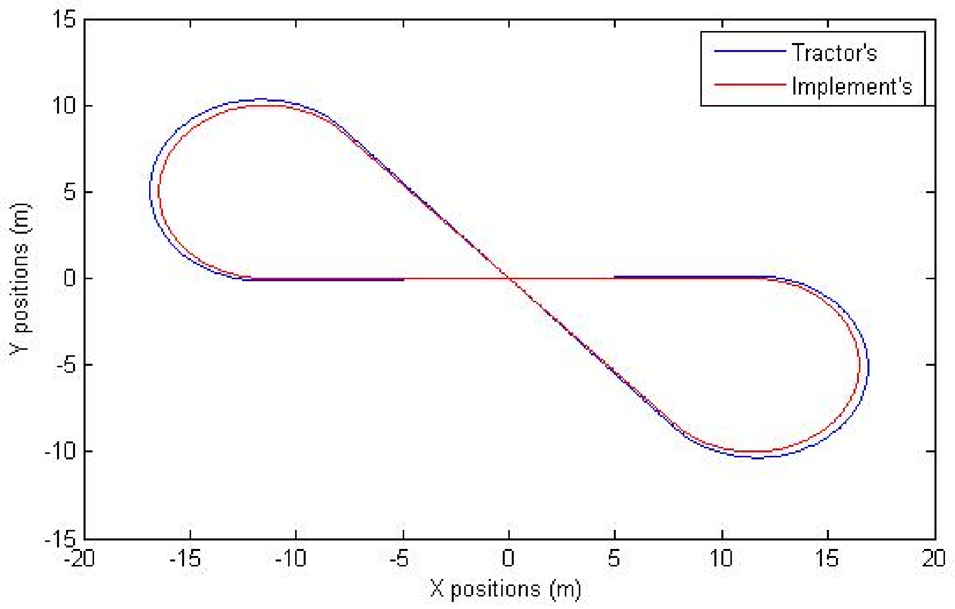


Figure 7.15: A reference trajectory for the tractor and the implement - The trajectory for the tractor on the turns is wider than the one for the implement making possible the implement to track it.

minimum to the maximum length of the side joint. The same parameter settings, that has been used throughout the chapter, are used for both ESCARLAs. Figure 7.16 show the error of both, the tractor and the implement, while using MSCARLA for learning the control policy. Figure 7.17 shows the accumulated error during every turn. The implement's performance is affected by the tractor's. Still, both controllers are able to learn in a shared process.

7.3 Summary

This chapter demonstrates how to use the MSCARLA learner for controlling multi-state problems. First, some benchmark problems from RL literature are used for showing a comparison of the different existing techniques and how the changes proposed in this dissertation enables the method to deal with state information. The application used as a demonstration is the autonomous tractor introduced in section 6.2.2, but instead of using the CARLA technique for optimizing a linear controller we directly learn the policy to map actions to the observed state by using MSCARLA.

