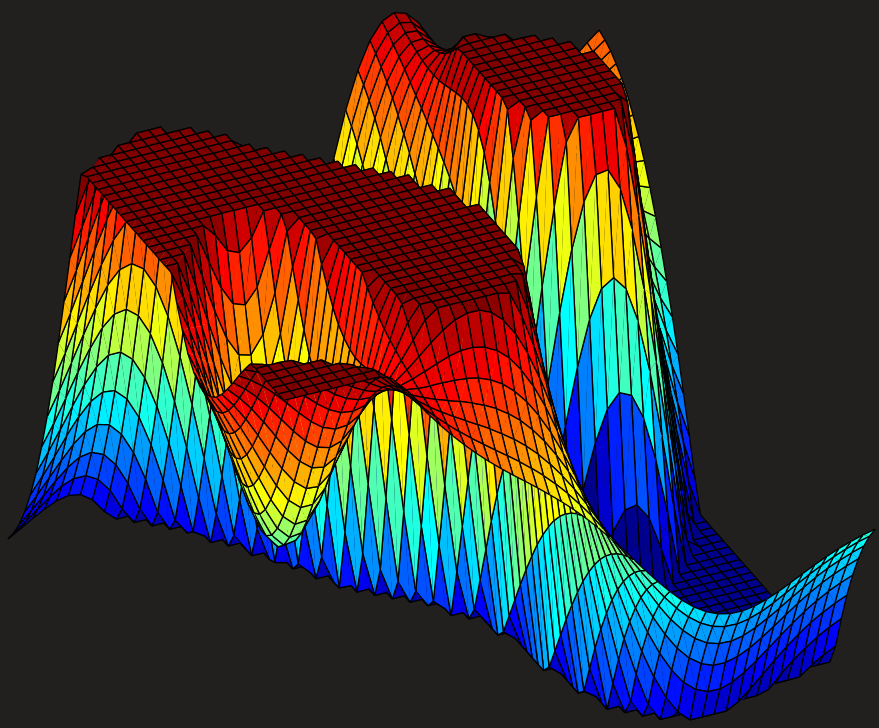
Online Model Learning Algorithms for Actor-Critic Control



Ivo Grondman

Online Model Learning Algorithms for Actor-Critic Control

Ivo Grondman

Cover: Saturated policy for the pendulum swing-up problem as learned by the model learning actor-critic algorithm, approximated using a network of radial basis functions.

Online Model Learning Algorithms for Actor-Critic Control

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op

woensdag 4 maart 2015 om 12:30 uur

door

Ivo GRONDMAN

Master of Science, Imperial College London, Verenigd Koninkrijk,
geboren te Losser.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. R. Babuška

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. R. Babuška	Technische Universiteit Delft, promotor

Onafhankelijke leden:

Prof. dr. ir. B. De Schutter	Technische Universiteit Delft
Prof. dr. ir. P.P. Jonker	Technische Universiteit Delft
Prof. dr. A. Nowé	Vrije Universiteit Brussel
Prof. dr. S. Jagannathan	Missouri University of Science & Technology
Prof. dr. D. Ernst	Université de Liège
Dr. I.L. Buşoniu	Universitatea Tehnicā din Cluj-Napoca

Dr. I.L. Buşoniu (Universitatea Tehnicā din Cluj-Napoca) heeft als begeleider in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

This thesis has been completed in partial fulfilment of the requirements of the Dutch Institute for Systems and Control (DISC) for graduate studies.

Published and distributed by: Ivo Grondman
E-mail: ivo@grondman.net
Web: http://www.grondman.net/

ISBN 978-94-6186-432-1

Printed in the Netherlands

Acknowledgements

During the past years there were quite a few moments where I thought quitting my PhD project was perhaps the best solution to all the problems and stress it was causing. Now that the thesis is finally finished, there are a lot of people I want to thank for their help, support and encouragement, which kept me from actually quitting. With the risk of forgetting someone who I definitely should have mentioned, here goes...

First, I would like to thank my promotor and supervisor, prof. dr. Robert Babuška, for giving me the opportunity to embark on a PhD and for his efforts to keep me going even after leaving the university. Getting a chance to give several lectures on various control systems courses to both BSc and MSc students was also a great experience. Robert, díky za všechno!

Despite the large distance between my workplace and his, my daily supervisor dr. Lucian Buşoniu has been of tremendous help. Whenever I got stuck he was always available for a discussion to get me back on track. His suggestions on and corrections to drafts of papers, which were always in abundance, were also greatly appreciated even though I might not have always shown it while working my way through those stacks of paper covered with red ink.

At the start of 2013, I had a very good time at the Missouri University of Science & Technology in Rolla, Missouri, for which I am grateful to prof. dr. Sarangapani Jagannathan and dr. Hao Xu.

Within the Delft Center for Systems and Control, I thank (former) colleagues Mernout, Edwin, Pieter, Gijs, Jan-Willem, Gabriel, Noortje, Kim, Jacopo, Andrea, Marco, Stefan, Subramanya, Sachin, Ilhan and Jan-Maarten

for their enjoyable company. Jeroen and Melody did a great job during their MSc projects and, although he left DCSC before I arrived, Maarten gave me an excellent starting point for my research.

Outside the academic environment, I want to thank my current colleagues, especially Rachel and Jo, for giving me the final push I needed to finish my PhD.

One of the best ways to relieve stress (and lose weight) during the past years turned out to be running, which I probably never would have discovered without my sisters Evelien and Judith.

A less healthy, but nevertheless very agreeable, way to get my mind off of things was provided in bars and clubs or during weekend outings with Herman, Edwin, Bram, Marinus, Wouter T., Wouter W., Achiel, Max, Bertjan, Joris, Chiel, Jochem and Jeroen.

Finally, I would like to thank my parents for their understanding and support during those many, many years I spent in university.

Ivo Grondman
Den Haag, February 2015

Contents

1 Introduction **1**
- 1.1 Model-Based Control Design . . . 1
- 1.2 Actor-Critic Reinforcement Learning . . . 2
- 1.3 Focus and Contributions . . . 3
  - 1.3.1 Online Model Learning for RL . . . 4
  - 1.3.2 Using Reward Function Knowledge . . . 6
- 1.4 Thesis Outline . . . 6

2 Actor-Critic Reinforcement Learning **9**
- 2.1 Introduction . . . 9
- 2.2 Markov Decision Processes . . . 12
  - 2.2.1 Discounted Reward . . . 13
  - 2.2.2 Average Reward . . . 14
- 2.3 Actor-Critic in the Context of RL . . . 16
  - 2.3.1 Critic-Only Methods . . . 16
  - 2.3.2 Actor-Only Methods and the Policy Gradient . . . 17
  - 2.3.3 Actor-Critic Algorithms . . . 19
  - 2.3.4 Policy Gradient Theorem . . . 23
- 2.4 Standard Gradient Actor-Critic Algorithms . . . 28
  - 2.4.1 Discounted Return Setting . . . 29
  - 2.4.2 Average Reward Setting . . . 32
- 2.5 Natural Gradient Actor-Critic Algorithms . . . 35
  - 2.5.1 Natural Gradient in Optimisation . . . 36
  - 2.5.2 Natural Policy Gradient . . . 40
  - 2.5.3 Natural Actor-Critic Algorithms . . . 42
- 2.6 Applications . . . 46
- 2.7 Discussion . . . 48

3 Efficient Model Learning Actor-Critic Methods **51**
3.1 Introduction and Related Work 52
3.2 Standard Actor-Critic 53
3.3 Model Learning Actor-Critic 53
3.3.1 The Process Model 54
3.3.2 Model-Based Policy Gradient 55
3.4 Reference Model Actor-Critic 57
3.5 Function Approximators 61
3.5.1 Radial Basis Functions 63
3.5.2 Local Linear Regression 64
3.5.3 Tile Coding 71
3.6 Example: Pendulum Swing-Up 72
3.6.1 Standard Actor-Critic 73
3.6.2 Model Learning Actor-Critic 80
3.6.3 Reference Model Actor-Critic 84
3.7 Discussion 88

4 Solutions to Finite Horizon Cost Problems Using Actor-Critic RL **93**
4.1 Introduction 93
4.2 Markov Decision Processes for the Finite Horizon Cost Setting . 95
4.3 Actor-Critic RL for Finite Horizon MDPs 97
4.3.1 Parameterising a Time-Varying Actor and Critic 97
4.3.2 Standard Actor-Critic 99
4.3.3 Model Learning Actor-Critic 100
4.3.4 Reference Model Actor-Critic 102
4.4 Simulation Results 103
4.4.1 Finite Horizon Standard Actor-Critic 105
4.4.2 Finite Horizon Model Learning Actor-Critic 106
4.4.3 Finite Horizon Reference Model Actor-Critic 110
4.5 Discussion 111

5 Simulations with a Two-Link Manipulator **113**
5.1 Simulation Setup 114
5.2 Consequences for Model Learning Methods 114
5.3 Case I: Learn to Inject Proper Damping 115
5.3.1 Standard Actor-Critic 117
5.3.2 Model Learning Actor-Critic 121
5.4 Case II: Learn to Find a Nontrivial Equilibrium 122
5.4.1 Standard Actor-Critic 123

5.4.2 Model Learning Actor-Critic 126
5.5 Discussion 128

6 Learning Rate Free RL Using a Value-Gradient Based Policy 131
6.1 Introduction 131
6.2 SARSA 133
6.3 Value-Gradient Based Policy Algorithm 134
6.3.1 Process Model Parametrisation 135
6.3.2 Critic Parametrisation 136
6.4 Simulation and Experimental Results 137
6.4.1 Underactuated Pendulum Swing-Up 138
6.4.2 Robotic Manipulator 142
6.5 Discussion 147

7 Conclusions and Recommendations 151
7.1 Conclusions 152
7.2 Directions for Future Research 155
7.2.1 Reinforcement Learning 155
7.2.2 Model Learning 156
7.2.3 Function Approximation 156

A Experimental Setups 159
A.1 Inverted Pendulum 159
A.2 Two-Link Manipulator 160

References 163

Glossary 175

Publications by the Author 179

Summary 181

Samenvatting 185

Curriculum Vitae 189

Chapter

1

Introduction

This chapter shortly introduces actor-critic reinforcement learning, which is the main concept on which this thesis is built. Subsequently, a more detailed description of the specific focus and contributions of this thesis is provided, as well as a textual and visual outline of the thesis.

1.1 Model-Based Control Design

The most common approach for finding a controller for a system consists of several steps. First, a model has to be constructed. The model usually consists of a set of equations that can, for example, be derived from first principles. Once the model is available and its parameters have been estimated through system identification, more often than not it will have to be linearised (possibly at more than one operating point) before a control strategy can be applied, as designing control laws straight from a non-linear model remains a tough subject. With the linearised model, one can choose from a number of control methods, e.g. a PID controller, a linear-quadratic (Gaussian) controller (LQR/LQG), an H_∞ controller, etc. (Franklin et al., 2002; Skogestad and Postlethwaite, 2008). All of these steps come with problems of their own. A perfect model of a system may well consist of a large number of equations, which means that one is bound to apply model reduction in order to bring the number of equations down to keep the model manageable at the cost of

some accuracy. Linearisation obviously introduces even more modelling errors around the operating points. Finally, practically all control methods require proper tuning in order to get the controller to satisfy certain constraints. For example, the gains for a PID controller and the weighting matrices/functions for LQR/LQG and H_∞ need to be carefully chosen.

As an alternative approach, there is the possibility of having the system *learn* a controller by itself while it is in operation (online) or offline. The advantages are that it is no longer necessary to construct a complex model for the system and it is possible to learn a non-linear control law. Unfortunately, the problem of proper tuning is still present.

1.2 Actor-Critic Reinforcement Learning

This thesis deals with reinforcement learning controllers (Sutton and Barto, 1998), a subject within the field of artificial intelligence and machine learning. The concept behind reinforcement learning is that a controller (the learning agent) can learn to behave in an optimal way in its environment by receiving rewards or punishments for its behaviour and processing these, quite similar to the way how children or pets learn certain tasks: behaviour that resulted in a punishment will unlikely be repeated, whereas behaviour that got rewarded will, i.e. that behaviour is *reinforced*. In order to achieve this type of learning, the learning agent needs some sort of memory, which stores the relation between behaviour and rewards. In reinforcement learning, this memory is called a *value function*. At every discrete time step, the fully measurable state of the environment is used as input to the *policy*, which governs the behaviour of the learning agent and tells it which action to perform. After executing this action, the environment changes state and a scalar reward is sent to the learning agent to indicate how good or bad the chosen action and the transition of the environment's state was. The learning agent can then process this reward and adjust its value function accordingly, to make a better decision the next time it encounters the same (or a comparable) state.

Reinforcement learning (RL) does not require a model of the system. Instead, the value function and policy only prescribe what action the learning agent should perform when the system is in a certain state. As such, model information is only implicitly stored in the value function. This means RL lessens the burden of having to model a system explicitly before designing a

controller for it. By interacting with the system, RL based controllers do not have to be derived offline and can keep up with small changes to the system. Moreover, optimal nonlinear and stochastic control laws may be learned.

Many RL algorithms embody the value function and the policy into one single function. One specific type of RL algorithms, called actor-critic algorithms (Barto et al., 1983; Konda and Tsitsiklis, 2003; Witten, 1977), split the two entities into two separate functions. This thesis is centred around the actor-critic class of RL algorithms, as these proved useful for control systems which have continuous state and input variables and in real-life applications, such as robotics, this is usually the case. Any RL algorithm used in practice will have to make use of function approximators for both value function and/or policy in order to cover the full continuous range of states and actions. Actor-critic algorithms facilitate the use of continuous state and action spaces in an easy way, as both the actor and the critic are usually parameterised functions and can therefore take a continuous domain as input using only a finite amount of parameters. Moreover, as the policy (the actor) and value function (the critic) are stored separately, generating a control action does not—in contrast to critic-only methods—require an expensive (continuous) optimisation procedure over the value function. Instead, control actions can be calculated directly from the learned policy. A more elaborate, technical description of actor-critic reinforcement learning is given in the next chapter.

1.3 Focus and Contributions

Although RL is in principle meant to be completely model-free, the absence of a model implies that learning will take a considerably long time as a lot of system states will have to be visited repeatedly to gather enough knowledge about the system such that an optimal policy may be found. A main challenge in RL is therefore to use the information gathered during the interaction with the system as efficiently as possible, such that an optimal policy may be reached in a short amount of time. The majority of RL algorithms measure the state, choose an action corresponding to this state, measure the transition to the next state and update a value function (and possible a separate policy). As such, the only source of information used for learning is the transition sample at each time step.

This thesis aims at increasing the learning speed by constructing algorithms

that search for a relation between the collected transition samples and use this relation to predict the system's behaviour from this by interpolation and/or extrapolation. This relation is in fact an approximation of the system's model and as such this particular feature is referred to as "model learning" in this thesis. Furthermore, if (partial) prior knowledge about the system or desired closed-loop behaviour is available, RL algorithms should be able to use this information to their advantage. The final approach to speed up learning addressed in this thesis is to make explicit use of the reward function, instead of only gathering function evaluations of it, that come as part of a transition sample.

1.3.1 Online Model Learning for RL

Two new model learning actor-critic learning algorithms are introduced in this thesis: model learning actor-critic (MLAC) and reference model actor-critic (RMAC). Both have in common that they learn a full-state model of the system to be controlled, which is then used to make one-step predictions about the states a system will end up in if a certain input is applied. The function approximator used for the process model and the reference model in case of RMAC can be pre-trained with prior knowledge about the system, although this is not explored further in this thesis.

Model Learning Actor-Critic

Many standard reinforcement learning algorithms are inefficient in their use of measured data. Once a transition sample—containing the previous state, the action taken, the subsequent state and the instantaneous reward—has been used to update the actor and critic, it is thrown away and never reused in future updates. To overcome this problem, several techniques have been proposed to remember and reuse measured data, such as experience replay (Adam et al., 2011; Lin, 1992; Wawrzyński, 2009) and prioritised sweeping (Moore and Atkeson, 1993). A drawback of these methods is that they require storage of all the samples gathered, making them memory intensive and computationally heavy. Dyna architectures (Sutton, 1990) combine reinforcement learning with the concept of planning, by learning a model of the process or environment online and using this model to generate experiences from which the critic (and thus the policy) can be updated. This results in more frequent updates and hence quicker learning.

In Model Learning Actor-Critic (MLAC), the learned process model is not used to generate experiences. Instead, the process model is used directly in the calculation of the policy gradient, aiming to get faster convergence of learning without increasing the number of updates for the actor and/or critic.

Having a learned process model available simplifies the update of the actor, as it allows to predict what the next state of the system will be, given some applied input. The value function then provides information on the value of that next state. Hence, calculating the optimal input to the system reduces to an optimisation problem, in which the objective function is the value function over the space of possible next states and the decision variable is the applied input or action. It is the gradient of this objective function with respect to the input that will then dictate how the actor should be updated.

Reference Model Actor-Critic

Reference Model Actor-Critic (RMAC) is different from the typical actor-critic methods in the sense that it does not learn an explicit mapping from state to action. Instead of an explicit actor/policy, RMAC learns a reference model that represents a desired behaviour of the system, based on the value function. Similar to MLAC, this algorithm learns a process model to facilitate one-step predictions about the system. The difference with respect to MLAC is that the explicit actor is now replaced by a composition of the learned reference model with the inverse of the learned process model to calculate an action.

Using a reference model provides a means for the storage of demonstration data. Some learning algorithms benefit from having the desired behaviour or task demonstrated to them. This can be done, for example, by a human manually moving a robot arm in such a way that a target task is performed. The demonstrated trajectory is then stored as a sequence of (sampled) states and it is exactly this type of information that can be stored in a reference model.

The applicability of the model learning actor-critic algorithms has been verified with simulation experiments on an inverted pendulum and a two-link manipulator. For the inverted pendulum, experiments have been carried out in both an infinite and finite horizon setting.

1.3.2 Using Reward Function Knowledge

Another way of making learning more efficient, is to make the reward function directly accessible to the learning agent. Classic reinforcement learning theory assumes that the reward function is part of the agent's environment and therefore unknown (Sutton and Barto, 1998). The learning agent only gathers rewards on a per-sample basis. For quite a lot of problems and especially the problems addressed in this thesis, though, the reward function is usually designed by an engineer. Hence, an explicit expression representing the reward function is available and as such can directly be used by the learning agent.

The final algorithm presented in this thesis is based on a Value-Gradient Based Policy (VGBP) by Doya (2000) and makes use of the explicit knowledge of the reward function and also learns a process model online. This enables the algorithm to select control actions by optimising over the right-hand side of the Bellman equation. Simulations and experiments with the underactuated pendulum swing-up task are carried out and additionally, experimental results for the more complex two-link robotic manipulator task are presented.

1.4 Thesis Outline

The remaining chapters of this paper are organised as follows. Chapter 2 provides the necessary background material to understand what reinforcement learning is and, more specifically, how actor-critic reinforcement learning algorithms work. It also discusses the difference between a regular/vanilla gradient and the natural gradient and ends with a short survey of existing actor-critic methods and their applications. Chapter 3 introduces the intuition behind and the implementation of the model learning algorithms MLAC and RMAC, together with a standard actor-critic (SAC) algorithm that is used as a benchmark in the simulation experiments. Furthermore, commonly used function approximators are explained and the chapter ends with a set of simulation results to demonstrate the effectiveness of the model learning algorithms. Chapter 4 extends the model learning algorithms, such that they may be used in a finite horizon cost setting and also evaluates them with a set of simulations. Without introducing any new theory, Chapter 5 evaluates the model learning algorithms in a tougher multi-input, multi-output setting. A novel algorithm based on using explicit knowledge of the reward function, in addition to learning a model, is introduced and evaluated in Chapter 6. Except

for Chapter 5, all of these chapters are based on published journal papers and/or conference papers, listed on page 179 of this thesis[1]. Chapter 7 is the final chapter of this thesis and summarises the conclusions drawn throughout the thesis and offers recommendations for future research. There is one appendix to this thesis, Appendix A, which describes the setups used for the simulation experiments in the thesis.

Figure 1.1 shows a graphical roadmap of the thesis, indicating with arrows the orders in which the separate chapters may be read.

[1] The converse is not true: some papers listed there are not embodied in this thesis.

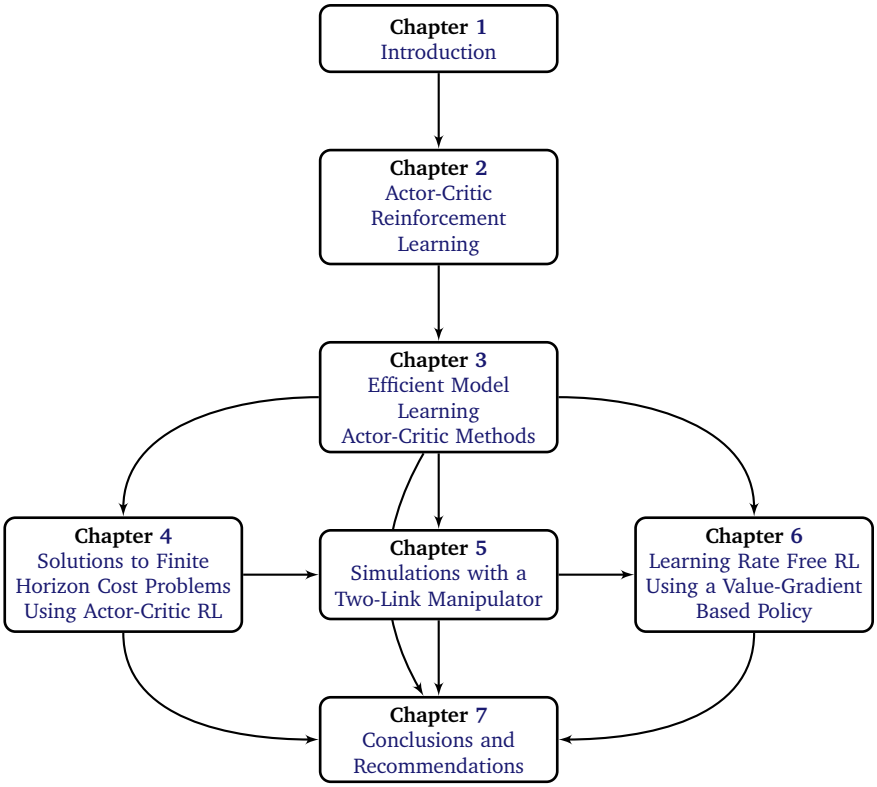


Figure 1.1 Roadmap of this thesis. Arrows indicate possible orders in which the separate chapters may be read.

Chapter

2

Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients

Policy gradient based actor-critic algorithms are amongst the most popular algorithms in the reinforcement learning framework. Their advantage of being able to search for optimal policies using low-variance gradient estimates has made them useful in several real-life applications, such as robotics, power control and finance. Since actor-critic algorithms are a central topic in this thesis, a thorough background on this type of algorithms should be given. This chapter therefore describes the state of the art of actor-critic algorithms, with a focus on methods that can work in an online setting and use function approximation in order to deal with continuous state and action spaces. After a discussion on the concepts of reinforcement learning and the origins of actor-critic algorithms, this chapter describes the workings of the natural gradient, which has made its way into many actor-critic algorithms in the past few years. A review of several standard and natural actor-critic algorithms follows and the chapter concludes with an overview of application areas and a discussion on open issues.

2.1 Introduction

Reinforcement learning is a framework inspired by animal learning in which an agent (or controller) optimises its behaviour by interacting with its environment in a trial-and-error fashion (Lewis and Vrabie, 2009). After taking an

action in some state, the agent receives a scalar reward from the environment, which gives the agent an indication of the quality of that action. The function that indicates the action to take in a certain state is called the *policy*. The main goal of the agent is to find a policy that maximises the total accumulated reward, also called the *return*. By following a given policy and processing the rewards, the agent can build estimates of the return. The function representing this estimated return is known as the *value function*. This value function allows the agent to make indirect use of past experiences to decide on future actions to take in or around a certain state.

Over the course of time, several types of RL algorithms have been introduced and they can be divided into into three groups (Konda and Tsitsiklis, 2003): actor-only, critic-only and actor-critic methods, where the words actor and critic are synonyms for the policy and value function, respectively. Actor-only methods typically work with a parameterised family of policies over which optimisation procedures can be used directly. The benefit of a parameterised policy is that a spectrum of continuous actions can be generated, but the optimisation methods used (typically called policy gradient methods) suffer from high variance in the estimates of the gradient, leading to slow learning (Baxter and Bartlett, 2001; Berenji and Vengerov, 2003; Boyan, 2002; Konda and Tsitsiklis, 2003; Richter et al., 2007).

Critic-only methods that use temporal difference learning have a lower variance in the estimates of expected returns (Berenji and Vengerov, 2003; Boyan, 2002; Sutton, 1988). A straightforward way of deriving a policy in critic-only methods is by selecting *greedy actions* (Sutton and Barto, 1998): actions for which the value function indicates that the expected return is the highest. However, to do this, one needs to resort to an optimisation procedure in every state encountered to find the action leading to an optimal value. This can be computationally intensive, especially if the action space is continuous. Therefore, critic-only methods usually discretise the continuous action space, after which the optimisation over the action space becomes a matter of enumeration. Obviously, this approach undermines the ability of using continuous actions and thus of finding the true optimal policy.

Actor-critic methods combine the advantages of actor-only and critic-only methods. While the parameterised actor brings the advantage of computing continuous actions without the need for optimisation procedures on a value function, the critic's merit is that it supplies the actor with low-variance knowledge of the performance. More specifically, the critic's estimate of the

expected return allows for the actor to update with gradients that have lower variance, speeding up the learning process. The lower variance is traded for a larger bias at the start of learning when the critic's estimates are far from accurate (Berenji and Vengerov, 2003). Actor-critic methods usually have good convergence properties, in contrast to critic-only methods (Konda and Tsitsiklis, 2003).

These nice properties of actor-critic methods have made them a preferred reinforcement learning algorithm, also in real-life application domains. General surveys on reinforcement learning already exist (Gosavi, 2009; Kaelbling et al., 1996; Szepesvári, 2010), but despite the growing popularity and recent developments in the field of actor-critic algorithms, no survey is specifically dedicated to them. The goal of this chapter is to give an overview of the work on (online) actor-critic algorithms, giving technical details of some representative algorithms, and also to provide references to a number of application papers. This provides a background for the remainder of the thesis. Additionally, the algorithms are presented in one unified notation, which allows for a better technical comparison of the variants and implementations. These templates will be used throughout the thesis. Because the discrete-time variant has been developed to a reasonable level of maturity, this thesis solely discusses algorithms in the discrete-time setting. Continuous-time variants of actor-critic algorithms (Hanselmann et al., 2007; Vamvoudakis and Lewis, 2010) and multi-agent actor-critic schemes (Li et al., 2008; Pennesi and Paschalidis, 2010) are not considered here.

The focus is put on actor-critic algorithms based on policy gradients, which constitute the largest part of actor-critic algorithms. A distinction is made between algorithms that use a *standard* (sometimes also called vanilla) gradient and the *natural* gradient that became more popular in the course of the last decade. The remaining part of actor-critic algorithms consists mainly of algorithms that choose to update their policy by moving it towards the greedy policy underlying an approximate state-action value function (Szepesvári, 2010). In this thesis, these algorithms are regarded as critic-only algorithms as the policy is implemented implicitly by the critic. Algorithms are only categorised as actor-critic here if they implement two separately parameterised representations for the actor and the critic. Furthermore, all algorithms make use of function approximation, which in real-life applications such as robotics is necessary in order to deal with continuous state and action spaces.

The remainder of this chapter is organised as follows. Section 2.2 intro-

duces the basic concepts of a Markov decision process, which is the cornerstone of reinforcement learning. Section 2.3 describes critic-only, actor-only and actor-critic RL algorithms and the important policy gradient theorem, after which Section 2.4 surveys actor-critic algorithms that use a standard gradient. Section 2.5 describes the natural gradient and its application to actor-critic methods, and also surveys several natural actor-critic algorithms. Section 2.6 briefly reviews the application areas of these methods. A concluding discussion is provided in Section 2.7.

2.2 Markov Decision Processes

This section introduces the concepts of discrete-time reinforcement learning, as laid out by Sutton and Barto (1998), but extended to the use of continuous state and action spaces and also assuming a stochastic setting, as covered more extensively by Peters and Schaal (2008a) and Buşoniu et al. (2010).

A reinforcement learning algorithm can be used to solve problems modelled as Markov decision processes (MDPs). An MDP is a tuple $\langle X, U, f, \rho \rangle$, where X denotes the state space, U the action space, $f : X \times U \times X \to [0, \infty)$ the state transition probability density function and $\rho : X \times U \times X \to \mathbb{R}$ the reward function. In this thesis, only stationary MDPs are considered, which means that the elements of the tuple $\langle X, U, f, \rho \rangle$ do not change over time.

The stochastic process to be controlled is described by the state transition probability density function f. It is important to note that since the state space is continuous, it is only possible to define a probability of reaching a certain state *region*, since the probability of reaching a particular state is zero. The probability of reaching a state x_{k+1} in the region $X_{k+1} \subseteq X$ from state x_k after applying action u_k is

$$\mathrm{P}(x_{k+1} \in X_{k+1} | x_k, u_k) = \int_{X_{k+1}} f(x_k, u_k, x') \mathrm{d}x'.$$

After each transition to a state x_{k+1}, the controller receives an immediate reward

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

which depends on the previous state, the current state and the action taken. The reward function ρ is assumed to be bounded. The action u_k taken in a state x_k is drawn from a stochastic policy $\pi : X \times U \to [0, \infty)$.

The goal of the reinforcement learning agent is to find the policy π which maximises the expected value of a certain function g of the immediate rewards received while following the policy π. This expected value is the cost-to-go function

$$J(\pi) = \mathrm{E}\left\{g(r_1, r_2, \ldots)|\pi\right\}.$$

In most cases[1], the function g is either the discounted sum of rewards or the average reward received, as explained next.

2.2.1 Discounted Reward

In the discounted reward setting (Bertsekas, 2007), the cost function J is equal to the expected value of the discounted sum of rewards when starting from an initial state $x_0 \in X$ drawn from an initial state distribution $x_0 \sim d_0(\cdot)$, also called the discounted return

$$\begin{aligned} J(\pi) &= \mathrm{E}\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| d_0, \pi\right\} \\ &= \int_X d_\gamma^\pi(x) \int_U \pi(x,u) \int_X f(x,u,x') \rho(x,u,x') \mathrm{d}x' \mathrm{d}u \mathrm{d}x, \end{aligned} \tag{2.1}$$

where $d_\gamma^\pi(x) = \sum_{k=0}^{\infty} \gamma^k p(x_k = x|d_0, \pi)$ is the discounted state distribution under the policy π (Peters and Schaal, 2008a; Sutton et al., 2000) and $\gamma \in [0,1)$ denotes the reward discount factor. Note that $p(x_k = x)$ is a probability density function here.

During learning, the agent will have to estimate the cost-to-go function J for a given policy π. This procedure is called *policy evaluation*. The resulting estimate of J is called the *value function* and two definitions exist for it. The state value function

$$V^\pi(x) = \mathrm{E}\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| x_0 = x, \pi\right\} \tag{2.2}$$

[1] Other cost functionals do exist and can be used for actor-critic algorithms, such as the risk-sensitive cost (Borkar, 2001).

only depends on the state x and assumes that the policy π is followed starting from this state. The state-action value function

$$Q^{\pi}(x,u) = \mathrm{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| x_0 = x, u_0 = u, \pi \right\} \tag{2.3}$$

also depends on the state x, but makes the action u chosen in this state a free variable instead of having it generated by the policy π. Once the first transition onto a next state has been made, π governs the rest of the action selection. The relationship between these two definitions for the value function is given by

$$V^{\pi}(x) = \mathrm{E}\left\{ Q^{\pi}(x,u) | u \sim \pi(x,\cdot) \right\}.$$

With some manipulation, Equations (2.2) and (2.3) can be put into a recursive form (Bertsekas, 2007). For the state value function this is

$$V^{\pi}(x) = \mathrm{E}\left\{ \rho(x,u,x') + \gamma V^{\pi}(x') \right\}, \tag{2.4}$$

with u drawn from the probability distribution function $\pi(x,\cdot)$ and x' drawn from $f(x,u,\cdot)$. For the state-action value function the recursive form is

$$Q^{\pi}(x,u) = \mathrm{E}\left\{ \rho(x,u,x') + \gamma Q^{\pi}(x',u') \right\}, \tag{2.5}$$

with x' drawn from the probability distribution function $f(x,u,\cdot)$ and u' drawn from the distribution $\pi(x',\cdot)$. These recursive relationships are called Bellman equations (Sutton and Barto, 1998).

Optimality for both the state value function V^{π} and the state-action value function Q^{π} is governed by the Bellman *optimality* equation. Denoting the optimal state value function with $V^*(x)$ and the optimal state-action value with $Q^*(x,u)$, the corresponding Bellman optimality equations for the discounted reward setting are

$$V^*(x) = \max_u \mathrm{E}\left\{ \rho(x,u,x') + \gamma V^*(x') \right\} \tag{2.6a}$$

$$Q^*(x,u) = \mathrm{E}\left\{ \rho(x,u,x') + \gamma \max_{u'} Q^*(x',u') \right\}. \tag{2.6b}$$

2.2.2 Average Reward

As an alternative to the discounted reward setting, there is also the approach of using the *average* return (Bertsekas, 2007). In this setting a starting state

x_0 does not need to be chosen, under the assumption that the process is ergodic (Sutton and Barto, 1998) and thus that J does not depend on the starting state. Instead, the value functions for a policy π are defined relative to the average expected reward per time step under the policy, turning the cost-to-go function into

$$J(\pi) = \lim_{n\to\infty} \frac{1}{n} \mathrm{E}\left\{ \sum_{k=0}^{n-1} r_{k+1} \middle| \pi \right\} = \int_X d^\pi(x) \int_U \pi(x,u) \int_X f(x,u,x')\rho(x,u,x')\mathrm{d}x'\mathrm{d}u\mathrm{d}x. \quad (2.7)$$

Equation (2.7) is very similar to Equation (2.1), except that the definition for the state distribution changed to $d^\pi(x) = \lim_{k\to\infty} p(x_k = x \,|\, \pi)$. For a given policy π, the state value function $V^\pi(x)$ and state-action value function $Q^\pi(x,u)$ are then defined as

$$V^\pi(x) = \mathrm{E}\left\{ \sum_{k=0}^{\infty} \big(r_{k+1} - J(\pi)\big) \middle| x_0 = x, \pi \right\}$$
$$Q^\pi(x,u) = \mathrm{E}\left\{ \sum_{k=0}^{\infty} \big(r_{k+1} - J(\pi)\big) \middle| x_0 = x, u_0 = u, \pi \right\}.$$

The Bellman equations for the average reward—in this case also called the Poisson equations (Bhatnagar et al., 2009)—are

$$V^\pi(x) + J(\pi) = \mathrm{E}\left\{\rho(x,u,x') + V^\pi(x')\right\}, \quad (2.8)$$

with u and x' drawn from the appropriate distributions as before and

$$Q^\pi(x,u) + J(\pi) = \mathrm{E}\left\{\rho(x,u,x') + Q^\pi(x',u')\right\}, \quad (2.9)$$

again with x' and u' drawn from the appropriate distributions. Note that Equations (2.8) and (2.9) both require the value $J(\pi)$, which is unknown and hence needs to be estimated in some way. This will be addressed in a later section. Bellman optimality equations, describing an optimum for the average reward case, are

$$V^*(x) + J^* = \max_u \mathrm{E}\left\{\rho(x,u,x') + V^*(x')\right\} \quad (2.10a)$$
$$Q^*(x,u) + J^* = \mathrm{E}\left\{\rho(x,u,x') + \max_{u'} Q^*(x',u')\right\}, \quad (2.10b)$$

where J^* is the optimal average reward as defined by (2.7) when an optimal policy π^* is used.

2.3 Actor-Critic in the Context of RL

As discussed in the introduction, the vast majority of reinforcement learning methods can be divided into three groups (Konda and Tsitsiklis, 2003): critic-only, actor-only and actor-critic methods. This section will give an explanation on all three groups, starting with critic-only methods. Section 2.3.2 introduces the concept of a policy gradient, which provides the basis for actor-critic algorithms. The final subsection explains the policy gradient theorem, an important result that is now widely used in many implementations of actor-critic algorithms.

In real-life applications, such as robotics, processes usually have continuous or very large discrete state and action spaces, making it impossible to store exact value functions or policies for each separate state or state-action pair. Any RL algorithm used in practice will have to make use of function approximators for the value function and/or the policy in order to cover the full range of states and actions. Therefore, this section assumes the use of such function approximators.

2.3.1 Critic-Only Methods

Critic-only methods, such as Q-learning (Bradtke et al., 1994; Watkins and Dayan, 1992; Watkins, 1989) and SARSA (Rummery and Niranjan, 1994), use a state-action value function and no explicit function for the policy. For continuous state and action spaces, this will be an approximate state-action value function. These methods learn the optimal value function by finding an approximate solution to the Bellman equation (2.6b) or (2.10b) online. A deterministic policy, denoted by $\pi : X \to U$ is calculated by using an optimisation procedure over the value function

$$\pi(x) = \arg\max_u Q(x, u). \tag{2.11}$$

There is no reliable guarantee on the near-optimality of the resulting policy for just any approximated value function when learning in an online setting. For example, Q-learning and SARSA with specific function approximators have

been shown not to converge even for simple MDPs (Baird, 1995; Gordon, 1995; Tsitsiklis and Van Roy, 1996). However, the counterexamples used to show divergence were further analysed in Tsitsiklis and Van Roy (1997) (with an extension to the stochastic setting in Melo et al. (2008)) and it was shown that convergence can be assured for linear-in-parameters function approximators if trajectories are sampled according to their on-policy distribution. Tsitsiklis and Van Roy (1997) also provide a bound on the approximation error between the true value function and the approximation learned by online temporal difference learning. An analysis of more approximate policy evaluation methods is provided by Schoknecht (2003), mentioning conditions for convergence and bounds on the approximation error for each method. Nevertheless, for most choices of basis functions an approximated value function learned by temporal difference learning will be biased. This is reflected by the state-of-the-art bounds on the least-squares temporal difference (LSTD) solution quality (Lazaric et al., 2010), which always include a term depending on the distance between the true value function and its projection on the approximation space. For a particularly bad choice of basis functions, this bias can grow very large.

The problem of off-policy methods not converging when function approximation is used has later been addressed by Sutton et al. (2009) in their gradient temporal difference (GTD) and linear TD with gradient correction (TDC) algorithms and by Maei et al. (2010) in their Greedy-GQ algorithm.

2.3.2 Actor-Only Methods and the Policy Gradient

Policy gradient methods, such as the stochastic real-valued (SRV) algorithm (Gullapalli, 1990) and REINFORCE (Williams, 1992) algorithms, are principally actor-only and do not use any form of a stored value function. Instead, the majority of actor-only algorithms work with a parameterised family of policies and optimise the cost defined by (2.1) or (2.7) directly over the parameter space of the policy. Although not explicitly considered in this chapter, work on non-parametric policy gradients does exist, see e.g. the work by Bagnell and Schneider (2003b); Kersting and Driessens (2008). A major advantage of actor-only methods over critic-only methods is that they allow the policy to generate actions in the complete continuous action space.

A policy gradient method is generally obtained by parameterising the policy π by the parameter vector $\vartheta \in \mathbb{R}^p$, with p the number of features used in

the approximator. Considering that both (2.1) and (2.7) are functions of the parameterised policy π_ϑ, they are in fact functions of ϑ. Assuming that the cost-to-go J is differentiable with respect to the policy and the parameterisation of the policy is differentiable with respect to ϑ, the gradient of the cost function with respect to ϑ is described by

$$\nabla_\vartheta J = \frac{\partial J}{\partial \pi_\vartheta} \frac{\partial \pi_\vartheta}{\partial \vartheta}. \tag{2.12}$$

Then, by using standard optimisation techniques, a locally optimal solution of the cost J can be found. The gradient $\nabla_\vartheta J$ is estimated per time step and the parameters are then updated in the direction of this gradient. For example, a simple gradient ascent method would yield the policy gradient update equation

$$\vartheta_{k+1} = \vartheta_k + \alpha_{a,k} \nabla_\vartheta J_k, \tag{2.13}$$

where $\alpha_{a,k} > 0$ is a small enough learning rate for the actor, by which it is obtained that[2] $J(\vartheta_{k+1}) \geq J(\vartheta_k)$.