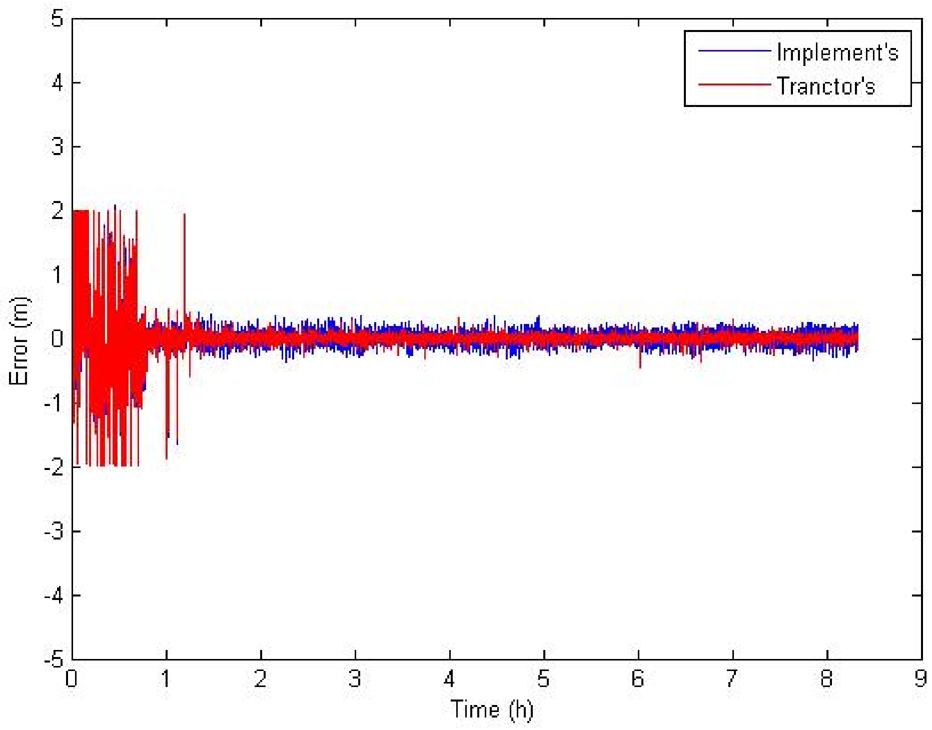


Figure 7.16: Error during the learning process using MSCARLA - The red curve represents the error of the tractor and the blue the implement's.

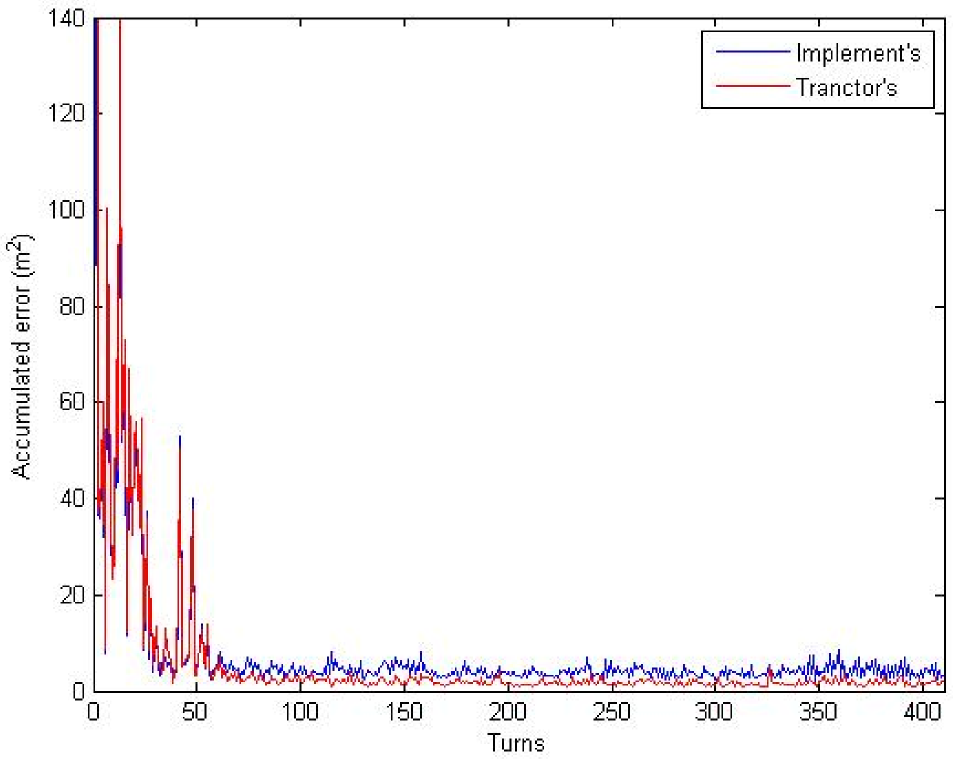


Figure 7.17: Accumulated error during the learning process using MSCARLA - The red curve represents the error of the tractor and the blue the implement's.

Chapter 8

Summary

Reinforcement learning (RL) has been used as an alternative to model based techniques for learning optimal controllers. The central theme in RL research is the design of algorithms that learn control policies solely from the observation of transition samples or trajectories, which are collected by interaction with the system. Such controllers have become part of our daily life by being present in some home appliances or really complex machines in industry.

This dissertation introduces an RL algorithm to control production machines, which is based on simple continuous action reinforcement learning automaton (CARLA). For better understanding the results we first present some background information on the general framework of bandit applications, Markov decision processs (MDPs), and RL for solving problems in discrete and continuous state-action spaces.