Several methods exist to estimate the gradient, e.g. by using infinitesimal perturbation analysis (IPA) or likelihood-ratio methods (Aleksandrov et al., 1968; Glynn, 1987). Baxter and Bartlett (2001); Peters and Schaal (2008b) provide a broader discussion on these methods. Approaches to model-based gradient methods are given by Dyer and McReynolds (1970); Hasdorff (1976); Jacobson and Mayne (1970) and in the more recent work of Deisenroth and Rasmussen (2011).

The main advantage of actor-only methods is their convergence property, which is naturally inherited from gradient descent methods. Convergence is obtained if the estimated gradients are unbiased and the learning rates $\alpha_{a,k}$ satisfy (Peters and Schaal, 2008b; Sutton and Barto, 1998)

$$\sum_{k=0}^{\infty} \alpha_{a,k} = \infty \qquad\qquad \sum_{k=0}^{\infty} \alpha_{a,k}^2 < \infty.$$

A drawback of the actor-only approach is that the estimated gradient may have a large variance (Riedmiller et al., 2007; Sutton et al., 2000). Also, every gradient is calculated without using any knowledge of past estimates (Konda and Tsitsiklis, 2003; Peters et al., 2010).

[2] One could also define the cost J such that it should be minimised. In that case, the plus sign in Equation (2.13) is replaced with a minus sign, resulting in $J(\vartheta_{k+1}) \leq J(\vartheta_k)$.

2.3.3 Actor-Critic Algorithms

Actor-critic methods (Barto et al., 1983; Konda and Tsitsiklis, 2003; Witten, 1977) aim to combine the advantages of actor-only and critic-only methods. Like actor-only methods, actor-critic methods are capable of producing continuous actions, while the large variance in the policy gradients of actor-only methods is countered by adding a critic. The role of the critic is to evaluate the current policy prescribed by the actor. In principle, this evaluation can be done by any policy evaluation method commonly used, such as TD(λ) (Bertsekas, 2007; Sutton, 1988), LSTD (Bertsekas, 2007; Boyan, 2002; Bradtke and Barto, 1996) or residual gradients (Baird, 1995). The critic approximates and updates the value function using samples. The value function is then used to update the actor's policy parameters in the direction of performance improvement. These methods usually preserve the desirable convergence properties of policy gradient methods, in contrast to critic-only methods. In actor-critic methods, the policy is not directly inferred from the value function by using (2.11). Instead, the policy is updated in the policy gradient direction using only a small step size α_a, meaning that a change in the value function will only result in a small change in the policy, leading to less or no oscillatory behaviour in the policy as described by Baird and Moore (1999).

Figure 2.1 shows the schematic structure of an actor-critic algorithm. The learning agent has been split into two separate entities: the actor (policy) and the critic (value function). The actor is only responsible for generating a control input u, given the current state x. The critic is responsible for processing the rewards it receives, i.e. evaluating the quality of the current policy by adapting the value function estimate. After a number of policy evaluation steps by the critic, the actor is updated by using information from the critic.

A unified notation for the actor-critic algorithms described in this thesis allows for an easier comparison between them. Also, most algorithms can be fitted to a general *template* of standard update rules. Therefore, two actor-critic algorithm templates are introduced: one for the discounted reward setting and one for the average reward setting. Once these templates are established, specific actor-critic algorithms can be discussed by only looking at how they fit into the general template or in what way they differ from it.

For both reward settings, the value function is parameterised by the parameter vector $\theta \in \mathbb{R}^q$, with q the number of features used in the approximator.

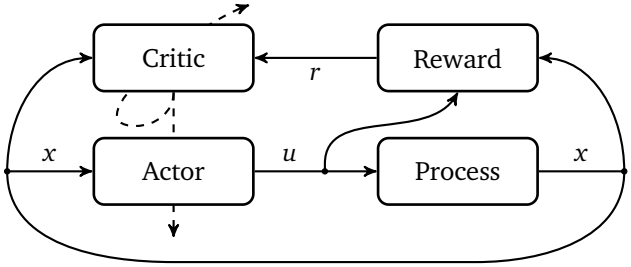


Figure 2.1 Schematic overview of an actor-critic algorithm. The dashed line indicates that the critic is responsible for updating the actor and itself.

This will be denoted with $V_\theta(x)$ or $Q_\theta(x,u)$. If the parameterisation is linear, the features (basis functions) will be denoted with ϕ, i.e.

$$V_\theta(x) = \theta^\top \phi(x) \qquad\qquad Q_\theta(x,u) = \theta^\top \phi(x,u). \tag{2.14}$$

The stochastic policy π is parameterised by $\vartheta \in \mathbb{R}^p$ and will be denoted with $\pi_\vartheta(x,u)$. If the policy is denoted with $\pi_\vartheta(x)$, it is deterministic and no longer represents a probability density function, but the direct mapping from states to actions $u = \pi_\vartheta(x)$.

The goal in actor-critic algorithms—or any other RL algorithm for that matter—is to find the best policy possible, given some stationary MDP. A prerequisite for this is that the critic is able to accurately evaluate a given policy. In other words, the goal of the critic is to find an approximate solution to the Bellman equation for that policy. The difference between the right-hand and left-hand side of the Bellman equation, whether it is the one for the discounted reward setting (2.4) or the average reward setting (2.8), is called the temporal difference (TD) error and is used to update the critic. Using the function approximation for the critic and a transition sample $(x_k, u_k, r_{k+1}, x_{k+1})$, the TD error is estimated as

$$\delta_k = r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k). \tag{2.15}$$

Perhaps the most standard way of updating the critic, is to exploit this TD error for use in a gradient descent update (Sutton and Barto, 1998)

$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k \nabla_\theta V_{\theta_k}(x_k), \tag{2.16}$$

where $\alpha_{c,k} > 0$ is the learning rate of the critic. For the linearly parameterised function approximator (2.14), this reduces to

$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k\phi(x_k). \tag{2.17}$$

This temporal difference method is also known as TD(0) learning, as no eligibility traces are used. The extension to the use of eligibility traces, resulting in TD(λ) methods, is straightforward and is explained next.

Using (2.16) to update the critic results in a one-step backup, whereas the reward received is often the result of a series of steps. Eligibility traces offer a better way of assigning credit to states or state-action pairs visited several steps earlier. The eligibility trace vector for all q features at time instant k is denoted with $z_k \in \mathbb{R}^q$ and its update equation is (Konda and Tsitsiklis, 2003; Sutton and Barto, 1998)

$$z_k = \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k).$$

It decays with time by a factor $\lambda\gamma$, with $\lambda \in [0, 1)$ the trace decay rate and also takes into account the discount factor γ of the return. This makes the recently used features more eligible for receiving credit. The use of eligibility traces speeds up the learning considerably. Using the eligibility trace vector z_k, the update (2.16) of the critic becomes

$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k z_k. \tag{2.18}$$

With the use of eligibility traces, the actor-critic template for the discounted return setting becomes

Actor-Critic Template 2.1 (Discounted Return)**.**

$$\delta_k = r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k) \tag{2.19a}$$
$$z_k = \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k) \tag{2.19b}$$
$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k z_k \tag{2.19c}$$
$$\vartheta_{k+1} = \vartheta_k + \alpha_{a,k}\nabla_\vartheta J_k. \tag{2.19d}$$

Although not commonly seen, eligibility traces may be introduced for the actor as well (Barto et al., 1983). As with actor-only methods (see Section 2.3.2), several ways exist to estimate $\nabla_\vartheta J_k$.

For the average reward case, the critic can be updated using the average-cost TD method (Tsitsiklis and Van Roy, 1999). Then, the Bellman equation (2.8) is considered, turning the TD error into

$$\delta_k = r_{k+1} - \hat{J}_k + V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k),$$

with $\hat{J}_k$ an estimate of the average cost at time k. Obviously, this requires an update equation for the estimate $\hat{J}$ as well, which usually is (Konda and Tsitsiklis, 2003)

$$\hat{J}_k = \hat{J}_{k-1} + \alpha_{J,k}(r_{k+1} - \hat{J}_{k-1}),$$

where $\alpha_{J,k} \in (0,1]$ is another learning rate. The critic still updates with Equation (2.18). The update of the eligibility trace also needs to be adjusted, as the discount rate γ is no longer present. The template for actor-critic algorithms in the average return setting then is

Actor-Critic Template 2.2 (Average Return)**.**

$$\hat{J}_k = \hat{J}_{k-1} + \alpha_{J,k}(r_{k+1} - \hat{J}_{k-1}) \tag{2.20a}$$
$$\delta_k = r_{k+1} - \hat{J}_k + V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k) \tag{2.20b}$$
$$z_k = \lambda z_{k-1} + \nabla_\theta V_{\theta_k}(x_k) \tag{2.20c}$$
$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k z_k \tag{2.20d}$$
$$\vartheta_{k+1} = \vartheta_k + \alpha_{a,k}\nabla_\vartheta J_k. \tag{2.20e}$$

For the actor-critic algorithm to converge, it is necessary that the critic's estimate is at least asymptotically accurate. This is the case if the step sizes $\alpha_{a,k}$ and $\alpha_{c,k}$ are deterministic, non-increasing and satisfy (Konda and Tsitsiklis,

2003)

$$\sum_k \alpha_{a,k} = \infty \qquad \sum_k \alpha_{c,k} = \infty \tag{2.21}$$

$$\sum_k \alpha_{a,k}^2 < \infty \qquad \sum_k \alpha_{c,k}^2 < \infty \qquad \sum_k \left(\frac{\alpha_{a,k}}{\alpha_{c,k}}\right)^d < \infty \tag{2.22}$$

for some $d \geq 0$. The learning rate $\alpha_{J,k}$ is usually set equal to $\alpha_{c,k}$. Note that such assumptions on learning rates are typical for all RL algorithms. They ensure that learning will slow down, but never stops and also that the update of the actor operates on a slower time-scale than the critic, to allow the critic enough time to, at least partially, evaluate the current policy.

Although TD(λ) learning is used quite commonly, other ways of determining the critic parameter θ do exist and some are even known to be superior in terms of convergence rate in both discounted and average reward settings (Paschalidis et al., 2009), such as least-squares temporal difference learning (LSTD) (Boyan, 2002; Bradtke and Barto, 1996). LSTD uses samples collected along a trajectory generated by a policy π to set up a system of temporal difference equations derived from or similar to (2.19a) or (2.20b). As LSTD requires an approximation of the value function which is linear in its parameters, i.e. $V_\theta(x) = \theta^\top \phi(x)$, this system is linear and can easily be solved for θ by a least-squares method. Regardless of how the critic approximates the value function, the actor update is always centred around Equation (2.13), using some way to estimate $\nabla_\vartheta J_k$.

For actor-critic algorithms, the question arises how the critic influences the gradient update of the actor. This is explained in the next subsection about the policy gradient theorem.

2.3.4 Policy Gradient Theorem

Many actor-critic algorithms now rely on the policy gradient theorem, a result obtained simultaneously by Konda and Tsitsiklis (2003) and Sutton et al. (2000), proving that an unbiased estimate of the gradient (2.12) can be obtained from experience using an approximate value function satisfying certain properties. The basic idea, given by Konda and Tsitsiklis (2003), is that since the number of parameters that the actor has to update is relatively small compared to the (usually infinite) number of states, it is not useful to

have the critic attempting to compute the exact value function, which is also a high-dimensional object. Instead, it should compute a projection of the value function onto a low-dimensional subspace spanned by a set of basis functions, which are completely determined by the parameterisation of the actor.

In the case of an approximated stochastic policy, but exact state-action value function Q^π, the policy gradient theorem is as follows.

Theorem 2.1 (Policy Gradient). *For any MDP, in either the average reward or discounted reward setting, the policy gradient is given by*

$$\nabla_\vartheta J = \int_X d^\pi(x) \int_U \nabla_\vartheta \pi(x,u) Q^\pi(x,u) du dx,$$

with $d^\pi(x)$ defined for the appropriate reward setting.

Proof: See Sutton et al. (2000)

This clearly shows the relationship between the policy gradient $\nabla_\vartheta J$ and the critic function $Q^\pi(x,u)$ and ties together the update equations of the actor and critic in the Templates 2.1 and 2.2.

For most applications, the state-action space is continuous and thus infinite, which means that it is necessary to approximate the state(-action) value function. The result in Konda and Tsitsiklis (2003); Sutton et al. (2000) shows that $Q^\pi(x,u)$ can be approximated with[3] $h_w : X \times U \to \mathbb{R}$, parameterised by w, without affecting the unbiasedness of the policy gradient estimate.

In order to find the closest approximation of Q^π by h_w, one can try to find the w that minimises the quadratic error

$$\mathcal{E}_w^\pi(x,u) = \frac{1}{2}\left[Q^\pi(x,u) - h_w(x,u)\right]^2.$$

The gradient of this quadratic error with respect to w is

$$\nabla_w \mathcal{E}_w^\pi(x,u) = \left[Q^\pi(x,u) - h_w(x,u)\right] \nabla_w h_w(x,u) \tag{2.23}$$

[3]This approximation of $Q^\pi(x,u)$ is not denoted with an accented Q as it is not actually the value function Q that it is approximating, as shown later in this section.

and this can be used in a gradient descent algorithm to find the optimal w. If the estimator of $Q^\pi(x,u)$ is unbiased, the expected value of Equation (2.23) is zero for the optimal w, i.e.

$$\int_X d^\pi(x) \int_U \pi(x,u) \nabla_w \mathcal{E}_w^\pi(x,u) \mathrm{d}u\mathrm{d}x = 0. \tag{2.24}$$

The policy gradient theorem with function approximation is based on the equality in (2.24).

Theorem 2.2 (Policy Gradient with Function Approximation)**.** *If h_w satisfies Equation (2.24) and*

$$\nabla_w h_w(x,u) = \nabla_\vartheta \ln \pi_\vartheta(x,u), \tag{2.25}$$

where $\pi_\vartheta(x,u)$ denotes the stochastic policy, parameterised by ϑ, then

$$\nabla_\vartheta J = \int_X d^\pi(x) \int_U \nabla_\vartheta \pi(x,u) h_w(x,u) du dx. \tag{2.26}$$

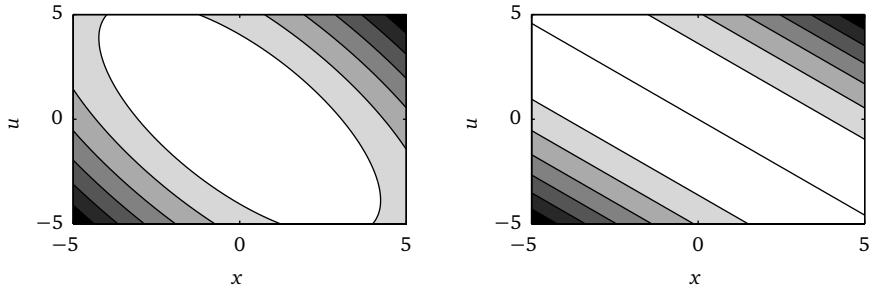
Proof: See Sutton et al. (2000).

An extra assumption in Konda and Tsitsiklis (2003) is that in (2.25), h actually needs to be an approximator that is linear with respect to some parameter w and features ψ, i.e. $h_w = w^\top \psi(x,u)$, transforming condition (2.25) into

$$\psi(x,u) = \nabla_\vartheta \ln \pi_\vartheta(x,u). \tag{2.27}$$

Features ψ that satisfy Equation (2.27) are also known as *compatible* features (Kakade, 2001; Konda and Tsitsiklis, 2003; Sutton et al., 2000). In the remainder of the thesis, these will always be denoted by ψ and their corresponding parameters with w.

A technicality, discussed in detail by Peters and Schaal (2008a) and Sutton et al. (2000), is that using the compatible function approximation $h_w =$



(a) Value function $Q^*(x,u)$ (b) Advantage function $A^*(x,u)$

Figure 2.2 The optimal value and advantage function for an example MDP (Peters and Schaal, 2008a). The system is $x_{k+1} = x_k + u_k$, using the optimal policy $\pi^*(x) = -Kx$ with K the state feedback solution based on the reward function $r_k = -x_k^2 - 0.1u_k^2$. The advantage function nicely shows the zero contour line of the optimal action $u = -Kx$.

$w^\top \nabla_\vartheta \ln \pi_\vartheta(x,u)$ gives

$$\int_U \pi(x,u) h_w(x,u) \mathrm{d}u = w^\top \nabla_\vartheta \underbrace{\int_U \pi_\vartheta(x,u) \mathrm{d}u}_{=1} = 0.$$

This shows that the expected value of $h_w(x,u)$ under the policy π_ϑ is zero for each state, from which can be concluded that h_w is generally better thought of as the *advantage function* $A^\pi(x,u) = Q^\pi(x,u) - V^\pi(x)$. In essence, this means that using only compatible features for the value function results in an approximator that can only represent the relative value of an action u in some state x correctly, but not the absolute value $Q(x,u)$. An example showing how different the value function $Q(x,u)$ and the corresponding advantage function $A(x,u)$ can look is shown in Figure 2.2. Because of this difference, the policy gradient estimate produced by just the compatible approximation will still have a high variance. To lower the variance, extra features have to be added on top of the compatible features, which take the role of modelling the difference between the advantage function $A^\pi(x,u)$ and the state-action value function $Q^\pi(x,u)$, which is in fact the value function $V^\pi(x)$. These extra features are therefore only state-dependent, as dependence on the action would introduce a bias into the gradient estimate. The state-dependent offset that is created

by these additional features is often referred to as a (reinforcement) baseline. The policy gradient theorem actually generalises to the case where a state-dependent baseline function is taken into account. Equation (2.26) would then read

$$\nabla_\vartheta J = \int_X d^\pi(x) \int_U \nabla_\vartheta \pi(x,u) \left[h_w(x,u) + b(x) \right] \mathrm{d}u\mathrm{d}x, \tag{2.28}$$

where $b(x)$ is the baseline function that can be chosen arbitrarily. Adding a baseline will not affect the unbiasedness of the gradient estimate, but can improve the accuracy of the critic's approximation and prevent an ill-conditioned projection of the value function on the compatible features ψ (Konda and Tsitsiklis, 2003). In that respect, this chapter treats w as part of the vector θ, and ψ as a subset of the features ϕ, meaning that the algorithm templates presented earlier are also suitable in the compatible feature setting. In practice, the optimal baseline, i.e. the baseline that minimises the variance in the gradient estimate for the policy π, is the value function $V^\pi(x)$ (Bhatnagar et al., 2009; Sutton et al., 2000). Peters et al. (2003) note that, in light of the policy gradient theorem that was only published many years later, the earlier idea of Gullapalli (1990) to use the temporal difference δ in the gradient used to update the policy weights can be shown to yield the true policy gradient $\nabla_\vartheta J(\vartheta)$, and hence corresponds to the policy gradient theorem with respect to Equation (2.28).

Theorem 2.2 yields a major benefit. Once a good parameterisation for a policy has been found, a parameterisation for the value function automatically follows and also guarantees convergence. Further on in this chapter, many actor-critic algorithms make use of this theorem.

Part of this chapter is dedicated to giving some examples of relevant actor-critic algorithms in both the standard gradient and natural gradient setting. As it is not possible to describe all existing actor-critic algorithms in detail, the algorithms addressed in this chapter are chosen based on their originality: either they were the first to use a certain technique, extended an existing method significantly or the containing paper provided an essential analysis. In Section 2.2 a distinction between the discounted and average reward setting was already made. The reward setting is the first major axis along which the algorithms in this chapter are organised. The second major axis is the gradient type, which will be either the standard gradient or the natural gradient. This results in a total of four categories to which the algorithms can (uniquely)

belong, see Table 2.1. References in bold are discussed from an algorithmic perspective. Section 2.4 describes actor-critic algorithms that use a standard gradient. Section 2.5 first introduces the concept of a natural gradient, after which natural actor-critic algorithms are discussed. References in italic are discussed in the Section 2.6 on applications.

Table 2.1 Actor-critic methods, categorised along two axes: gradient type and reward setting.