In a bandit setting, the task presented to the learner is to maximize the long-run pay-off for stateless systems. The framework is described for discrete and continuous settings and the strategies for solving such problems are also presented. In addition, a description of discrete games where two or more learners interact in order to maximize their long-run profit is given as well. Exploring selfish reinforcement learning (ESRL) is presented as an exploration method for a set of independent learners playing a repeated discrete action game.

MDPs are formally defined in order to facilitate introducing RL. We first present the deterministic case. Afterwards, the stochastic case is considered. We also discuss the notion of optimality for both cases. Most RL algorithms focus on discrete action-state spaces. The two well-known classes of RL are presented: value and policy iteration. For each class, the most popular algorithms in the literature are explained. The standard discrete RL techniques are provided with function approximators to represent the value functions over the infinite action-state space. Two classes of approximators are described: parametric and non-parametric. The former class defines a fixed set of parameters in advance while in the latter class the set of parameters of the approximators is adapted to the available data at every instant during learning. After formally introducing the function approximators, the three RL architectures (approximate value iteration, approximate policy iteration, and approximate policy search) are extended and well-known algorithms are provided.

After discussing the background information we present the extensions we propose to the CARLA algorithm in order to make the learning feasible for the applications tackled in this research. The first change to the original method extends the mathematical analysis of the method and allows us to avoid the numerical integration. This enables the method to run faster in simple processors attached to the machines that are to be controlled. After obtaining this result, the analysis of the convergence of the method is performed in bandit applications where only one learner is present. This analysis makes it possible to derive a simple way of tuning the spreading rate parameter of the method, which in addition to the reward transformation we propose, improves the performance from the sampling point of view. The convergence analysis is also performed for game settings, where multiple learners interact. Based on this analysis, we demonstrate the necessity of a better exploration protocol and the extension of ESRL to continuous action spaces is introduced. Finally, in order to enable the CARLA method for optimizing the control action in a Markovian process, we combine the learning technique with function approximators over the state space.

After introducing our methods, we separate the applications in two groups, state-less applications and multi-state applications. First, some test cases are shown in order to demonstrate the performance of the learning methods derived in this dissertation. Afterwards, we present the real world applications and evaluate the algorithm's performance on these realistic problems.

8.1 Contributions

The main contributions of the dissertation are the theoretical changes to the CARLA method and the demonstration of how to use the method for controlling production machines. We first enumerate the theoretical contributions:

1. If we want to control real production machines we can only rely on simple processors attached to these machines. This restriction, in addition to the limited time that the

learners have for responding with the appropriate control action, leads us to the first contribution being a significant reduction of the computational cost of the method. This reduction is carried out by analytical derivations of the learning rule to avoid numerical integration.

2. Since collecting data from a real machine is very costly in time (and most probably in effort as well) it is also necessary to use a method that can learn as fast as possible using an optimal exploration of the action space. The second contribution is an analysis of the convergence time of the method and a procedure to adjust the spreading rate of the method (which is a parameter controlling the exploration ratio) that, in combination with the reshaping of the reward signal, leads to an improvement of convergence to the optimal solution in terms of sampling time.

3. The third contribution is aimed at distributed control applications. Inspired by the ESRL method for coordinating discrete learning automata, with limited communication, in order to guarantee convergence to the global optimum (escaping local optima) we introduce exploring selfish continuous action reinforcement learning automaton (ESCARLA) which is a version of ESRL but for continuous action spaces. This change improves the global convergence (in terms of ratio of convergence to the global maximum) of the method in distributed control set-ups.

4. Finally, a combination of the CARLA method with function approximators in order to deal with multi-state applications is presented. This combination allows the learning method to deal with more complex tasks with a state feedback signals.

We address a set of control problems, concerning production machines. The practical contributions are enumerated bellow:

5. State-less applications

   (a) The first application is a basic control problem: controlling a linear motor. The term linear comes from the direction of the force produced by the motor, not from its dynamics. This stationary continuous action bandit example is representative of several production machines where we need to control a motor with information obtained from limited sensors. Our method was able to control this setup relatively fast.