	Standard gradient	Natural gradient
Discounted return	**Barto et al. (1983), FACRLN (Cheng et al., 2004; Wang et al., 2007), CACM (Niedzwiedz et al., 2008), Bhatnagar (2010)**, *Li et al. (2009), Kimura et al. (2001), Raju et al. Raju et al. (2003)*	***(e)NAC (Peters and Schaal, 2008a; Peters et al., 2003)*, Park et al. (2005), Girgin and Preux (2008), Kimura (2008)**,*Richter et al. (2007), Kim et al. (2010), Nakamura et al. (2007), El-Fakdi et al. (2010)*
Average return	**Konda and Tsitsiklis (2003), *Paschalidis et al. (2009)*, ACFRL (Berenji and Vengerov, 2003; Vengerov et al., 2005), Algorithm I (Bhatnagar et al., 2009)**, *ACSMDP (Usaha and Barria, 2007)*	**Algorithms II–IV (Bhatnagar et al., 2009), gNAC (Morimura et al., 2009)**

2.4 Standard Gradient Actor-Critic Algorithms

Many papers refer to Barto et al. (1983) as the starting point of actor-critic algorithms, although there the actor and critic were called the associative search element and adaptive critic element, respectively. That paper itself mentions that the implemented strategy is closely related to the work by Samuel (1959) and Witten (1977). Either way, it is true that Barto et al. (1983)

defined the actor-critic structure that resembles the recently proposed actor-critic algorithms the most. Therefore, the discussion on standard actor-critic algorithms starts with this work, after which other algorithms in the discounted return setting follow. As many algorithms based on the average return also exist, they are dealt with in a separate section.

2.4.1 Discounted Return Setting

Barto et al. (1983) use simple parameterisations

$$V_\theta(x) = \theta^\top \phi(x) \qquad\qquad \pi_\vartheta(x) = \vartheta^\top \phi(x),$$

with the same features $\phi(x)$ for the actor and critic. They chose binary features, i.e. for some state x only one feature $\phi_i(x)$ has a non-zero value, in this case $\phi_i(x) = 1$. For ease of notation, the state x was taken to be a vector of zeros with only one element equal to 1, indicating the activated feature. This allowed the parameterisation to be written as

$$V_\theta(x) = \theta^\top x \qquad\qquad \pi_\vartheta(x) = \vartheta^\top x.$$

Then, they were able to learn a solution to the well-known cart-pole problem using the update equations

$$\delta_k = r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k) \tag{2.29a}$$
$$z_{c,k} = \lambda_c z_{c,k-1} + (1-\lambda_c) x_k \tag{2.29b}$$
$$z_{a,k} = \lambda_a z_{a,k-1} + (1-\lambda_a) u_k x_k \tag{2.29c}$$
$$\theta_{k+1} = \theta_k + \alpha_c \delta_k z_{c,k} \tag{2.29d}$$
$$\vartheta_{k+1} = \vartheta_k + \alpha_a \delta_k z_{a,k}, \tag{2.29e}$$

with

$$u_k = \tau\left(\pi_{\vartheta_k}(x_k) + n_k\right),$$

where τ is a threshold, sigmoid or identity function, n_k is noise which accounts for exploration and z_c, z_a are eligibility traces for the critic and actor, respectively. Note that these update equations are similar to the ones in Template 2.1, considering the representation in binary features. The product $\delta_k z_{a,k}$ in Equation (2.29e) can then be interpreted as the gradient of the performance with respect to the policy parameter.

Although no use was made of advanced function approximation techniques, good results were obtained. A mere division of the state space into boxes

meant that there was no generalisation among the states, indicating that learning speeds could definitely be improved upon. Nevertheless, the actor-critic structure itself remained and later work largely focused on better representations for the actor and the calculation of the critic.

Based on earlier work by Cheng et al. (2004), Wang et al. (2007) introduced the Fuzzy Actor-Critic Reinforcement Learning Network (FACRLN), which uses only one fuzzy neural network based on radial basis functions for both the actor and the critic. That is, they both use the same input and hidden layers, but differ in their output by using different weights. This is based on the idea that both actor and critic have the same input and also depend on the same system dynamics. Apart from the regular updates to the actor and critic based on the temporal difference error, the algorithm not only updates the parameters of the radial basis functions in the neural network, but also adaptively adds and merges fuzzy rules. Whenever the TD error or the squared TD error is too high and the so-called ϵ-completeness property (Lee, 1990) is violated, a new rule, established by a new radial basis function, is added to the network. A closeness measure of the radial basis functions decides whether two (or more) rules should be merged into one. For example, when using Gaussian functions in the network, if two rules have their centres and their widths close enough to each other, they will be merged into one. FACRLN is benchmarked against several other (fuzzy) actor-critic algorithms, including the original work by Barto et al. (1983), and turns out to outperform them all in terms of the number of trials needed, without increasing the number of basis functions significantly.

At about the same time, Niedzwiedz et al. (2008) also claimed, like with FACRLN, that there is redundancy in learning separate networks for the actor and critic and developed their Consolidated Actor-Critic Model (CACM) based on that same principle. They too set up a single neural network, using sigmoid functions instead of fuzzy rules, and use it for both the actor and the critic. The biggest difference is that here, the size of the neural network is fixed, so there is no adaptive insertion/removal of sigmoid functions.

More recently, work on the use of actor-critic algorithms using function approximation for discounted cost MDPs under multiple inequality constraints appeared in Bhatnagar (2010). The constraints considered are bounds on the

expected values of discounted sums of single-stage cost functions ρ_n, i.e.

$$S_n(\pi) = \sum_{x \in X} d_0(x) W_n^\pi(x) \leq s_n, \quad n = 1 \dots N,$$

with

$$W_n^\pi(x) = \mathrm{E}\left\{ \sum_{k=0}^{\infty} \gamma^k \rho_n(x_k, u_k) \middle| x_0 = x, \pi \right\}$$

and d_0 a given initial distribution over the states. The approach is, as in usual constrained optimisation problems, to extend the discounted cost function $J(\pi)$ to a Lagrangian cost function

$$L(\pi, \bar{\mu}) = J(\pi) + \sum_{n=1}^{N} \mu_k G_n(\pi),$$

where $\bar{\mu} = (\mu_1, \dots, \mu_N)^\top$ is the vector of Lagrange multipliers and $G_n(\pi) = S_n(\pi) - s_n$ the functions representing the inequality constraints.

The algorithm generates an estimate of the policy gradient using simultaneous perturbation stochastic approximation (SPSA) (Spall, 1992), which has been found to be efficient even in high-dimensional parameter spaces. The SPSA gradient requires the introduction of two critics instead of one. The first critic, parameterised by $\theta^\top \phi(x)$, evaluates a policy parameterised by ϑ_k. The second critic, parameterised by ${\theta'}^\top \phi(x)$ evaluates a slightly perturbed policy parameterised by $\vartheta_k + \epsilon \Delta_k$ with a small $\epsilon > 0$. The element-wise policy parameter update is then given by[4]

$$\vartheta_{i,k+1} = \Gamma_i \left[\vartheta_k + \alpha_a \sum_{x \in X} d_0(x) \left(\frac{(\theta_k - \theta'_k)^\top \phi(x)}{\epsilon \Delta_i(k)} \right) \right], \tag{2.30}$$

where Γ_i is a truncation operator. The Lagrange parameters μ also have an update rule of their own, which introduces a third learning rate $\alpha_{L,k}$ into the algorithm for which the regular conditions

$$\sum_k \alpha_{L,k} = \infty \qquad \sum_k \alpha_{L,k}^2 < \infty$$

[4]This requires two simultaneous simulations of the constrained MDP.

must be satisfied and another constraint relating $\alpha_{L,k}$ to the actor step size $\alpha_{a,k}$

$$\lim_{k\to\infty} \frac{\alpha_{L,k}}{\alpha_{a,k}} = 0$$

must also hold, indicating that the learning rate for the Lagrange multipliers should decrease quicker than the actor's learning rate. Under these conditions, the authors prove the almost sure convergence to a locally optimal policy.

2.4.2 Average Reward Setting

Together with the presentation of the novel ideas of compatible features, discussed in Section 2.3.4, Konda and Tsitsiklis (2003) introduced two actor-critic algorithms, differing only in the way they update the critic. The general update equations for these algorithms are

$$\widehat{J}_k = \widehat{J}_{k-1} + \alpha_{c,k}(r_{k+1} - \widehat{J}_{k-1}) \tag{2.31a}$$

$$\delta_k = r_{k+1} - \widehat{J}_k + Q_{\theta_k}(x_{k+1}, u_{k+1}) - Q_{\theta_k}(x_k, u_k) \tag{2.31b}$$

$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k z_k \tag{2.31c}$$

$$\vartheta_{k+1} = \vartheta_k + \alpha_{a,k}\Gamma(\theta_k)Q_{\theta_k}(x_k, u_k)\psi(x_k, u_k), \tag{2.31d}$$

where ψ is the vector of compatible features as defined in Equation (2.27), and the parameterisation Q_θ also contains these compatible features. The first and the second equation depict the standard update rules for the estimate of the average cost and the temporal difference error, as seen in Template 2.2. The third equation is the update of the critic. Here, the vector z_k represents an eligibility trace (Sutton and Barto, 1998) and it is exactly this what distinguishes the two different algorithms described in the paper. The first algorithm uses a TD(1) critic, basically taking an eligibility trace with decay rate $\lambda = 1$. The eligibility trace is updated as

$$z_k = \begin{cases} z_{k-1} + \phi_k(x_k, u_k) & \text{if } x_k \neq x^S \\ \phi_k(x_k, u_k) & \text{otherwise} \end{cases}$$

where x^S is a special reset state for which it is assumed that the probability of reaching it from any initial state x within a finite number of transitions is bounded away from zero for any sequence of randomised stationary policies. Here, the eligibility trace is reset when encountering a state that meets this

assumption. The second algorithm is a TD(λ) critic, simply updating the eligibility trace as

$$z_k = \lambda z_{k-1} + \phi_k(x_k, u_k).$$

The update of the actor in Equation (2.31d) uses the policy gradient estimate from Theorem 2.2. It leaves out the state distribution $d^\pi(x)$ earlier seen in Equation (2.26) of the policy gradient theorem, as the expected value of $\nabla J(\vartheta_k)$ is equal to that of $\nabla_\vartheta \pi(x,u)\widehat{Q}^\pi_w(x,u)$ and puts the critic's current estimate in place of $\widehat{Q}^\pi_w(x,u)$. Finally, $\Gamma(\theta_k)$ is a truncation term to control the step size of the actor, taking into account the current estimate of the critic. For this particular algorithm, some further assumptions on the truncation operator Γ must hold, which are not listed here.

It is known that using least-squares TD methods for policy evaluation is superior to using regular TD methods in terms of convergence rate as they are more data efficient (Boyan, 2002; Bradtke and Barto, 1996). Inevitably, this resulted in work on actor-critic methods using an LSTD critic (Peters et al., 2003; Williams et al., 2006). However, Paschalidis et al. (2009) showed that it is not straightforward to use LSTD without modification, as it undermines the assumptions on the step sizes as given by Equations (2.21) and (2.22). As a result of the basics of LSTD, the step size schedule for the critic should be chosen as $\alpha_{c,k} = \frac{1}{k}$. Plugging this demand into Equations (2.21) and (2.22) two conditions on the step size of the actor conflict:

$$\sum_k \alpha_{a,k} = \infty \qquad \sum_k (k\alpha_{a,k})^d < \infty \text{ for some } d > 0.$$

They conflict because the first requires α_a to decay at a rate slower than $1/k$, while the second demands a rate faster than $1/k$. This means there is a trade-off between the actor having too much influence on the critic and the actor decreasing its learning rate too fast. The approach presented by Paschalidis et al. (2009) to address this problem is to use the following step size schedule for the actor. For some $K \gg 1$, let $L = \lfloor k/K \rfloor$ and

$$\alpha_{a,k} := \frac{1}{L+1}\hat{\alpha}_a(k+1-LK),$$

where $\sum_k (k\hat{\alpha}_a(k))^d \leq \infty$ for some $d > 0$. As a possible example,

$$\hat{\alpha}_{a,k}(b) := \varrho(C) \cdot b^{-C}$$

is provided, where $C > 1$ and $\varrho(C) > 0$. The critic's step size schedule is redefined as

$$\alpha_{c,k} := \frac{1}{k - \kappa(L,K)}.$$

Two extreme cases of $\kappa(L,K)$ are $\kappa(L,K) \triangleq 0$ and $\kappa(L,K) = LK - 1$. The first alternative corresponds to the unmodified version of LSTD and the latter corresponds to "restarting" the LSTD procedure when k is an integer multiple of K. The reason for adding the κ term to the critic update is theoretical, as it may be used to increase the accuracy of the critic estimates for $k \to \infty$. Nevertheless, choosing $\kappa(L,K) = 0$ gave good results in simulations (Paschalidis et al., 2009). These step size schedules for the actor and critic allow the critic to converge to the policy gradient, despite the intermediate actor updates, while constantly reviving the learning rate of the actor such that the policy updates do not stop prematurely. The actor step size schedule does not meet the requirement $\sum_k (k\alpha_a)^d < \infty$ for some $d > 0$, meaning that convergence of the critic for the entire horizon cannot be directly established. What *is* proven by the authors is that the critic converges before every time instant $k = JK$, at which point a new epoch starts[5]. For the actor, the optimum is not reached during each epoch, but in the long run it will move to an optimal policy. A detailed proof of this is provided by Paschalidis et al. (2009).

Berenji and Vengerov (2003) used the actor-critic algorithm of Konda and Tsitsiklis (2003) to provide a proof of convergence for an actor-critic fuzzy reinforcement learning (ACFRL) algorithm. The fuzzy element of the algorithm is the actor, which uses a parameterised fuzzy Takagi-Sugeno rulebase. The authors show that this parameterisation adheres to the assumptions needed for convergence given by Konda and Tsitsiklis (2003), hence providing the convergence proof. The update equations for the average cost and the critic are the same as Equations (2.31a) and (2.31c), but the actor update is slightly changed into

$$\vartheta_{k+1} = \Gamma\left(\vartheta_k + \alpha_{a,k}\theta_k^\top \phi_k(x_k,u_k)\psi_k(x_k,u_k)\right),$$

where the truncation operator Γ is now acting on the complete update expression, instead of limiting the step size based on the critic's parameter. While applying ACFRL to a power management control problem, it was acknowledged that the highly stochastic nature of the problem and the

[5]The authors use the term "episode", but this might cause confusion with the commonly seen concept of episodic tasks in RL, which is not the case here.

presence of delayed rewards necessitated a slight adaptation to the original framework of Konda and Tsitsiklis (2003). The solution was to split the updating algorithm into three phases. Each phase consists of running a finite number of simulation traces. The first phase only estimates the average cost $\hat{J}$, keeping the actor and critic fixed. The second phase only updates the critic, based on the $\hat{J}$ obtained in the previous phase. This phase consists of a finite number of traces during which a fixed *positive* exploration term is used on top of the actor's output and an equal number of traces during which a fixed *negative* exploration term is used. The claim is that this systematic way of exploring is very beneficial in problems with delayed rewards, as it allows the critic to better establish the effects of a certain direction of exploration. The third and final phase keeps the critic fixed and lets the actor learn the new policy. Using this algorithm, ACFRL consistently converged to the same neighbourhood of policy parameters for a given initial parameterisation. Later, Vengerov et al. (2005) extended the algorithm to ACFRL-2, which took the idea of systematic exploration one step further by learning two separate critics: one for positive exploration and one for negative exploration.

Bhatnagar et al. (2009) introduced four algorithms. The first one uses a regular gradient and will therefore be discussed in this section. The update equations for this algorithm are

$$\widehat{J}_k = \widehat{J}_{k-1} + \alpha_{J,k}(r_{k+1} - \widehat{J}_{k-1}) \tag{2.32a}$$
$$\delta_k = r_{k+1} - \hat{J}_k + V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k) \tag{2.32b}$$
$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k\phi(x_k) \tag{2.32c}$$
$$\vartheta_{k+1} = \Gamma(\vartheta_k + \alpha_{a,k}\delta_k\psi(x_k,u_k)). \tag{2.32d}$$

The critic update is simply an update in the direction of the gradient $\nabla_\theta V$. The actor update uses the fact that $\delta_k\psi(x_k,u_k)$ is an unbiased estimate of $\nabla_\vartheta J$ under certain conditions (Bhatnagar et al., 2009). The operator Γ is a projection operator, ensuring boundedness of the actor update. Three more algorithms are discussed by Bhatnagar et al. (2009), but these make use of a natural gradient for the updates and hence are discussed in Section 2.5.3.

2.5 Natural Gradient Actor-Critic Algorithms

The previous section introduced actor-critic algorithms which use standard gradients. The use of standard gradients comes with drawbacks. Standard

gradient descent is most useful for cost functions that have a single minimum and whose gradients are isotropic in magnitude with respect to any direction away from its minimum (Amari and Douglas, 1998). In practice, these two properties are almost never true. The existence of multiple local minima of the cost function, for example, is a known problem in reinforcement learning, usually overcome by exploration strategies. Furthermore, the performance of methods that use standard gradients relies heavily on the choice of a coordinate system over which the cost function is defined. This non-covariance is one of the most important drawbacks of standard gradients (Bagnell and Schneider, 2003a; Kakade, 2001). An example for this will be given later in this section.

In robotics, it is common to have a "curved" state space (manifold), e.g. because of the presence of angles in the state. A cost function will then usually be defined in that curved space too, possibly causing inefficient policy gradient updates to occur. This is exactly what makes the *natural* gradient interesting, as it incorporates knowledge about the curvature of the space into the gradient. It is a metric based not on the choice of coordinates, but on the manifold that those coordinates parameterise (Kakade, 2001).

This section is divided into two parts. The first part explains what the concept of a natural gradient is and what its effects are in a simple optimisation problem, i.e. not considering a learning setting. The second part is devoted to actor-critic algorithms that make use of this type of gradient to update the actor. As these policy updates are using natural gradients, these algorithms are also referred to as natural policy gradient algorithms.

2.5.1 Natural Gradient in Optimisation

To introduce the notion of a natural gradient, this section summarises work presented in Amari (1998); Amari and Douglas (1998); Bagnell and Schneider (2003a). Suppose a function $J(\vartheta)$ is parameterised by ϑ. When ϑ lives in a Euclidean space, the squared Euclidean norm of a small increment $\Delta\vartheta$ is given by the inner product

$$\|\Delta\vartheta\|_E^2 = \Delta\vartheta^\top \Delta\vartheta.$$

A steepest descent direction is then defined by minimising $J(\vartheta + \Delta\vartheta)$ while keeping $\|\Delta\vartheta\|_E$ equal to a small constant. When ϑ is transformed to other coordinates $\widetilde{\vartheta}$ in a non-Euclidean space, the squared norm of a small increment

$\Delta\widetilde{\vartheta}$ with respect to that *Riemannian* space is given by the product

$$\|\Delta\widetilde{\vartheta}\|_R^2 = \Delta\widetilde{\vartheta}^\top G(\widetilde{\vartheta})\Delta\widetilde{\vartheta}$$

where $G(\widetilde{\vartheta})$ is the Riemannian metric tensor, an $n \times n$ positive definite matrix characterising the intrinsic local curvature of a particular manifold in an n-dimensional space. The Riemannian metric tensor $G(\widetilde{\vartheta})$ can be determined from the relationship (Amari and Douglas, 1998):

$$\|\Delta\vartheta\|_E^2 = \|\Delta\widetilde{\vartheta}\|_R^2.$$

Clearly, for Euclidean spaces $G(\widetilde{\vartheta})$ is the identity matrix.