   (b) A much more complex task is demonstrated by optimizing a linear controller for driving an autonomous tractor. The RL technique is used here for learning the optimal parameters of the linear controller. In this problem, our method is able to tune the parameters to obtain a performance comparable with the best manual tuning.
   (c) We also show how to control a wire winding machine as a non-stationary example. In this example, out method is capable to adapt to the changes in the process over time.
   (d) The last state-less application is in a game setting. It relates to a distributed hydrologic system where local controllers exist and we want to decide on their set-point for an optimal performance.

6. Multi-state applications

   (a) To finalize the applications we present a solution of the autonomous driving of the tractor not by means of optimizing a linear controller but using the multi-state learning scheme that we present in this dissertation. Our method performed much better than the optimized linear controller.

8.2 Future work

This dissertation presents several methods that allow the application of a simple learner to control problems. Still, some new research directions arise from the results obtained so far.

- **Smarter initialization of the policy**: In section 7.2.1 we use multi-state continuous action reinforcement learning automaton (MSCARLA) to learn a near optimal policy completely from the exploration of the state-action space. The learning ignored any any prior knowledge. As a consequence, the learning process is not robust at all. During a first phase, due to the explorative behavior of the learner, the tractor deviates too much from the target trajectory making the learning process very long and unstable. This problem could be avoided by allowing the learner to interact with a stable controller (not necessarily optimal) to guide the exploration during learning. Different ideas can be tried to pursuit this goal. The learner could most often select the action chosen by the controller at the beginning of learning when it has a lot to learn from the controller. Over time, when it starts “feeling” more confident of its actions (meaning that the actions does not make the driving unstable anymore) it can agree to select the action of the controller less often, until it finally ignores it. Although this can shorten the learning process significantly and may avoid undesired behavior of the learner it can also bias the learner too much in the direction of the suboptimal controller. Still, it is an open direction to explore. Another possible solution is to collect some data in advance and then use a batch method to initialize the policy of the learner. The learner can then start using such a policy on the real problem (driving the tractor in the field) and finish the learning process.

- **Taking penalties into account**: The CARLA algorithm uses a reward-inaction scheme for updating the policy. This means that the probability for the selected action over time is increased no matter the nature of the outcome. Bad rewards lead to no change in the policy or even a slight increase. Using a reward-penalty scheme means that bad rewards actually lower the probability of selecting the action in the future. Including penalties in the CARLA learning rule requires to carefully control the extension of the neighborhood around the action that is going to be penalized to avoid overlapping between the bonus obtained when selecting a good action and penalties when selecting an action close to good ones. Further derivations have to be made in order to ensure, without a considerable cost, that the probability function does not turn negative for any action.

- **Further applications in Cuban industry**: Optimal control is little used in Cuban industry. There exist cheap means to integrate simple processors to machines. We also count with the knowledge required to start projects in order to optimize production. Agriculture is getting more and more attention in Cuba and there are plenty of applications that can be assisted with optimal control. A clear example was the autonomous tractor driving but other applications are possible, e.g., some recognition drones could be used to probe information from the air or to spray products. Other examples can be found in repetitive production machines involving linear motors or DC motors.

References

ANTOS, A., MUNOS, R. & SZEPESVÁRI, C. (2008). Fitted q-iteration in continuous action-space mdps. *Advances in Neural Information Processing Systems*. 44

AUER, P., CESA-BIANCHI, N. & FISCHER, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, **47**, 235–256. 10, 12

BARTO, A.G., SUTTON, R.S. & ANDERSON, C.W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, **13**, 833–846. 37

BAXTER, J. & BARTLETT, P.L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*. 47

BELLMAN, R. (2003). *Dynamic Programming*. Dover Publications. 1

BERENJI, H.R. & KHEDKAR, P. (1992). Learning and tuning fuzzy logic controllers through reinforcements. *IEEE Transactions on Neural Networks*, **3**. 37

BERENJI, H.R. & VENGEROV, D. (2003). A convergent actor-critic-based frl algorithm with application to power management of wireless transmitters. *IEEE Transactions on Fuzzy Systems*, **11**. 37

BERTSEKAS, D. (1995). Dynamic programming and optimal control. *Athena Scientific*. 112

BERTSEKAS, D. (2007). *Dynamic Programming and Optimal Control*, vol. 2. Athena Scientific, 3rd edn. 37