Standard gradient descent for the new parameters $\widetilde{\vartheta}$ would define the steepest descent with respect to the norm $\|\Delta\widetilde{\vartheta}\|^2 = \Delta\widetilde{\vartheta}^\top\Delta\widetilde{\vartheta}$. However, this would result in a different gradient direction, despite keeping the same cost function and only changing the coordinates. The *natural* gradient avoids this problem, and always points in the "right" direction, by taking into account the Riemannian structure of the parameterised space over which the cost function is defined. So now, $\widetilde{J}(\widetilde{\vartheta}+\Delta\widetilde{\vartheta})$ is minimised while keeping $\|\Delta\widetilde{\vartheta}\|_R$ small ($\widetilde{J}$ here is just the original cost J, but written as a function of the new coordinates). This results in the natural gradient $\widetilde{\nabla}_{\widetilde{\vartheta}}\widetilde{J}(\widetilde{\vartheta})$ of the cost function, which is just a linear transformation of the standard gradient $\nabla_{\widetilde{\vartheta}}\widetilde{J}(\widetilde{\vartheta})$ by the inverse of $G(\widetilde{\vartheta})$:

$$\widetilde{\nabla}_{\widetilde{\vartheta}}\widetilde{J}(\widetilde{\vartheta}) = G^{-1}(\widetilde{\vartheta})\nabla_{\widetilde{\vartheta}}\widetilde{J}(\widetilde{\vartheta}).$$

As an example of optimisation with a standard gradient versus a natural gradient, consider a cost function based on polar coordinates

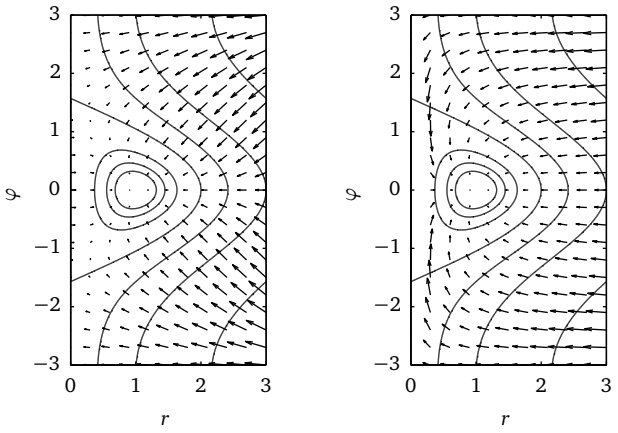
$$J_P(r,\varphi) = \frac{1}{2}\left[(r\cos\varphi - 1)^2 + r^2\sin^2\varphi\right]. \tag{2.33}$$

This cost function is equivalent to $J_E(x,y) = (x-1)^2 + y^2$, where x and y are Euclidean coordinates, so the relationship between (r,φ) and (x,y) is given by

$$x = r\cos\varphi \qquad\qquad y = r\sin\varphi.$$

Figure 2.3a shows the contours and antigradients of $J_P(r,\varphi)$ for $0 \le r \le 3$ and $|\varphi| \le \pi$, where

$$-\nabla_{(r,\varphi)}J_P(r,\varphi) = -\begin{bmatrix} r - \cos\varphi \\ r\sin\varphi \end{bmatrix}.$$



(a) Standard gradient (b) Natural gradient

Figure 2.3 Standard and natural antigradients of the cost function $J_P(r,\varphi)$ in polar coordinates.

The magnitude of the gradient clearly varies widely over the (r,φ)-plane. When performing a steepest descent search on this cost function, the trajectories from any point (r,φ) to an optimal one will be far from straight paths. For the transformation of Euclidean coordinates into polar coordinates, the Riemannian metric tensor is (Amari and Douglas, 1998)

$$G(r,\varphi) = \begin{bmatrix} 1 & 0 \\ 0 & r^2 \end{bmatrix},$$

so that the natural gradient of the cost function in (2.33) is

$$\begin{aligned} -\widetilde{\nabla}_{(r,\varphi)} J_P(r,\varphi) &= -G(r,\varphi)^{-1} \nabla_{(r,\varphi)} J_P(r,\varphi) \\ &= -\begin{bmatrix} r - \cos\varphi \\ \dfrac{\sin\varphi}{r} \end{bmatrix}. \end{aligned}$$

Figure 2.3b shows the natural gradients of $J_P(r,\varphi)$. Clearly, the magnitude of the gradient is now more uniform across the space and the angles of the gradients also do not greatly vary away from the optimal point $(1,0)$.

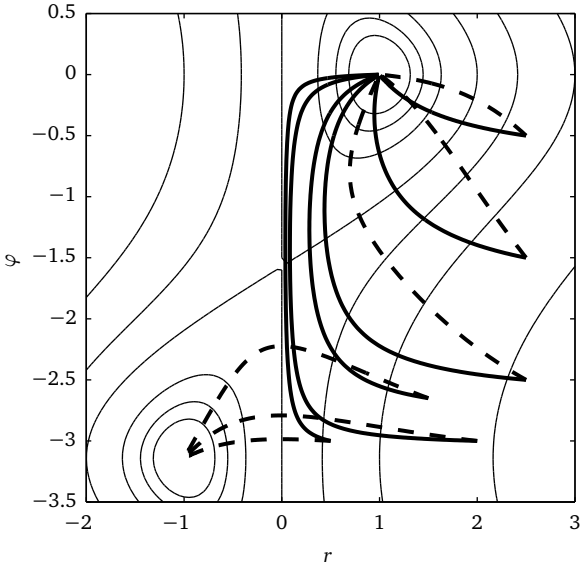


Figure 2.4 Trajectories for standard gradient (dashed) and natural gradient (solid) algorithms for minimising $J_P(r, \varphi)$ in polar coordinates.

Figure 2.4 shows the difference between a steepest descent method using a standard gradient and a natural gradient on the cost $J_P(r, \varphi)$ using a number of different initial conditions. The natural gradient clearly performs better as it always finds the optimal point, whereas the standard gradient generates paths that are leading to points in the space which are not even feasible, because of the radius which needs to be positive.

To get an intuitive understanding of what the effect of a natural gradient is, Figure 2.5 shows trajectories for the standard and natural gradient that have been transformed onto the Euclidean space. Whatever the initial condition[6] is, the natural gradient of $J_P(r, \varphi)$ always points straight to the optimum and follows the same path that the standard gradient of $J_E(x, y)$ would do.

When $J(\vartheta)$ is a quadratic function of ϑ (like in many optimisation problems,

[6] The exemplified initial conditions are not the same as in Figure 2.4.

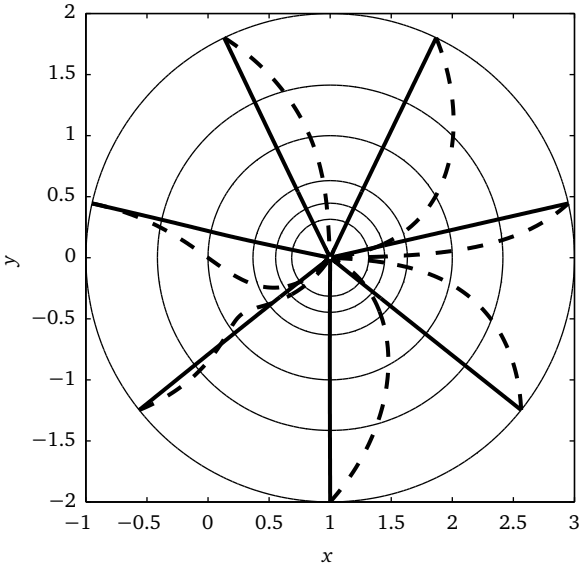


Figure 2.5 Trajectories for standard gradient (dashed) and natural gradient (solid) algorithms for minimising $J_P(r, \varphi)$, transformed to Euclidean coordinates.

including for example those solved in supervised learning), the Hessian $H(\vartheta)$ is equal to $G(\vartheta)$ for the underlying parameter space, and there is no difference between using Newton's method and natural gradient optimisation. In general, however, natural gradient optimisation differs from Newton's method, since $G(\vartheta)$ is always positive definite by construction, whereas the Hessian $H(\vartheta)$ may not be (Amari and Douglas, 1998). The general intuition developed in this section is essential before moving on to the natural policy gradient in MDPs, explained next.

2.5.2 Natural Policy Gradient

The possibility of using natural gradients in online learning was first appreciated in Amari (1998). As shown above, the crucial property of the natural gradient is that it takes into account the structure of the manifold over which

the cost function is defined, locally characterised by the Riemannian metric tensor. To apply this insight in the context of *policy* gradient methods, the main question is then what an appropriate manifold is, and once that is known, what its Riemannian metric tensor is.

Consider first just the parameterised stochastic policy $\pi_\vartheta(x,u)$ at a single state x; a probability distribution over the actions u. This policy is a point on a manifold of such probability distributions, found at coordinates ϑ. For a manifold of distributions, the Riemannian tensor is the so-called Fisher information matrix (FIM) (Amari, 1998), which for the policy above is (Kakade, 2001)

$$\begin{aligned} F(\vartheta,x) &= \mathrm{E}\left[\nabla_\vartheta \ln \pi_\vartheta(x,u) \nabla_\vartheta \ln \pi_\vartheta(x,u)^\top\right] \\ &= \int_U \pi_\vartheta(x,u) \nabla_\vartheta \ln \pi_\vartheta(x,u) \nabla_\vartheta \ln \pi_\vartheta(x,u)^\top \mathrm{d}u. \end{aligned} \tag{2.34}$$

The single-state policy is directly related with the expected immediate reward, over a single step from x. However, it does not tell much about the overall expected return $J(\pi)$, which is defined over entire state trajectories. To obtain an appropriate overall FIM, in the average reward case, Kakade (2001) made the intuitive choice of taking the expectation of $F(\vartheta,x)$ with respect to the stationary state distribution $d^\pi(x)$

$$F(\vartheta) = \int_X d^\pi(x) F(\vartheta,x) \mathrm{d}x. \tag{2.35}$$

He was, however, unsure whether this was the right choice.

Later on, Peters et al. (2003) and Bagnell and Schneider (2003a) independently showed that Equation (2.35) is indeed a true FIM, for the manifold of *probability distributions over trajectories* in the MDP. When used to control the MDP with stochastic dynamics f, $\pi_\vartheta(x,u)$ gives rise to different controlled trajectories with different probabilities, so each value of the parameter ϑ yields such a distribution over trajectories. To understand how this distribution is relevant to the value $J(\pi)$ of the policy, observe that this value can be written as the expected value of the infinite-horizon return over all possible paths, where the expectation is taken with respect to precisely the trajectory distribution. Furthermore, Peters et al. (2003) and Bagnell and Schneider (2003a) show that this idea also extends to the discounted reward case, where the FIM is still given by Equation (2.35), only with $d^\pi(x)$ replaced by the discounted state distribution $d^\pi_\gamma(x)$, as defined in Section 2.2.1.

Examples of the difference in performance between regular policy gradients and natural policy gradients are provided by Bagnell and Schneider (2003a); Kakade (2001); Peters et al. (2003).

2.5.3 Natural Actor-Critic Algorithms

This section describes several representative actor-critic algorithms that employ a natural policy gradient. Again, a distinction is made between algorithms using the discounted return and the average return.

Discounted Return Setting

After the acknowledgement by Amari (1998) that using the natural gradient could be beneficial for learning, the aptly called Natural Actor-Critic algorithm by Peters et al. (2003) was the first actor-critic algorithm that successfully employed a natural gradient for the policy updates. Together with Kakade (2001), they gave a proof that the natural gradient $\widetilde{\nabla}_{\vartheta} J(\vartheta)$ is in fact the compatible feature parameter w of the approximated value function, so

$$\widetilde{\nabla}_{\vartheta} J(\vartheta) = w.$$

Consequently, they were able to use a natural gradient without explicitly calculating the Fisher Information Matrix. This turns the policy update step into

$$\vartheta_{k+1} = \vartheta_k + \alpha_a \widetilde{\nabla}_{\vartheta} J(\vartheta) \tag{2.36a}$$

$$= \vartheta_k + \alpha_a w_{k+1}. \tag{2.36b}$$

For the policy evaluation step of the algorithm, i.e. the calculation of the critic parameter w, LSTD-Q(λ) was used, which was their own extension to LSTD(λ) by Boyan (2002). The Natural Actor-Critic outperformed standard gradient policy gradient methods on a cart-pole balancing setup. Later, the work was extended by Peters and Schaal (2008a), where it was shown that several well-known reinforcement algorithms (for example, the actor-critic algorithm of Sutton and Barto (1998) and the Q-learning algorithm of Bradtke et al. (1994)) are strongly related to natural actor-critic algorithms. Furthermore, the paper presents the successful application of an episodic variant of Natural Actor-Critic (eNAC) on an anthropomorphic robot arm. For another example of a natural-actor critic algorithm with a regression-based critic, see Melo and Lopes (2008).

Park et al. (2005) extend the original work of Peters et al. (2003) by using a recursive least-squares method in the critic, making the parameter estimation of the critic more efficient. They then successfully apply it to the control of a two-link robot arm.

Girgin and Preux (2008) improve the performance of natural actor-critic algorithms, by using a neural network for the actor, which includes a mechanism to automatically add hidden layers to the neural network if the accuracy is not sufficient. Enhancing the eNAC method of Peters and Schaal (2008a) with this basis expansion method clearly showed its benefits on a cart-pole simulation.

Though a lot of (natural) actor-critic algorithms use sophisticated function approximators, Kimura (2008) showed that a simple policy parameterisation using rectangular coarse coding can still outperform conventional Q-learning in high-dimensional problems. In the simulations, however, Q-learning did outperform the natural actor-critic algorithm in low-dimensional problems.

Average Reward Setting

Bhatnagar et al. (2009) introduced four algorithms, three of which are natural-gradient algorithms. They extend the results of Konda and Tsitsiklis (2003) by using temporal difference learning for the actor and by incorporating natural gradients. They also extend the work of Peters and Schaal (2008a) by providing the first convergence proofs and the first fully incremental natural actor-critic algorithms. The contribution of convergence proofs for natural-actor critic algorithms is important, especially since the algorithms utilised both function approximation and a bootstrapping critic, a combination which is essential to large-scale applications of reinforcement learning. The second algorithm only differs from the first algorithm, described at the end of Section 2.4.2 with Equation (2.32), in the actor update (2.32d). It directly substitutes the standard gradient with the natural gradient, yielding

$$\vartheta_{k+1} = \Gamma(\vartheta_k + \alpha_{a,k} F_k^{-1}(\vartheta)\delta_k \psi(x_k, u_k)), \tag{2.37}$$

where F is the Fisher Information Matrix (FIM). This requires the actual calculation of the FIM. Since the FIM can be written using the compatible features ψ as

$$F(\vartheta) = \int_X d^\pi(x) \int_U \pi(x,u)\psi(x,u)\psi^\top(x,u)\mathrm{d}u\mathrm{d}x,$$

sample averages can be used to compute it with

$$F_k(\vartheta) = \frac{1}{k+1} \sum_{l=0}^{k} \psi(x_l, u_l) \psi^\top(x_l, u_l).$$

After converting this equation to a recursive update rule, and putting the critic's learning rate in place, the Sherman-Morrison matrix inversion lemma is used to obtain an iterative update rule for the inverse of the FIM[7].

$$F_k^{-1}(\vartheta) = \frac{1}{1-\alpha_{c,k}} \left[F_{k-1}^{-1} - \alpha_{c,k} \frac{(F_{k-1}^{-1}\psi_k)(F_{k-1}^{-1}\psi_k)^\top}{1-\alpha_{c,k}(1-\psi_k^\top F_{k-1}^{-1}\psi_k)} \right],$$

where the initial value F_0^{-1} is chosen to be a scalar multiple of the identity matrix. This update rule, together with the adjusted update of the actor then form the second algorithm.

The third algorithm by Bhatnagar et al. (2009) uses the fact that the compatible approximation $w^\top \psi(x,u)$ is better thought of as an advantage function approximator instead of a state-action value function approximator, as mentioned in Section 2.3.4. Hence, the algorithm tunes the weights w, such that the squared error $\mathcal{E}^\pi(w) = \mathrm{E}\left[(w^\top \psi(x,u) - A^\pi(x,u))^2\right]$ is minimised. The antigradient of this error is

$$\nabla_w \mathcal{E}^\pi(w) = 2 \sum_X d^\pi(x) \sum_U \pi(x,u) \left[w^\top \psi(x,u) - A^\pi(x,u) \right] \psi(x,u).$$

As δ_k is an unbiased estimate of $A^\pi(x_k, u_k)$ (see Bhatnagar et al. (2008)), the gradient is estimated with

$$\widehat{\nabla_w \mathcal{E}^\pi}(w) = 2(\psi_k \psi_k^\top w - \delta_k \psi_k) \tag{2.38}$$

and the gradient descent update rule for w (using the same learning rate as the critic) is

$$w_{k+1} = w_k - \alpha_{c,k}(\psi_k \psi_k^\top w_k - \delta_k \psi_k). \tag{2.39}$$

Furthermore, the natural gradient estimate is given by w (as shown by Peters and Schaal (2008a)), and an explicit calculation for the FIM is no longer

[7] For readability, $\psi(x_k, u_k)$ is replaced by ψ_k for the remainder of this section.

necessary. Therefore, the third algorithm is obtained by using Equation (2.39) and replacing the actor in Equation (2.37) with

$$\vartheta_{k+1} = \Gamma(\vartheta_k + \alpha_{a,k} w_{k+1}). \tag{2.40}$$

The fourth algorithm by Bhatnagar et al. (2009) is obtained by combining the second and third algorithm. The explicit calculation of F_k^{-1} is now used for the update of the compatible parameter w. The update of w now also follows its natural gradient, by premultiplying the result in Equation (2.38) with F_k^{-1}, giving

$$\widehat{\widetilde{\nabla}_w \mathcal{E}^\pi}(w) = 2F_k^{-1}(\psi_k \psi_k^\top w - \delta_k \psi_k),$$

turning the update of w into

$$\begin{aligned} w_{k+1} &= w_k - \alpha_{c,k} F_k^{-1}(\psi_k \psi_k^\top w_k - \delta_k \psi_k) \\ &= w_k - \alpha_{c,k} \underbrace{F_k^{-1} \psi_k \psi_k^\top}_{I} w_k + \alpha_{c,k} F_k^{-1} \delta_k \psi_k \\ &= w_k - \alpha_{c,k} w_k + \alpha_{c,k} F_k^{-1} \delta_k \psi_k, \end{aligned}$$

where clever use is made of the fact that F_k is written as the squared ψ's. The actor update is still Equation (2.40).

Although most natural actor-critic algorithms use the natural gradient as defined in Section 2.5, the generalised Natural Actor-Critic (gNAC) algorithm by Morimura et al. (2009) does not. Instead, a *generalised* natural gradient (gNG) is used, which combines properties of the Fisher Information Matrix and natural gradient as defined before with the properties of a differently defined Fisher Information Matrix and natural gradient from the work by Morimura et al. (2008). They consider the fact that the average reward $J(\vartheta)$ is not only affected by the policy π, but also by the resulting state distribution $d^\pi(x)$ and define the Fisher Information Matrix of the state-action joint distribution as

$$F_{\mathrm{SA}}(\vartheta) = F_{\mathrm{S}}(\vartheta) + F_{\mathrm{A}}(\vartheta), \tag{2.41}$$

where $F_{\mathrm{S}}(\vartheta)$ is the FIM of the stationary state distribution $d^\pi(x)$ and $F_{\mathrm{A}}(\vartheta)$ the FIM as defined in Equation (2.35). Morimura et al. (2008) consider the use of $F_{\mathrm{SA}}(\vartheta)$ as the FIM for learning better than using the original FIM because of three reasons: (i) Learning with $F_{\mathrm{SA}}(\vartheta)$ still benefits from the concepts of natural gradient, since it necessarily and sufficiently accounts for

the probability distributions that the average reward depends on. (ii) $F_{\mathrm{SA}}(\vartheta)$ is analogous to the Hessian matrix of the average reward. (iii) Numerical experiments have shown a strong tendency of avoiding plateaus in learning.

Nevertheless, the original FIM $F_{\mathrm{A}}(\vartheta)$ accounts for the distribution over an infinite amount of time steps, whereas $F_{\mathrm{SA}}(\vartheta)$ only accounts for the distribution over a single time step. This might increase the mixing time of the Markov chain drastically, making it hard for the RL learning agent to estimate a gradient with a few samples. Therefore, the authors suggest to use a weighted average, using a weighting factor ι, of both FIM's defined in Equations (2.34) and (2.41). The gNG is then calculated by using the inverse of this weighted average, leading to the policy gradient

$$\widetilde{\nabla}_{\vartheta} J(\vartheta) = \left(\iota F_{\mathrm{S}} + F_{\mathrm{A}}\right)^{-1} \nabla_{\vartheta} J(\vartheta).$$

The implementation of the algorithm is similar to that of NAC, with the slight difference that another algorithm, $\mathcal{LS}$LSD (Morimura et al., 2010), is used to estimate $\nabla_{\vartheta} d^{\pi}(x)$. If $\iota = 0$, gNAC is equivalent to the original NAC algorithm of Peters et al. (2003), but now optimising over the average return instead of the discounted return. In a numerical experiment with a randomly synthesised MDP of 30 states and 2 actions, gNAC with $\iota > 0$ outperformed the original NAC algorithm.

2.6 Applications

This section provides references to papers that have applied actor-critic algorithms in several domains. Note that the list of applications is not exhaustive and that other application domains for actor-critic algorithms and more literature on the applications mentioned below exists.