BERTSEKAS, D. & TSITSIKLIS, J. (1996). *Neuro-Dynamic Programming*. Athena Scientific. 28, 40

BETHKE, B., HOW, J. & OZDAGLAR, A. (2008). Approximate dynamic programming using support vector regression. In *In Proceedings 47th IEEE Conference on Decision and Control (CDC-08)*, 3811–3816. 43

BORKAR, V. (2005). An actor-critic algorithm for constrained markov decision processes. *Systems & Control Letters*, **54**. 37

BREIMAN, L. (2001). Random forests. *Machine Learning*, 5–32. 43

BREIMAN, L., FRIEDMAN, J., STONE, C.J. & OLSHEN, R. (1984). *Classification and Regression Trees*. Wadsworth International. 43

BUBECK, S., MUNOS, R., STOLTZ, G. & SZEPESVÁRI, C. (2011). X-Armed bandits. *Journal of Machine Learning Research*, **12**, 1655–1695. xvii, 21, 22, 83

BUŞONIU, L., BABUŠKA, R., DE SCHUTTER, B. & ERNST, D. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press. xvii, 3, 27, 33, 36, 39, 40, 43, 45, 46, 47, 49, 53, 111, 115

CASTELLETTI, A., PIANOSI, F. & RESTELLI, M. (2012). Tree-based fitted q-iteration for multi-objective markov decision problems. In *IJCNN*, 1–8, IEEE. 106

CLAUS, C. & BOUTILIER, C. (1998). The dynamics of reinforcement learning in cooperative multiagent systems. In *In Proceedings of National Conference on Artificial Intelligence (AAAI-98*, 746–752. 16

COPE, E. (2009). Regret and convergence bounds for immediate-reward reinforcement learning with continuous action spaces. *IEEE Transactions on Automatic Control*, **54**, 1243–1253. 19

CRISTIANINI, N. & SHAWE-TAYLOR, J. (2000). *An Introduction to Support Vector Machines and Other*

Kernel-Based Learning Methods. Cambridge University Press. 43

Degris, T., Pilarski, P. & Sutton, R. (2012). Model-free reinforcement learning with continuous action in practice. In *In Proceedings of the 2012 American Control Conference*. 39

Deisenroth, M.P., Rasmussen, C.E. & Peters, J. (2009). Gaussian process dynamic programming. *Neurocomputing*, 1508–1524. 43

Engel, Y., Mannor, S. & Meir, R. (2003). Bayes meets bellman: The gaussian process approach to temporal difference learning. In *In Proceedings 20th International Conference on Machine Learning (ICML-03)*, 154–161. 43

Engel, Y., Mannor, S. & Meir, R. (2005). Reinforcement learning with gaussian processes. In *In Proceedings 22nd International Conference on Machine Learning (ICML-05)*, 201208. 43

Ernst, D. (2005). Selecting concise sets of samples for a reinforcement learning agent. In *In Proceedings 3rd International Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS-05)*. 43, 44

Ernst, D., Glavic, M., Geurts, P. & Wehenkel, L. (2006a). Approximate value iteration in the reinforcement learning context. application to electrical power system control. *International Journal of Emerging Electric Power Systems*. 43

Ernst, D., Stan, G.B., GonÇalves, J. & Wehenkel, L. (2006b). Clinical data based optimal sti strategies for hiv: A reinforcement learning approach. In *In Proceedings 45th IEEE Conference on Decision & Control*. 44

Farahmand, A., Ghavamzadeh, M., Szepesvári, C. & Mannor, S. (2009a). Regularized fitted q-iteration for planning in continuous-space markovian decision problems. In *In Proceedings 2009 American Control Conference (ACC-09)*, 725–730. 43

Farahmand, A., Ghavamzadeh, M., Szepesvári, C. & Mannor, S. (2009b). Regularized fitted q-iteration for planning in continuous-space markovian decision problems. In *In Proceedings 2009 American Control Conference (ACC-09)*. 44

Gintis, H. (2009). *Game Theory Evolving: A Problem-Centered Introduction to Modeling Strategic Interaction (Second Edition)*. Princeton University Press, 2nd edn. 14

Gordon, G. (2001). Reinforcement learning with function approximation converges to a region. In *In Advances in Neural Information Processing Systems*. 46

Hilgard, E. & Bower, G. (1966). *Theories of learning*. Appleton-Century-Crofts. 13

Hofmann, T., Schölkopf, B. & Smola, A.J. (2008). Kernel methods in machine learning. *Annals of Statistics*. 43

Horiuchi, T., Fujino, A., Katai, O. & Sawaragi, T. (1996). Fuzzy interpolation based q-learning with continuous states and actions. In *In Proceedings 5th IEEE International Conference on Fuzzy Systems (FUZZ-IEEE-96)*, 594–600. 44

Howell, M. & Best, M. (2000). On-line pid tuning for engine idle-speed control using continuous action reinforcement learning automata. *Control Engineering Practice*. 3, 95

Howell, M.N., Frost, G.P., Gordon, T.J. & Wu, Q.H. (1997). Continuous action reinforcement learning applied to vehicle suspension control. *Mechatronics*. xvii, 2, 3, 23, 24, 95

Jaakkola, T., Jordan, M. & Singh, S. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, **6**, 1185–1201. 34

Jodogne, S., Briquet, C. & Piater, J.H. (2006). Approximate policy iteration for closed-loop learning of visual tasks. In *In Proceedings 17th European Conference on Machine Learning (ECML-06)*, 210–221. 43

JOUFFE, L. (1998). Fuzzy inference system learning by reinforcement methods. *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews*. 44

JUNG, T. & POLANI, D. (2007). Kernelizing lspe(λ). In *In Proceedings 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL-07)*. 43

KAELBLING, L., LITTMAN, M. & MOORE, A.W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*. 28

KAPETANAKIS, S. & KUDENKO, D. (2002). Reinforcement learning of coordination in cooperative multi-agent systems. 326–331. 16

KAPETANAKIS, S., KUDENKO, D. & STRENS, M. (2003a). Learning to coordinate using commitment sequences in cooperative multiagent-systems. In *in Proceedings of the Third Symposium on Adaptive Agents and Multi-agent Systems (AAMAS-03*, 2004. 16

KAPETANAKIS, S., KUDENKO, D. & STRENS, M. (2003b). Learning to coordinate using commitment sequences in cooperative multiagent-systems. In *in Proceedings of the Third Symposium on Adaptive Agents and Multi-agent Systems (AAMAS-03*, 2004. 17

KONDA, V.R. & TSITSIKLIS, J.N. (2003). On actor-critic algorithms. *SIAM Journal on Control and Optimization*, **42**. 37

LAGOUDAKIS, M.G. & PARR, R. (2003). Reinforcement learning as classification: Leveraging modern classifiers. In *In Proceedings 20th International Conference on Machine Learning (ICML-03)*, 424431. 43

LEWIS, F. & SYRMOS, V. (1995). *Optimal control*. John Wiley. 49

LIN, L.J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*. 44

MARBACH, P. & TSITSIKLIS, J.N. (2003). Approximate gradient methods in policyspace optimization of markov reward processes. *Discrete Event Dynamic Systems: Theory and Applications*, **13**, 111–148. 47

MELO, F.S., MEYN, S.P. & RIBEIRO, M.I. (2008). An analysis of reinforcement learning with function approximation. In *In Proceedings 25th International Conference on Machine Learning (ICML-08)*. 44, 46

MUNOS, R. (2006). Policy gradient in continuous time. *Journal of Machine Learning Research*, **7**, 771–791. 47

MUNOS, R. & SZEPESVÁRI, C. (2008). Finite time bounds for fitted value iteration. *Journal of Machine Learning Research*. 44

MURPHY, S. (2005). A generalization error for q-learning. *Journal ofMachine Learning Research*. 44

NAKAMURA, Y., MORIA, T., SATOC, M. & ISHIIA, S. (2007). Reinforcement learning for a biped robot based on a cpg-actor-critic method. *Neural Networks*, **20**. 37

NG, A.Y., HARADA, D. & RUSSELL, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, 278–287, Morgan Kaufmann. 38

NOWÉ, A. (1992). A self-tuning robust fuzzy controller. *Microprocessing and Microprogramming*, **35**, 719 – 726. 44