In the field of robotics, early successful results of using actor-critic type methods on real hardware were shown on a ball on a beam setup (Benbrahim et al., 1992), a peg-in-hole insertion task (Gullapalli, 1993) and biped locomotion (Benbrahim and Franklin, 1997). Peters and Schaal (2008a) showed that their natural actor-critic method was capable of getting an anthropomorphic robot arm to learn certain motor skills (see Figure 2.6). Kim et al. (2010) recently successfully applied a modified version of the algorithm by Park et al. (2005) to motor skill learning. Locomotion of a two-link robot arm was learned using a recursive least-squares natural actor-critic method

Figure 2.6 The episodic Natural Actor-Critic method (Peters and Schaal, 2008a) applied to an anthropomorphic robot arm performing a baseball bat swing task.

in Park et al. (2005). Another successful application on a real four-legged robot is given by Kimura et al. (2001). Nakamura et al. (2007) devised an algorithm based on the work by Peters and Schaal (2008a) which made a biped robot walk stably. Underwater cable tracking (El-Fakdi et al., 2010) was done using the NAC method of Peters and Schaal (2008a), where it was used in both a simulation and real-time setting: once the results from simulation were satisfactory, the policy was moved to an actual underwater vehicle, which continued learning during operation, improving the initial policy obtained from simulation.

An example of a logistics problem solved by actor-critic methods is given by Paschalidis et al. (2009), which successfully applies such a method to the problem of dispatching forklift trucks in a warehouse. This is a high-dimensional problem because of the number of products, forklift trucks and depots involved. Even with over 200 million discrete states, the algorithm was able to converge to a solution that performed 20% better in terms of cost than a heuristic solution obtained by taking the exact solution of a smaller problem

and expanding this to a large state space.

Usaha and Barria (2007) use the algorithm from Konda and Tsitsiklis (2003) described in Section 2.4.2, extended to handle semi-Markov decision processes[8], for call admission control in lower earth orbit satellite networks. They compared the performance of this actor-critic semi-Markov decision algorithm (ACSMDP) together with an optimistic policy iteration (OPI) method to an existing routing algorithm. While both ACSMDP and OPI outperform the existing routing algorithm, ACSMDP has an advantage in terms of computational time, although OPI reaches the best result. Based on the FACRLN from Wang et al. (2007) in Section 2.4.1, Li et al. (2009) devised a way to control traffic signals at an intersection and showed in simulation that this method outperformed the commonly seen time slice allocation methods. Richter et al. (2007) showed similar improvements in road traffic optimisation when using natural actor-critic methods.

Finally, an application to the finance domain was described by Raju et al. (2003), where older work on actor-critic algorithms (Konda and Borkar, 1999) was applied in the problem of determining dynamic prices in an electronic retail market.

2.7 Discussion

When applying reinforcement learning to a certain problem, knowing a priori whether a critic-only, actor-only or actor-critic algorithm will yield the best control policy is virtually impossible. However, a few rules of thumb should help in selecting the most sensible class of algorithms to use. The most important thing to consider first is the type of control policy that should be learned. If it is necessary for the control policy to produce actions in a continuous space, critic-only algorithms are no longer an option, as calculating a control law would require solving the possibly non-convex optimisation procedure of Equation (2.11) over the continuous action space. Conversely, when the controller only needs to generate actions in a (small) countable, finite space, it makes sense to use critic-only methods, as Equation (2.11) can be solved by enumeration. Using a critic-only method also overcomes the problem

[8] Semi-Markov decision processes extend regular MDPs by taking into account a (possibly stochastically) varying transition time from one state to another.

of high-variance gradients in actor-only methods and the introduction of more tuning parameters (e.g. extra learning rates) in actor-critic methods.

Choosing between actor-only and actor-critic methods is more straightforward. If the problem is modelled by a (quasi-)stationary MDP, actor-critic methods should provide policy gradients with lower variance than actor-only methods. Actor-only methods are however more resilient to fast changing non-stationary environments, in which a critic would be incapable of keeping up with the time-varying nature of the process and would not provide useful information to the actor, cancelling the advantages of using actor-critic algorithms. In summary, actor-critic algorithms are most sensibly used in a (quasi-)stationary setting with a continuous state and action space.

Once the choice for actor-critic has been made, the issue of choosing the right features for the actor and critic, respectively, remains. There is consensus, though, that the features for the actor and critic do not have to be chosen independently. Several actor-critic algorithms use the exact same set of features for both the actor and the critic, while the policy gradient theorem indicates that it is best to first choose a parameterisation for the actor, after which *compatible features* for the critic can be derived. In this sense, the use of compatible features is beneficial as it *lessens* the burden of choosing a separate parameterisation for the value function. Note that compatible features do not *eliminate* the burden of choosing features for the value function completely (see Section 2.3.4). Adding state-dependent features to the value function on top of the compatible features remains an important task as this is the only way to further reduce the variance in the policy gradient estimates. How to choose these additional features remains a difficult problem.

Choosing a good parameterisation for the policy in the first place also remains an important issue as it highly influences the performance after learning. Choosing this parameterisation does seem less difficult than for the value function, as in practice it is easier to get an idea what shape the policy has than the corresponding value function.

One of the conditions for successful application of reinforcement learning in practice is that learning should be quick. Although this chapter and other parts of this thesis focus on gradient-based algorithms and how to estimate this gradient, it should be noted that it is not only the quality of the gradient estimate that influences the speed of learning. Balancing the exploration and exploitation of a policy and choosing good learning rate schedules also have a

large effect on this, although more recently expectation-maximisation (EM) methods that work without learning rates have been proposed (Kober and Peters, 2011; Vlassis et al., 2009). With respect to gradient type, the natural gradient seems to be superior to the standard gradient. However, an example of standard Q-learning on low-dimensional problems by Kimura (2008) and relative entropy policy search (REPS) (Peters et al., 2010) showed better results than the natural gradient. Hence, even though the field of natural gradient actor-critic methods is still a very promising area for future research, it does not always show superior performance compared to other methods. A number of applications which use natural gradients are mentioned in this chapter. The use of compatible features makes it straightforward to calculate approximations of natural gradients, which implies that any actor-critic algorithm developed in the future should attempt to use this type of gradient, as it speeds up learning without any real additional computational effort.

Chapter

3

Efficient Model Learning Actor-Critic Methods

The previous chapter introduced the concept of actor-critic reinforcement learning algorithms and described several example algorithms. All the algorithms discussed so far do not employ models of any kind of the controlled process. In this chapter, two new actor-critic algorithms for reinforcement learning are proposed. A crucial feature of these two algorithms is that they learn a process model online, which provides an efficient policy update for faster learning.

The first algorithm uses a novel model-based update rule for the actor parameters. The second algorithm does not use an explicit actor, but learns a reference model which represents a desired behaviour, from which control actions can be calculated using the local inverse of the learned process model. It should be noted that even though these methods use a process model, they are still considered model-free methods, as there is no prior knowledge of the system available at the start of learning.

The two novel methods and a standard actor-critic algorithm are applied to the pendulum swing-up problem, in which the novel model-learning methods achieve faster learning than the standard algorithm. Moreover, the algorithms are capable of handling both parametric and non-parametric function approximators and a comparison is made between using radial basis functions (RBFs), which is parametric, and local linear regression (LLR), which

is non-parametric, as the function approximator in these algorithms.

3.1 Introduction and Related Work

Many processes in industry can potentially benefit from control algorithms that learn to optimise a certain cost function, when synthesising a controller analytically is tough or impossible. Reinforcement learning can offer a good solution, but it typically starts without any knowledge of the process and has to improve its behaviour through trial and error. Because of this, the process goes through a long period of unpredictable and potentially damaging behaviour. This is usually unacceptable in industry and the long period of trial and error learning must be considerably reduced for RL controllers to become useful in practice.

This chapter introduces two novel algorithms that employ an efficient policy update which increases the learning speed considerably compared to standard actor-critic methods. The first novel algorithm learns a process model and employs it to update the actor. However, instead of using the process model to generate simulated experiences as most model learning RL algorithms do (Kuvayev and Sutton, 1996; Moore and Atkeson, 1993; Sutton, 1992), it uses the model to directly calculate an accurate policy gradient, which accelerates learning compared to other policy gradient methods.

The second novel algorithm not only learns a process model, but also a reference model which represents desired closed-loop behaviour by mapping states to subsequent desired states. The reference model and inverse process model are then coupled to serve as an overall actor, which is used to calculate new inputs to the system. This second algorithm is similar to "learning from relevant trajectories" (Atkeson and Schaal, 1997), in which LLR is used to learn the process model of a robotic arm holding a pendulum. The process model is then employed to control the arm along a demonstrated trajectory that effectively swings up the pendulum. The main difference is that in this chapter a trajectory is not demonstrated to the learning controller, but a reference model, representing the desired closed-loop behaviour, is learned and updated online.

The next sections first describe the benchmark algorithm and the novel methods, after which a number of possible function approximators are discussed. The chapter closes with an analysis of simulation results.

3.2 Standard Actor-Critic

A standard temporal difference based actor-critic method described by Buşoniu et al. (2010) serves as a baseline to compare the novel model learning methods to. This baseline will be referred to as the standard actor-critic (SAC) algorithm. The algorithm uses the regular critic update with eligibility traces given by the discounted return Template 2.1, with a heuristic estimate (Bhatnagar et al., 2009; Buşoniu et al., 2010) for the policy gradient $\nabla_\vartheta J$, giving

$$\nabla_\vartheta J(x_k) \approx \delta_k \Delta u_k \nabla_\vartheta \pi_\vartheta(x_k),$$

in which Δu_k is the random exploration term, drawn from a zero-mean normal distribution, that was added to the policy's output at time k. This results in the update rule for the actor in SAC being

$$\vartheta_{k+1} = \vartheta_k + \alpha_a \delta_k \Delta u_k \nabla_\vartheta \pi_\vartheta(x_k). \tag{3.1}$$

The product of the exploration term Δu_k and the temporal difference error δ_k serves as a sign switch for the gradient $\nabla_\vartheta \pi_\vartheta(x_k)$. When the exploration Δu_k leads to a positive TD error, the direction of exploration is deemed beneficial to the performance and the policy is adjusted towards the perturbed action. Conversely, when δ_k is negative, the policy is adjusted away from this perturbation.

The diagram of SAC is shown in Figure 2.1. For clarity, the pseudocode of the SAC algorithm is given in Algorithm 1.

3.3 Model Learning Actor-Critic

The standard actor-critic algorithm described in the previous section is, like many standard reinforcement learning algorithms, inefficient in its use of measured data. Once a sample, containing the previous state, the action taken and the subsequent state, has been used to update the actor and critic, it is thrown away and never reused in future updates. To overcome this problem, several techniques have been proposed to remember and reuse measured data, such as experience replay (Adam et al., 2011; Lin, 1992; Wawrzyński, 2009) and prioritised sweeping (Moore and Atkeson, 1993). A drawback of these methods is that they require storage of all the samples gathered, making them memory intensive and computationally heavy. Dyna

Algorithm 1 Standard Actor-Critic (SAC)

Input: γ, λ and learning rates α

$z_0 = 0$
Initialise x_0 and function approximators
Apply random input u_0
$k \leftarrow 0$
loop
    Measure x_{k+1}, r_{k+1}
    $\delta_k \leftarrow r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k)$
    *// **Choose action / update actor***
    $u_{k+1} \leftarrow \pi_{\vartheta_k}(x_{k+1})$
    $\vartheta_{k+1} \leftarrow \vartheta_k + \alpha_a \delta_k \Delta u_k \nabla_\vartheta \pi_\vartheta(x_k)$
    *// **Update critic***
    $z_k \leftarrow \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k)$
    $\theta_{k+1} \leftarrow \theta_k + \alpha_c \delta_k z_k$
    Choose exploration $\Delta u_{k+1} \sim \mathcal{N}(0, \sigma^2)$
    Apply $u_{k+1} + \Delta u_{k+1}$
    $k \leftarrow k + 1$
end loop

architectures (Sutton, 1990) combine reinforcement learning with the concept of planning, by learning a model of the process or environment online and using this model to generate experiences from which the critic (and thus the policy) can be updated. This results in more frequent updates and hence quicker learning.

In MLAC, the learned process model is not used to generate experiences. Instead, the process model is used directly in the policy gradient, aiming to get faster convergence of learning without increasing the number of updates for the actor and/or critic.

3.3.1 The Process Model

Whereas SAC only learns the actor and critic functions, MLAC also learns an approximate process model $x' = \hat{f}_\zeta(x, u)$. In case of a parametric function approximator, like radial basis functions (RBFs), the process model is parameterised by $\zeta \in \mathbb{R}^{r \cdot n}$, where r is the number of basis functions per

element of process model output used for the approximation and n is the state dimension.

Having a learned process model available simplifies the update of the actor, as it allows to predict what the next state x' will be, given some input u. The value function then provides information on the value $V(x')$ of the next state x'. The best action u to choose in state x would be

$$u = \arg\max_{u' \in U} \left[\rho(x, u', \hat{f}_\zeta(x, u')) + \gamma V(\hat{f}_\zeta(x, u')) \right]. \tag{3.2}$$

Using a model for the purpose of predicting a next state and using this information for updates of a policy and/or value function is also called *planning* (Sutton and Barto, 1998).

However, since the action space is assumed to be continuous, it is impossible to enumerate over all possible inputs u and therefore a policy gradient, directly employing the process model, is put into place.

3.3.2 Model-Based Policy Gradient

With appropriately chosen function approximators, the gradient of the value function with respect to the state x and the Jacobian of the process model with respect to the input u can be estimated. Then, by applying the chain rule, the Jacobian of the value function with respect to the input u becomes available.

The value function[1] adheres to the Bellman equation

$$V(x_k) = \rho(x_k, \pi(x_k), x_{k+1}) + \gamma V(x_{k+1}). \tag{3.3}$$

The learned process model $\widehat{f}_\zeta$ allows predictions for x_{k+1} by taking

$$\widetilde{x}_{k+1} = \widehat{f}_\zeta(x_k, \widetilde{u}_k) \tag{3.4}$$

where the input $\widetilde{u}_k$ is chosen according the current policy, i.e.

$$\widetilde{u}_k = \pi_\vartheta(x_k). \tag{3.5}$$

Combining Equations (3.3)–(3.5) provides the approximation

$$V(x_k) \approx \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1}) + \gamma V(\widetilde{x}_{k+1}). \tag{3.6}$$

[1] The value function is assumed to be exact here.

Performing gradient ascent using the gradient of Equation (3.6) corresponds to moving the policy towards choosing greedy actions according to Equation (3.2).

The gradient of $V(x)$ with respect to the policy parameter ϑ, evaluated at x_k is then given by

$$\begin{aligned}\nabla_\vartheta V(x_k) &= \nabla_\vartheta \left[\rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1}) + \gamma V(\widetilde{x}_{k+1})\right] \\ &= \nabla_\vartheta \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1}) + \gamma \nabla_\vartheta V(\widetilde{x}_{k+1}).\end{aligned}$$

The term related to the reward function ρ can be expanded as

$$\begin{aligned}\nabla_\vartheta \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1}) = \nabla_u \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1})^\top \nabla_\vartheta \pi_\vartheta(x_k) \\ + \nabla_{x_{k+1}} \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \nabla_\vartheta \pi_\vartheta(x_k).\end{aligned} \tag{3.7}$$

The term related to the value function V expands to

$$\gamma \nabla_\vartheta V(\widetilde{x}_{k+1}) = \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \nabla_\vartheta \pi_\vartheta(x_k). \tag{3.8}$$

Adding up Equations (3.7) and (3.8), the gradient $\nabla_\vartheta V(x_k)$ is obtained.

$$\begin{aligned}\nabla_\vartheta V(x_k) &= \nabla_u \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1})^\top \nabla_\vartheta \pi_\vartheta(x_k) \\ &\quad + \nabla_{x_{k+1}} \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \nabla_\vartheta \pi_\vartheta(x_k) \\ &\quad + \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \nabla_\vartheta \pi_\vartheta(x_k) \\ &= \left\{ \left[\nabla_{x_{k+1}} \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1})^\top + \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \right] \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \right. \\ &\quad \left. + \nabla_u \rho(x_k, \widetilde{u}_k, \widetilde{x}_{k+1})^\top \right\} \nabla_\vartheta \pi_\vartheta(x_k).\end{aligned}$$

In case ρ is not dependent on the future state x_{k+1}, i.e. $\rho(x_k, u_k, x_{k+1}) = \rho(x_k, u_k)$ this simplifies to

$$\nabla_\vartheta V(x_k) = \left\{ \nabla_u \rho(x_k, \widetilde{u}_k)^\top + \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \right\} \nabla_\vartheta \pi_\vartheta(x_k).$$

Since for the discounted case it holds that $J(x) = V(x)$, this allows the policy gradient $\nabla_\vartheta J$ to be approximated by

$$\nabla_\vartheta J(x_k) \approx \left\{ \nabla_u \rho(x_k, \widetilde{u}_k)^\top + \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, \widetilde{u}_k) \right\} \nabla_\vartheta \pi_\vartheta(x_k). \tag{3.9}$$

The process model itself is updated by applying a gradient descent update, using the error between the real output of the system and the output of the

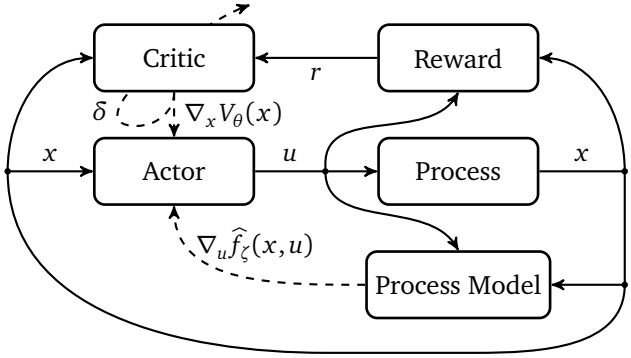


Figure 3.1 Schematic overview of MLAC.

process model, scaled by a learning rate α_p, so

$$\zeta_{k+1} = \zeta_k + \alpha_p(x_{k+1} - \widehat{f}_{\zeta_k}(x_k, u_k))\nabla_\zeta \widehat{f}_{\zeta_k}(x_k, u_k).$$

Here, the input is simply u_k and not $\widetilde{u}_k$ as prescribed by the policy, since exploration may have been used during this particular time step.

Note that with the SAC algorithm, exploration is needed in the actor update (3.1). Because MLAC uses the gradient in Equation (3.9) it knows in what direction to update the actor such that higher state values will be encountered, without having to perform exploratory actions. As a result, the MLAC algorithm can estimate the policy gradient, and learn the task, without exploration. Exploration is nevertheless still needed because it gives a more complete value function over the entire state space, whereas without exploration the current policy would only visit a part of the state space. Finally, exploration improves the model of the process dynamics. A schematic overview of the MLAC algorithm is given in Figure 3.1 and its pseudocode is given in Algorithm 2.

3.4 Reference Model Actor-Critic

Reference Model Actor-Critic (RMAC) is different from the typical actor–critic methods in the sense that it does not learn an explicit mapping from state x_k to action u_k. Instead of an explicit actor, RMAC learns a reference model

Algorithm 2 Model Learning Actor-Critic (MLAC)

Input: γ, λ and learning rates α

$z_0 = 0$
Initialise x_0 and function approximators
Apply random input u_0
$k \leftarrow 0$
loop
  Measure x_{k+1}, r_{k+1}
  $\delta_k \leftarrow r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k)$
  // Choose action / update actor
  $u_{k+1} \leftarrow \pi_{\vartheta_k}(x_{k+1})$
  $\vartheta_{k+1} \leftarrow \vartheta_k + \alpha_a \nabla_\vartheta J(x_k) \approx \left\{ \nabla_u \rho(x_k, u_k)^\top + \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, u_k) \right\} \nabla_\vartheta \pi_\vartheta(x_k)$
  // Update process model
  $\zeta_{k+1} \leftarrow \zeta_k + \alpha_p (x_{k+1} - \widehat{f}_{\zeta_k}(x_k, u_k)) \nabla_\zeta \widehat{f}_{\zeta_k}(x_k, u_k)$
  // Update critic
  $z_k \leftarrow \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k)$
  $\theta_{k+1} \leftarrow \theta_k + \alpha_c \delta_k z_k$
  Choose exploration $\Delta u_{k+1} \sim \mathcal{N}(0, \sigma^2)$
  Apply $u_{k+1} + \Delta u_{k+1}$
  $k \leftarrow k + 1$
end loop

that represents a desired behavior of the system, based on the value function. Similar to MLAC, this algorithm learns a process model, through which it identifies a desired next state x' with the highest possible value $V(x')$. The difference with respect to MLAC is that an actor, mapping a state x onto an action u, is not explicitly stored. Instead, the reference model is used in combination with the inverse of the learned process model to calculate the action u.

Using a reference model provides a means for the storage of demonstration data. Some learning algorithms benefit from having the desired behaviour or task demonstrated to them (see, e.g., Khansari-Zadeh and Billard (2011)). This can be done, for example, by a human manually moving a robot arm in such a way that a target task is performed. The demonstrated trajectory is then stored

as a sequence of (sampled) states and it is exactly this type of information that can be stored in a reference model.