NOWÉ, A., VRANCX, P. & DE HAUWERE, Y.M. (????). *Reinforcement Learning: State-of-the-Art*, chap. Game Theory and Multi-agent Reinforcement Learning, 441–470. 15

ORMONEIT, D. & SEN, S. (2002). Kernel-based reinforcement learning. *Machine Learning*, 161–178. 43, 44

PANAIT, L., SULLIVAN, K. & LUKE, S. (2006). Lenient learners in cooperative multiagent systems. In *Proceedings of the fifth international joint conference* *on Autonomous agents and multiagent systems*, AAMAS '06, 801–803, ACM, New York, NY, USA. 16

Parzen, E. (1960). *Modern Probability Theory And Its Applications*. Wiley-interscience. 24, 57

Peters, J., Vijayakumar, S. & Schaal, S. (2003). Reinforcement Learning for Humanoid Robotics. In *Conference on Humanoid Robots*. 47

Puterman, M.L. (1994). *Markov Decision Processes-Discrete Stochastic Dynamic Programming*. Wiley. 27

Rasmussen, C.E. & Williams, C.K.I. (2006). *Gaussian Processes for Machine Learning*. MIT Press. 43

Riedmiller, M. (2005). Neural fitted q-iteration – first experiences with a data efficient neural reinforcement learning method. In *In Proceedings 16th European Conference on Machine Learning (ECML-05)*. 44

Riedmiller, M., Peters, J. & Schaal, S. (2007). Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *In Proceedings 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL-07)*, 254–261. 47

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the AMS*, **58**, 527–535. 9

Rodríguez, A., Ábalo, R.G. & Nowé, A. (2011). Continuous action reinforcement learning automata - performance and convergence. In *ICAART (2)*, 473–478. 23, 55

Rodriguez, A., Gagliolo, M., Vrancx, P., Grau, R. & Nowé, A. (2011). Improving the performance of Continuous Action Reinforcement Learning Automata. 55

Rodríguez, A., Vrancx, P., Grau, R. & Nowé, A. (2012). An rl approach to common-interest continuous action games. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '12, 1401–1402, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. 55

Rodriguez, A., Vrancx, P., Grau, R. & Nowé, A. (2012). An rl approach to coordinate exploration with limited communication in continuous action games. In *ALA-2012*, 17–24. 55

Rückstiess, T., Sehnke, F., Schaul, T., Wierstra, D., Sun, Y. & Schmidhuber, J. (2010). Exploring parameter space in reinforcement learning. *Paladyn*, **1**, 14–24. 47

Rummery, G.A. & Niranjan, M. (1994). On-line q-learning using connectionist systems. Tech. rep. 35

Santamaria, J.C., Sutton, R. & Ram, A. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, **2**, 163–218. 42, 46, 111

Schölkopf, B.C., B. & Smola, A. (1999). *Advances in Kernel Methods: Support Vector Learning*. MIT Press. 43

Shawe-Taylor, J. & Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press. 43

Sherstov, A. & Stone, P. (2005). Function approximation via tile coding: Automating parameter choice. In *In Proceedings 6th International Symposium on Abstraction, Reformulation and Approximation (SARA-05)*. 44

Singh, S.P., Jaakkola, T. & Jordan, M.I. (1995). Reinforcement learning with soft state aggregation. *Advances in Neural Information Processing Systems*. 44

Smola, A.J. & Scholkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing*, 199–222. 43

Stengel, R. (1994). Optimal control and estimation. *Dover Publications*. 112

SUTTON, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, **3**, 9–44. 41

SUTTON, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *In Advances in Neural Information Processing Systems*. 46

SUTTON, R. & BARTO, A. (1998). *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA. 1, 27, 29, 33, 34, 35, 45, 113

SUTTON, R., BARTO, A. & WILLIAMS, R. (1992). Reinforcement learning is adaptive optimal control. *IEEE Control Systems Magazine*, **2**, 19–22. 33

SUTTON, R.S., MCALLESTER, D.A., SINGH, S.P. & MANSOUR, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *In Advances in Neural Information Processing Systems*, 1057–1063. 37, 47

SZEPESVÁRI, C. & MUNOS, R. (2005). Finite time bounds for sampling based fitted value iteration. In *In Proceedings 22nd International Conference on Machine Learning (ICML-05)*. 44