The parameterised reference model $R_\eta(x)$, with parameter $\eta \in \mathbb{R}^{s\times n}$, where s is the number of basis functions per element of reference model output used for the approximation and n is the state dimension, maps the state x_k to a desired next state $\widehat{x}_{k+1}$, i.e.

$$\widehat{x}_{k+1} = R_{\eta_k}(x_k).$$

The process is controlled towards this desired next state by using the inverse of the learned process model $x_{k+1} = \widehat{f}_{\zeta_k}(x_k, u_k)$. The reference model $R_{\eta_k}(x_k)$ and the inverse process model $u_k = \widehat{f}^{-1}_{\zeta_k}(x_k, x_{k+1})$ together act as a policy, by using the relation $u_k = \widehat{f}^{-1}_{\zeta_k}(x_k, R_{\eta_k}(x_k))$. This does require that the process model, or more specifically the function approximator that represents it, is (at least locally) invertible. Here, invertibility is achieved by using a first order Taylor expansion of the process model around the point (x_k, u_{k-1}), so

$$\widehat{x}_{k+1} \approx \widehat{f}_{\zeta_k}(x_k, u_{k-1}) + \nabla_u \widehat{f}_{\zeta_k}(x_k, u_{k-1})(u_k - u_{k-1}).$$

From this equation, u_k can be directly calculated, given x_k, $\widehat{x}_{k+1}$ and u_{k-1}. The reason for taking the operating point (x_k, u_{k-1}) is that the control signal u is assumed to be smooth, meaning that u_{k-1} is close to u_k which is a main requirement for an accurate approximation using a Taylor series.

The introduction of a reference model also requires the introduction of an update rule for the reference model's parameters. A natural update rule for these parameters is to move them in the direction that will yield the highest value, i.e.

$$\eta_{k+1} = \eta_k + \alpha_r \nabla_x V(x_{k+1})^\top \nabla_\eta R_{\eta_k}(x_k) \tag{3.10}$$

where $\alpha_r > 0$ is the learning rate of the reference model. Update (3.10) may eventually lead to an infeasible reference model if the output of $R_\eta(x)$ is not kept within the reachable set

$$\mathcal{R}_x = \left\{ x' \in X | x' = f(x,u), u \in U \right\},$$

which is the set of all states that can be reached from the current state x within a single sampling interval.

It is not straightforward to determine this set because it depends on the current state, the (nonlinear) process dynamics and the action space U.

To overcome this problem, it is assumed that the reachable set $\mathcal{R}_x$ can be defined by only using the extreme values of the action space U. Defining the set U_e as the finite discrete set containing all the combinations of extreme values[2] of U, the learned process model can be used to calculate the estimated next state when applying these extreme values, as the reachable set $\mathcal{R}_x$ is now reduced to

$$\widehat{\mathcal{R}}_x = \left\{ x' \in X | x' = \widehat{f}_\zeta(x,u), u \in U_e \right\}$$

As an example of what a set U_e can look like, consider a system which takes two inputs at every time step, i.e. $u = (u_1, u_2)$. The set U_e would then be

$$U_e = \left\{ (u_{1,\max}, u_{2,\max}), (u_{1,\max}, u_{2,\min}), (u_{1,\min}, u_{2,\max}), (u_{1,\min}, u_{2,\min}) \right\}.$$

Subsequently, the critic provides the estimated value of those next states and the state that yields the highest value is selected as the next desired state[3]

$$x_r = \arg\max_{x' \in \widehat{\mathcal{R}}_x} V_\theta(x').$$

This procedure is justified by the assumption that if the sampling interval is short enough, both the process model and critic can be approximated locally by a linear function. As linear functions always have their extreme values on the edges of their input domain, only the values of next states x' that are reached by using extreme values of the input have to be checked.

The state x_r is then used in the update the reference model

$$\eta_{k+1} = \eta_k + \alpha_r (x_r - \widehat{x}_{k+1}) \nabla_\eta R_{\eta_k}(x_k). \tag{3.11}$$

Because of the approximation of $\mathcal{R}_x$ the reference model will be updated by a desired state x_r that is the result of applying the extremes of u. However, by using the learning rate α_r in the update of $R_\eta(x)$, a smooth reference model and a smooth policy can still be achieved. This approximation does mean, though, that the quality of the solution brought up by the algorithm is compromised and a more accurate calculation of the reachable set should improve the performance.

[2] This requires the action space U to be a hyperbox.

[3] The reasons for not using a maximisation over the expression $\rho(x,u) + V(x')$ are that a quadratic reward function is used and the assumption that $|u_{\min}| = |u_{\max}|$, leading to $\rho(x, u_{\min}) = \rho(x, u_{\max})$.

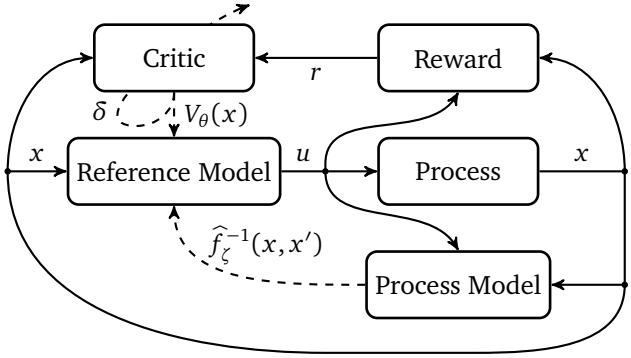


Figure 3.2 Schematic overview of RMAC.

Learning the reference model online and from scratch might pose convergence issues, since another bootstrapping approximation comes into play. Nonetheless, convergence is likely to be achieved when following the same reasoning that standard actor-critic algorithms use. For those algorithms, convergence is ensured as long as the actor is updated on a slower timescale than the critic (Borkar, 1997; Konda and Tsitsiklis, 2003). Now that the reference model is taking up the role of the actor, it reasonable to assume that keeping the learning rate of the reference model below that of the critic should also give convergence.

In contrast to SAC, the RMAC improves the reference model using (3.11) which does not involve exploration. Instead, it improves the reference model on the basis of previous experiences, but just like with MLAC exploration is still needed to get a more complete value function over the entire state space and a more complete model of the process dynamics. The scheme of the RMAC algorithm is given in Figure 3.2 and its pseudocode in Algorithm 3.

3.5 Function Approximators

Three different function approximators have been used to test the performance of the actor-critic algorithms presented in this chapter. The first two approximator types, radial basis functions (RBFs) and local linear regression (LLR), can be applied to SAC, MLAC and RMAC. The third function approximator, tile

Algorithm 3 Reference Model Actor-Critic (RMAC)

Input: γ, λ and learning rates α

$z_0 = 0$
Initialise x_0 and function approximators
Apply random input u_0
$k \leftarrow 0$
loop
    Measure x_{k+1}, r_{k+1}
    $\delta_k \leftarrow r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k)$
    // Choose action / update reference model
    $\widehat{x}_{k+2} \leftarrow R_{\eta_k}(x_{k+1})$
    $u_{k+1} \leftarrow \widehat{f}^{-1}_{\zeta_k}(x_{k+1}, \widehat{x}_{k+2})$
    Select best reachable state x_r from x_k
    $\eta_{k+1} \leftarrow \eta_k + \alpha_r(x_r - \widehat{x}_{k+1})\nabla_\eta R_{\eta_k}(x_k)$
    // Update process model
    $\zeta_{k+1} \leftarrow \zeta_k + \alpha_p(x_{k+1} - \widehat{f}_{\zeta_k}(x_k, u_k))\nabla_\zeta \widehat{f}_{\zeta_k}(x_k, u_k)$
    // Update critic
    $z_k \leftarrow \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k)$
    $\theta_{k+1} \leftarrow \theta_k + \alpha_c \delta_k z_k$
    Choose exploration $\Delta u_{k+1} \sim \mathcal{N}(0, \sigma^2)$
    Apply $u_{k+1} + \Delta u_{k+1}$
    $k \leftarrow k + 1$
end loop

coding, can only be applied to SAC and *not* to MLAC and RMAC, as these model learning algorithms need approximators that are continuously differentiable, which is not the case for tile coding because of its binary features. In other words, tile coding can not provide gradient information.

RBFs readily fit the parametric description of the algorithms presented in the previous sections because they form a parametric approximator. Section 3.5.2 describes LLR, which is a *non-parametric* function approximator. Therefore, some of the theory of the algorithms changes slightly, which will also be discussed in the section on LLR. Tile coding, described in Section 3.5.3, is only used in SAC and fits the description in Section 3.2 without further adjustments to the theory.

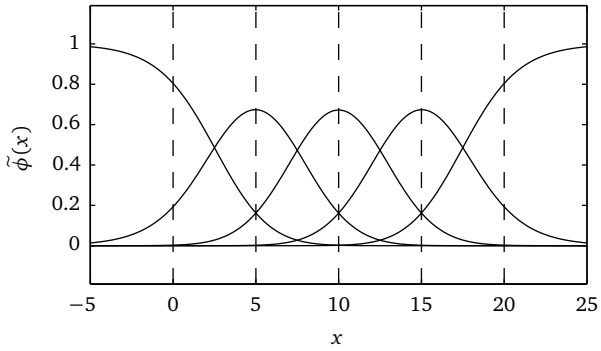


Figure 3.3 One-dimensional example of a grid of five normalised RBFs. The dashed lines show a uniform distribution of the centers.

3.5.1 Radial Basis Functions

The first type of approximator is a linear combination of normalised RBFs. The critic, for example, is modeled by

$$V_\theta(x) = \theta^\top \phi(x),$$

where $\phi(x)$ is a column vector with the value of normalised RBFs, of which each element is given by

$$\phi_i(x) = \frac{\widetilde{\phi}_i(x)}{\sum_j \widetilde{\phi}_j(x)}, \tag{3.12}$$

with

$$\widetilde{\phi}_i(x) = e^{-\frac{1}{2}(x-c_i)^\top B^{-1}(x-c_i)}, \tag{3.13}$$

where c_i are the centers of the RBFs and B is a diagonal matrix defining the widths of the RBFs. Figure 3.3 shows a one-dimensional example of a grid of RBFs.

For MLAC and RMAC, the gradient of the approximated functions is needed. With the normalisation done in Equation (3.12), this gradient is

$$\nabla_x \phi_i(x) = -\phi_i(x) \left[B^{-1}(x - c_i) + \frac{\sum_j \nabla_x \widetilde{\phi}_j(x)}{\sum_j \widetilde{\phi}_j(x)} \right].$$

The setup used for the experiments later in this chapter is a system with input saturation. This means that for the real system, the process gradient $\nabla_u f(x_k, u_k) = 0$ when u_k is outside or on the boundary of the allowed input range. As MLAC learns a process model, this input saturation needs to be dealt with when learning. Otherwise, a good policy will not be produced. In the RBF case, this problem is dealt with by setting the gradient $\nabla_u \widehat{f}_\zeta(x_k, u_k)$ in Equation (3.9) to zero when u_k is close to the input saturation bounds. This is measured by taking the Euclidean norm of the difference between the saturation bounds and the current input, which should not exceed a certain limit defined by the algorithm. Note that this approach does require prior knowledge on the bounds of the input space, which is generally available.

3.5.2 Local Linear Regression

The algorithms introduced in this chapter may also use local linear regression (LLR) as a function approximator. LLR is a non-parametric memory-based method for approximating nonlinear functions.

Memory-based methods are also called case-based, exemplar-based, lazy, instance-based or experience-based (Wettschereck et al., 1997; Wilson and Martinez, 2000). It has been shown that memory-based learning can work in RL and can quickly approximate a function with only a few observations. This is especially useful at the start of learning. Memory-based learning methods have successfully been applied to RL before, mostly as an approximator for the value function (Gabel and Riedmiller, 2005; Neumann and Peters, 2009) and in some cases also for the process model (Forbes and Andre, 2002; Ng et al., 2006).

The main advantage of memory-based methods is that the user does not need to specify a global structure or predefine features for the (approximate) model. Instead of trying to fit a global structure to observations of the unknown function, LLR simply stores the observations in a memory.

Given a memory of a certain size, filled with input/output samples, it is possible to estimate the output for any arbitrary "query input", by:

1. Finding the k samples in the memory that are nearest to the query input, according to a (weighted) distance metric.

2. Fitting a linear model to these k samples by performing a least squares fit.

3. Applying this model to the query input.

More formally, a stored observation is called a sample $s_i = [x_i^\top \mid y_i^\top]^\top$ with $i = 1, \ldots, N$. One sample s_i is a column vector containing the input data $x_i \in \mathbb{R}^n$ and output data $y_i \in \mathbb{R}^m$. The samples are stored in a matrix called the memory M with size $(n+m) \times N$ whose columns each represent one sample.

When a query x_q is made, LLR uses the stored samples to give a prediction $\hat{y}_q$ of the true output y_q. The prediction is computed by finding a local neighbourhood of x_q in the samples stored in memory. This neighbourhood is found by applying a weighted distance metric d_i (for example, the 1-norm or 2-norm) to the query point x_q and the input data x_i of all samples in M. The weighting is used to scale the inputs x and has a large influence on the resulting neighbourhood and thus on the accuracy of the prediction. Searching through the memory for nearest neighbour samples is computationally expensive. Here, a simple sorting algorithm was used, but one can reduce the computational burden by using, for instance, k-d trees (Bentley and Friedman, 1979).

By selecting a limited number of K samples with the smallest distance d, a subset $\mathcal{K}(x_q)$ with the indices of nearest neighbour samples is created. Only these K nearest neighbours are then used to make a prediction of $\hat{y}_q$. The prediction is computed by fitting a linear model to these nearest neighbours. Applying the resulting linear model to the query point x_q yields the predicted value $\hat{y}_q$. An example of this is given in Figure 3.4.

When the K nearest samples have been selected, they are split into an input matrix X and an output matrix Y, defined as

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_K \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$
$$Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_K \end{bmatrix}.$$

The last row of X consists of ones to allow for a bias on the output, making the model affine instead of truly linear.

Most of the time, the X and Y matrices form an over-determined set of equations

$$Y = \beta X$$

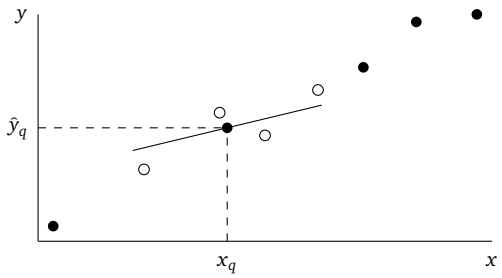


Figure 3.4 One-dimensional example of local linear regression. A least-squares fit on the four nearest neighbours of x_q yields the estimate output $\hat{y}_q$.

which can be solved for the model parameter matrix $\beta \in \mathbb{R}^{m\times(n+1)}$ by, for example, the least squares method using the right pseudo inverse of X, giving

$$\beta = YX^\top(XX^\top)^{-1}.$$

Finally, the model parameter matrix β is used to compute the predicted output corresponding to the query x_q in the simple linear equation

$$\hat{y}_q = \beta x_q.$$

As a result, the globally nonlinear function is approximated locally by a linear function. At the start of a trial, the matrices X and Y will not yet form a fully determined set of equations. In this case, there are infinitely many solutions and β will be chosen as the solution with the smallest norm or can even be set to the trivial zero solution and the value y_q will be set to the value of the closest neighbour.

Memory-based methods directly incorporate new observations, which makes it possible to get a good local estimate of the function after incorporating only a few observations. Note that if every observation were stored, the memory would grow indefinitely and so would the computational effort of finding $\mathcal{K}(x_q)$. One has to apply memory management to keep the memory from growing past a certain size. Several approaches can be implemented for this, for example:

- Purge the oldest samples in the memory.
- If two samples are too close to each other, one of them can be removed.
- Remove a sample if it is representing a value close to what would have been yielded by local linear regression on its neighbours.

The use of LLR comes with a few assumptions. The first and foremost one is that the approximated function should be smooth enough so that it can be captured by locally linear models. Any function with discontinuities or other non-smooth behaviour will be tough to approximate. This also depends on the maximum possible number of samples in the LLR memory. This number should be large enough so that the neighbourhood in which a locally linear model is calculated is small enough, i.e. the linear model is indeed local enough. More specifically when applying LLR in RL algorithms, the sampling time used should be small enough so that a locally linear model calculated at one time step is still good enough at the next time step. This is because the model is also used for predictions at the next time step.

The saturation issue with the process model learning pictured earlier when discussing RBFs also holds in the LLR case. Here, however, a solution for keeping the values bounded is much more straightforward. As the LLR memories hold input/output samples, it is possible to saturate the output part of the actor's memory, such that it can never produce inputs u beyond the saturation bound. Because of this, setting $\nabla_u \widehat{f}(x_k, u_k) = 0$ when u_k is close to the input saturation bounds is not necessary here.

Application to SAC

The only two LLR memories involved in SAC are the critic memory and the actor memory. The critic memory M^{C} holds samples of the form $s_i = [x_i{}^\top \mid V_i \mid z_i]^\top$ with $i = 1, \ldots, N^{\mathrm{C}}$ and stores states with their associated value and also keeps track of the eligibility trace. The actor memory M^{A} has samples $s_i = [x_i{}^\top \mid u_i{}^\top]^\top$ with $i = 1, \ldots, N^{\mathrm{A}}$ and stores the mapping from states to actions.

At the start of learning, both memories are empty. During learning, the LLR memories are updated in two ways:

1. By inserting the last observed sample into the memory, as the most up-to-date knowledge should be incorporated in any approximation calculated from the memory.

2. By adjusting the output parts of the nearest neighbour samples s_i that relate to some query point x_q.

The exact method of updating them is explained in more detail in the remainder of this section.

Translating Algorithm 1 to an LLR implementation, yields the following steps in order to update the actor and critic memories. First, the input-output sample $\left[x_k^\top \mid V_{\theta_k}(x_k) \mid 1\right]$ is added to the critic memory, setting its eligibility to 1. Then, the value $V_{\theta_k}(x_{k+1})$ is evaluated such that a temporal difference can be calculated. The samples corresponding to x_k and its nearest neighbours are then updated using the temporal difference error δ_k, by adjusting only their output parts by the increment $\alpha_c \delta_k$. After one update step, the eligibility trace value for all samples is discounted by λ.

The actor is updated in the same way: the sample $[x_k^\top \mid u_k^\top]$ is added to the memory. This sample and its nearest neighbours will then have their output parts adjusted by the increment $\alpha_a \delta_k \Delta u_k$. Should this update result in samples that represent actions outside of the allowable input range, then these output values are saturated in the memory directly after the update, such that a query on the memory will always return allowable inputs.

Application to MLAC

For MLAC, one extra LLR memory has to be introduced in addition to the actor and critic memories that were also used in SAC. This is the process memory M^{P}, which has samples $s_i = [x_i^\top \; u_i^\top \mid x_i'^\top]^\top$ with $i = 1, \ldots, N^{\mathrm{P}}$, where x' denotes the observed next state, i.e. $x' = f(x, u)$.

The LLR memory of the critic is updated in the same way as in SAC. With the process model in place, however, the actor update is replaced by the model-based policy gradient described earlier. That is, the actor is updated by using the local gradients of the reward function, value function and process model to obtain a gradient of the right hand side of the Bellman equation with respect to a chosen input u. By adjusting the input u in the direction given by this gradient, the actor is trying to move the policy towards a greedy policy with

respect to the value function:

$$u_i \leftarrow u_i + \alpha_a \left\{ \nabla_u \rho(x_k, u_k)^\top + \gamma \nabla_x V(\widetilde{x}_{k+1})^\top \nabla_u \widehat{f}_\zeta(x_k, u_k) \right\} \qquad \forall i \in \mathcal{K}(x) \quad (3.14)$$

Recall that $\widetilde{x}_{k+1}$ is given by the state transition function $\widetilde{x}_{k+1} = f(x, \widetilde{u})$, which is approximated here by $\hat{f}$, based on the process model memory M^{P}. Furthermore, $\widetilde{u} = \pi(x)$, based on the actor's memory M^{A}.

The value function is approximated by LLR which estimates a local linear model on the basis of previous observations of $V(x)$. The local linear model is of the form

$$\begin{aligned} V(x) &= \beta^{\mathrm{C}} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \beta_x^{\mathrm{C}} & \beta_b^{\mathrm{C}} \end{bmatrix} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}. \end{aligned}$$

This model has an input vector of length $n+1$ (n state dimensions plus a bias), a scalar output V and a model parameter matrix β^{C} of size $1 \times (n+1)$. The gradient $\frac{\partial V}{\partial x}$ is the part of β^{C} that relates the input x to the output V. This part is denoted as β_x^{C} and has size $1 \times n$.