SZEPESVÁRI, C. & SMART, W.D. (2004). Interpolation-based q-learning. In *In Proceedings 21st International Conference on Machine Learning (ICML-04)*, 791–798. 44

TESAURO, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, **8**, 257–277. 44

THATHACHAR, M.A.L. & SASTRY, P.S. (2004). *Networks of Learning Automata: Techniques for Online Stochastic Optimization*. Kluwer Academic Publishers. xvii, 2, 12, 13, 19, 20

TSETLIN, M. (1962). The behavior of finite automata in random media. *Avtomatika i Telemekhanika*. 12

TSITSIKLIS, J. (1994). Asynchronous stochastic approximation and q-learning. *Machine Learning*, **1**, 185–202. 34

TUYLS, K., MAES, S. & MANDERICK, B. (2002). Q-learning in simulated robotic soccer – large state spaces and incomplete information. In *In Proceedings 2002 International Conference on Machine Learning and Applications (ICMLA-02)*, 226–232. 44

VAN HASSELT, H. (2012). *Reinforcement Learning: State-of-the-Art*, chap. Reinforcement Learning in Continuous State and Action Spaces, 207–251. Springer. xvii, 48, 50

VAN HASSELT, H. & WIERING, M. (2007). Reinforcement learning in continuous action spaces. In *In Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-2007)*, 272–279. 48

VAN HASSELT, H. & WIERING, M. (2009). Using continuous action spaces to solve discrete problems. In *In Proceedings of the International Joint Conference on Neural Networks (IJCNN-2009)*, 1149–1156. 48

VERBEECK, K. (2004). *Coordinated Exploration in Multi-Agent Reinforcement Learning*. Ph.D. thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, DINF, Computational Modeling Lab. 3, 17

VRABIE, D. & LEWIS, F.L. (2010). Integral reinforcement learning for online computation of feedback nash strategies of nonzero-sum differential games. In *CDC*, 3066–3071. 52

VRABIE, D., PASTRAVANU, O., ABU-KHALAF, M. & LEWIS, F. (2009). Adaptive optimal control for continuous-time linear systems based on policy iteration. *Automatica*, **2**, 477–484. xvii, 33, 48, 50, 51

VRANCX, P. (2010). *Decentralised Reinforcement Learning in Markov Games*. Ph.D. thesis, Vrije Universiteit Brussel. 37

WATKINS, C. & DAYAN, P. (1992). Q-learning. *Machine Learning*. 34

WATKINS, C.J.C.H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College. 34, 41

WHEELER JR., R. & NARENDRA, K. (1986). Decentralized learning in finite markov chains. *IEEE Transactions on Automatic Control*, **31**, 519–526. 37, 75

WILLIAMS, R.J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, 229–256. 47

WITTEN, I.H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control*, **34**, 286–295. 37, 75

XU, X., HU, D. & LU, X. (2007). Kernel-based least-squares policy iteration for reinforcement learning. *IEEE Transactions on Neural Networks*, 973–992. 43

Index

bandit problems, 9–26
- continuous action bandits, 19–25
- discrete action bandits, 10–13

Boltzmann exploration, 35

control, 1
- controller, 2, 27–53
- distributed control, 3, 4

Dynamic programming, 1

ϵ-greedy, 10, 35

function approximators, 3, 40–43, 75–79
- Gaussian kernel based, 43, 111–121
- tile coding, 41, 111–129

games, 14–18
- exploring selfish RL, 17–18, 71–74, 106–109
- Nash equilibrium, 15, 17
- normal form, 14
- Pareto optimality, 15, 18

Markov decision process, 27–32
- Bellman equation, 30, 32, 34
- finite-horizon discounted return, 29
- infinite-horizon discounted return, 28, 31
- Q-function, 29, 31
- V-function, 30, 32

production machines, 4, 81–129
- autonomous tractor, 93–101, 121–129
- linear motor, 91–93
- wire winder, 101–106

Reinforcement learning, 1, 2, 27–53
- learning automata, 2, 4, 12–13, 17–25, 55–129
- policy iteration, 37–38, 46
  - actor critic, 37–38, 47–52
- value iteration, 34–36, 44–46
  - approximate Q-learning, 45–46
  - Q-learning, 34–35
  - SARSA learning, 35–36