The gradient $\frac{\partial x'}{\partial u}$ can be found by LLR on previous observations of the process dynamics. The local linear process model is of the form:

$$\begin{aligned} x' = \hat{f}(x,u) &= \beta^{\mathrm{P}} \cdot \begin{bmatrix} x \\ u \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \beta_x^{\mathrm{P}} & \beta_u^{\mathrm{P}} & \beta_b^{\mathrm{P}} \end{bmatrix} \cdot \begin{bmatrix} x \\ u \\ 1 \end{bmatrix}. \end{aligned}$$

This model has an input vector of length $n+m+1$ (n state dimensions, m action dimensions plus a bias), an output vector x' of length n and a model parameter matrix β^{C} of size $n \times (n+m+1)$. The gradient $\frac{\partial x'}{\partial u}$ is the part of β^{P} that relates u to x', denoted as β_u^{P} and has size $n \times m$.

Now, β_x^{C}, β_u^{P} and (3.14) can be used to improve the actor by adapting the nearest neighbour samples with

$$u_i \leftarrow u_i + \alpha \left\{ \nabla_u \rho(x_k, u_k)^\top + \gamma \beta_x^{\mathrm{C}} \beta_u^{\mathrm{P}} \right\} \qquad \forall i \in \mathcal{K}(x).$$

Application to RMAC

The reference model M^{R} has samples $s_i = [x_i^\top \mid \hat{x}_i^\top]^\top$ with $i = 1, \ldots, N^{\mathrm{R}}$, where $\hat{x}$ denotes a desired next state. The process model is still represented by an LLR memory, i.e.

$$x_{k+1} = \hat{f}(x_k, u_k) = \underbrace{\begin{bmatrix} \beta_x^{\mathrm{P}} & \beta_u^{\mathrm{P}} & \beta_b^{\mathrm{P}} \end{bmatrix}}_{\beta^{\mathrm{P}}} \cdot \begin{bmatrix} x_k \\ u_k \\ 1 \end{bmatrix}$$

By replacing x_{k+1} in this equation with the desired next state $\hat{x}_{k+1} = R(x_k)$ as given by the reference model and inverting the process model the action

$$u_k = (\beta_u^{\mathrm{P}^\top} \beta_u^{\mathrm{P}})^{-1} \beta_u^{\mathrm{P}^\top} \cdot \left(R(x_k) - \beta_x^{\mathrm{P}} x_k - \beta_b^{\mathrm{P}} \right)$$

is obtained. $R(x)$ is improved by adapting the desired state $\hat{x}$ of the nearest neighbour samples s_i ($i \in \mathcal{K}(x_{k-1})$) towards higher state-values using the gradient update rule

$$\hat{x}_i \leftarrow \hat{x}_i + \alpha \left. \frac{\partial V}{\partial x} \right|_{x=x'} \qquad i \in \mathcal{K}(x_{k-1}) \tag{3.15}$$

then calculate which of these states yields the highest value, using the local linear model of the value function

$$V(x) = \beta^{\mathrm{C}} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}.$$

The state x_r that corresponds to the highest value is then used to update the reference model $R(x)$, giving

$$\begin{aligned} x_r &= \arg\max_{x \in \mathcal{X}_R} \beta^{\mathrm{C}} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix} \\ \hat{x}_i &\leftarrow \hat{x}_i + \alpha_r (x_r - \hat{x}) \quad i \in \mathcal{K}(x_{k-1}). \end{aligned}$$

Because of the approximation of X_R the reference model will be updated by a desired state x_r that is the result of applying the extremes of u. However, the discussion in Section 3.4 also applies here and by using the learning rate α_r in the update of $R(x)$ a smooth reference model and a smooth policy can still be achieved.

The above method to update samples already present in the reference model is also used to insert new samples into the reference model. Once the desired state x_r has been calculated for the state x_{k-1}, the sample $[x_{k-1}^\top \mid x_r^\top]^\top$ is inserted into the reference model memory. This means that the reference model is completely learned and updated from scratch, since it can initialise itself this way online without using prior knowledge. However, initialising the reference model with samples learned from a demonstrated trajectory offline is still possible if they are available.

LLR inserts new experiences (samples) directly into the (initially empty) memory, whereas parameterised function approximators need to have initial values set for the parameters that are incrementally adjusted. LLR also allows for broad generalisation over states at the start of learning when the number of collected samples is low. As the sample density grows, the neighbourhood of the local model grows smaller and the scope of generalisation decreases. This causes LLR to work better than the parametric function approximators at the start of learning.

3.5.3 Tile Coding

Tile coding is a classical function approximator commonly used in RL, also allowing for fast computation. It uses a limited number of tilings which each divide the space into a number of tiles. The distribution of the tiles is often the uniform grid-like distribution—which is the approach used in this chapter—but any tile shape and distribution are possible. An illustration of a tile coding example is shown in Figure 3.5.

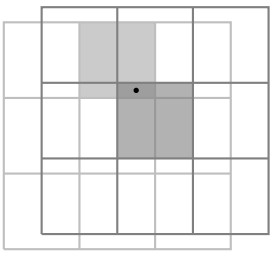


Figure 3.5 Tile coding example. The dot represents a point in the state space. Two tilings each have one (shaded) tile to which that particular point belongs.

A point in the state space either belongs to a tile or not, meaning that the tiles are in fact binary features (Sutton and Barto, 1998). The average of the parameter values of the T tiles that the state belongs to is used to compute the prediction.

$$\hat{y}_q = \frac{1}{T}\sum_{i=1}^{T}\theta_i$$

3.6 Example: Pendulum Swing-Up

To evaluate and compare the performance of the model learning algorithms, they are applied to the task of learning to swing up an inverted pendulum and benchmarked against the standard algorithm. The swing-up task was chosen because it is a low-dimensional, but challenging, highly nonlinear control problem still commonly used in RL literature (Arruda and Von Zuben, 2011; Pazis and Lagoudakis, 2011). As the process has two states and one action it allows for easy visualisation of the functions of interest. A description and a picture of this system is given in Appendix A.1.

The task is to learn to swing the pendulum from the pointing-down position to the upright position as quickly as possible and stabilise it in this position. The (fully measurable) state x consists of the angle φ and the angular velocity $\dot{\varphi}$ of the pendulum:

$$x = \begin{bmatrix} \varphi \\ \dot{\varphi} \end{bmatrix}$$

The actuation signal u is limited to $u \in [-3, 3]$ V, making it impossible to directly move the pendulum to the upright position. Instead, the controller has to learn to increase the momentum of the pendulum by swinging it back and forth before it can push it up.

A continuous quadratic reward function ρ is used to define the swing-up task. This reward function has its maximum in the upright position $[0\ 0]^\top$ and quadratically penalises non-zero values of φ, $\dot{\varphi}$ and u.

$$\rho(x_k, u_k) = -{x_k}^\top Q x_k - P u_k^2 \tag{3.16}$$

with

$$Q = \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix} \qquad P = 1$$

The standard actor-critic method SAC and the two novel methods MLAC and RMAC are applied to the pendulum swing-up problem described above, using different function approximation techniques. The algorithms run for 30 minutes of simulated time, consisting of 400 consecutive trials with each trial lasting 3 seconds. The pendulum needs approximately 1 second to swing up with a near-optimal policy. Every trial begins in the upside down position with zero angular velocity, $x_0 = [\pi\ 0]^\top$. A *learning experiment* is defined as one complete run of 400 consecutive trials.

The sum of rewards received per trial is plotted over the time which results in a learning curve. This procedure is repeated for 30 complete learning experiments to get an estimate of the mean, maximum, minimum and 95% confidence interval of the learning curve.

Tuning of all algorithm/approximator combinations was done by iterating over a grid of parameters, which included parameters of the algorithm (i.e. the learning rates) as well as parameters of the function approximator (neighborhood sizes in case of LLR and widths of RBFs, for example). For all experiments, the sampling time was set to 0.03 s, the reward discount rate γ to 0.97 and the decay rate of the eligibility trace to $\lambda = 0.65$. Exploration is done every time step by randomly perturbing the policy with normally distributed zero mean white noise with standard deviation $\sigma = 1$, so

$$\Delta u \sim \mathcal{N}[0, 1].$$

The next sections will describe the results of the simulation, ordered by algorithm. All the results shown are from simulations that used an optimal set of parameters after tuning them over a large grid of parameter values. For each algorithm, the simulation results are discussed separately for each type of function approximator. At the end of each algorithm section, the results for all function approximators are compared. Section 3.7 will compare the results of all algorithms.

3.6.1 Standard Actor-Critic

Tile Coding

The tile coding (Section 3.5.3) setup for SAC consists of 16 tilings, each being a uniform grid of 7×7 tiles. The partitions are equidistantly distributed over each dimension of the state space.

Large exploratory actions appear to be beneficial for learning. This can be explained by the fact that the representation of the value function by tile coding is not perfect. There is a constant error in the approximation causing the temporal difference to continuously vary around a certain level. For small exploratory actions, their contribution to the resulting temporal difference is small compared to the contribution of the approximation error. This causes the update of the actor by (3.1) to be very noisy.

The SAC algorithm, using the actor learning rate $\alpha_a = 0.005$ and the critic learning rate $\alpha_c = 0.1$, applied to the pendulum swing-up task results in the learning curve as shown in Figure 3.6. Examples of the final tile coding approximations of $V(x)$ and $\pi(x)$ after a representative learning experiment are shown in Figure 3.7 and Figure 3.8.

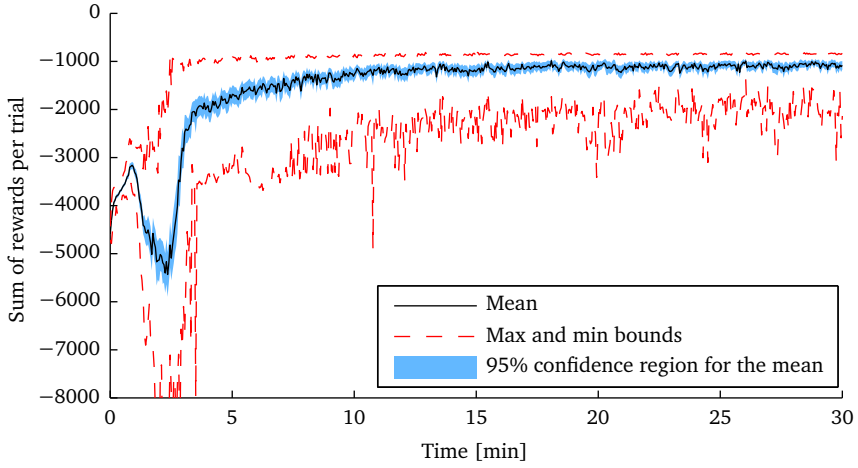


Figure 3.6 Results for the SAC algorithm using tile coding. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

The method takes, on average, about 10 minutes of simulated time to converge. One striking characteristic of the curve in Figure 3.6 is the short drop in performance in the first minutes. This can be explained by the fact that the value function is initialised to zero, which is too optimistic compared to the true value function. As a result, the algorithm collects a lot of negative rewards before it eventually learns the true value of “bad” states and adapts

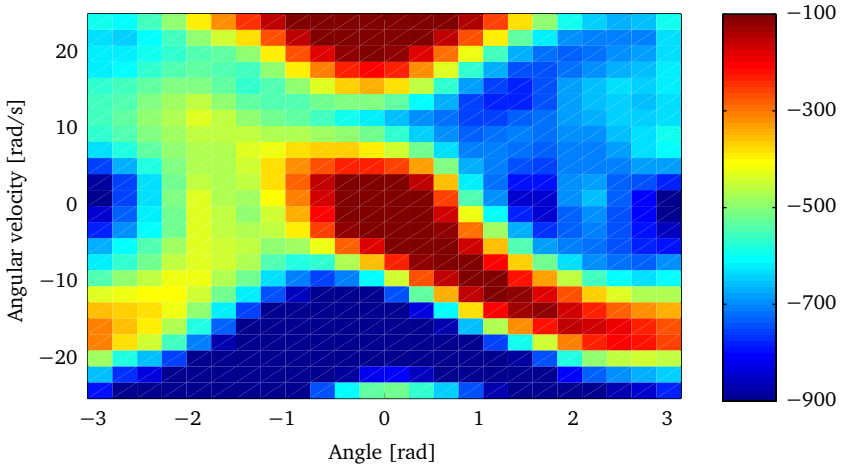


Figure 3.7 Final critic $V(x)$ for the SAC algorithm after one learning experiment.

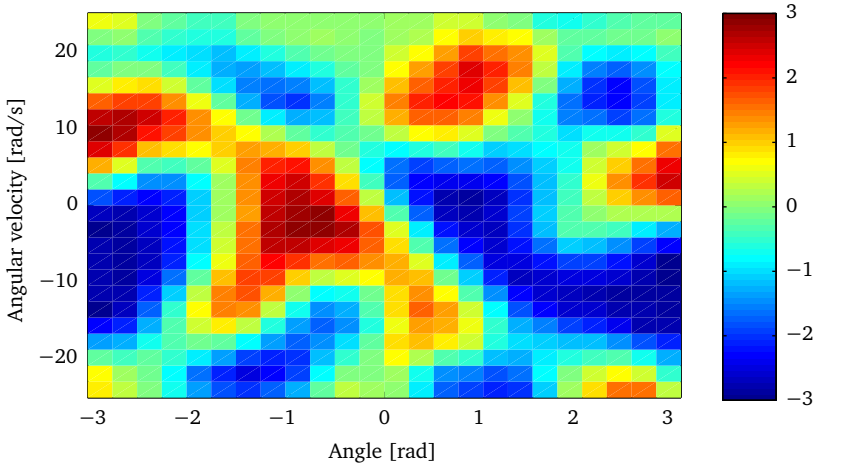


Figure 3.8 Final actor $\pi(x)$ for the SAC algorithm after one learning experiment.

the actor to avoid these. In order to prevent this initial drop in performance, the value function can be initialised with low values, but this decreases the overall learning speed as all new unvisited states are initially assumed to be bad and are avoided. An example of this is given in Figure 3.9, where the value function has been initialised pessimistically, by using the infinite discounted

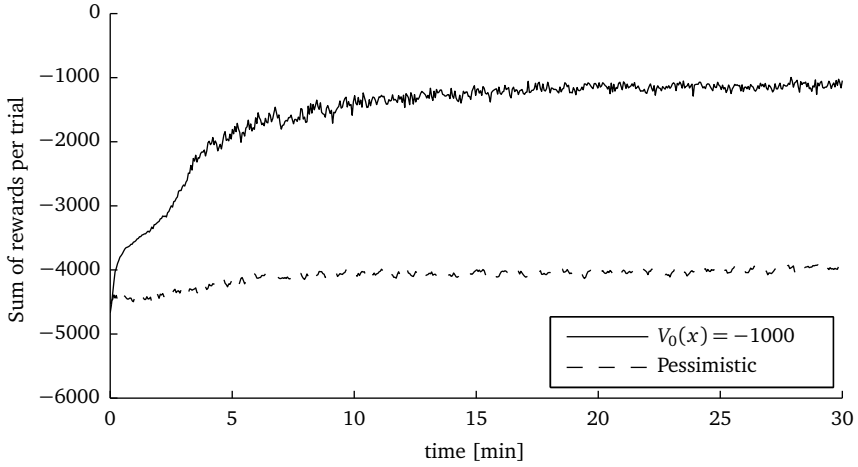


Figure 3.9 Results for the SAC algorithm with two initialisations for the value function: $V_0(x) = -1000$ for all x and $V_0(x) = \frac{1}{1-\gamma} \min_{x,u} r(x,u) \approx -4050$ (pessimistic).

sum of minimum rewards, i.e.

$$V_0(x) = \sum_{j=0}^{\infty} \gamma^j \min_{x \in X, u \in U} \rho(x,u) = \frac{1}{1-\gamma} \min_{x \in X, u \in U} \rho(x,u) \tag{3.17}$$

with ρ defined as in (3.16). Obviously, ρ has no minimum and tends towards $-\infty$ for $x \to \infty$, but assuming[4] that the angular velocity $|\dot{\theta}|$ will never exceed 8π rad/s and using the facts that $|\theta| \leq \pi$ rad and $|u| \leq 3$ V, it is still possible to calculate the worst possible immediate reward that can be received, by using $x = [\pi \; 8\pi]^\top$ and $u = 3$ in (3.16). This worst possible reward is then used as the minimum of r in (3.17). With $\gamma = 0.97$, the value function is then initialised

[4] This assumption is justified by the fact that typical trajectories do not exceed this velocity.

with $V_0(x) \approx -4050$. In Figure 3.9, a plot is also given for the case where the value function was initialised with $V_0(x) = -1000$ for all x. The drop in performance is gone and learning is still quick. However, estimating an initial value that will achieve this sort of behaviour is done by trial and error and is therefore very hard.

The final performance can be improved by increasing the number of partitions and number of tiles per partition in the tile coding, but this decreases the learning speed.

Radial Basis Functions

Figure 3.10 shows the learning curve for SAC, when combined with radial basis functions as a function approximator. The tuned parameters used for this particular simulation can be found in Table 3.1.

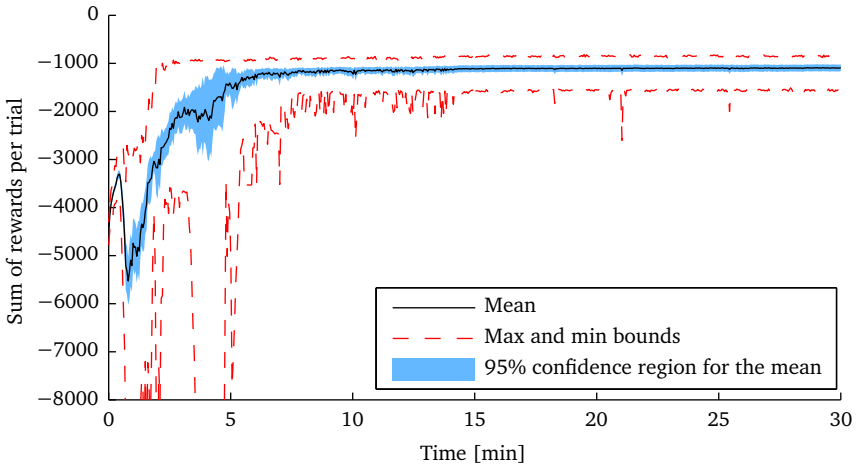


Figure 3.10 Results for the SAC algorithm using RBFs. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

The RBF implementation clearly also shows a dip in the learning performance at the start of learning. As with tile coding, the parametric nature of the approximator, in combination with an optimistic initial value for the parameters, is causing this behaviour.

Table 3.1 RBF parameters for the SAC, MLAC and RMAC methods.

	SAC	MLAC	RMAC
Actor / Reference model parameters			
learning rate $\alpha_{a/r}$	0.05	0.05	0.5
number of RBFs	$\begin{bmatrix} 20 \\ 10 \end{bmatrix}$	$\begin{bmatrix} 15 \\ 10 \end{bmatrix}$	$\begin{bmatrix} 10 \\ 10 \end{bmatrix}$
RBF intersection	0.9	0.9	0.9
Critic parameters			
learning rate α_c	0.4	0.1	0.15
number of RBFs	$\begin{bmatrix} 20 \\ 10 \end{bmatrix}$	$\begin{bmatrix} 15 \\ 10 \end{bmatrix}$	$\begin{bmatrix} 10 \\ 10 \end{bmatrix}$
RBF intersection	0.5	0.7	0.7
Process model parameters			
learning rate α_p	-	0.5	0.7
RBF intersection	-	0.9	0.7

Table 3.2 LLR parameters for the SAC, MLAC and RMAC methods.

	SAC	MLAC	RMAC
Actor / Reference model parameters			
learning rate $\alpha_{a/r}$	0.05	0.05	0.04
memory size $N^{\mathrm{a/r}}$	2000	2000	2000
nearest neighbours $K^{\mathrm{a/r}}$	20	25	15
Critic parameters			
learning rate α_c	0.3	0.3	0.2
memory size N^{C}	1000	2000	2000
nearest neighbours K^{C}	10	15	15
Process model parameters			
memory size N^{P}	100	100	100
nearest neighbours K^{P}	10	10	10

Local Linear Regression

When local linear regression is used as a function approximator, the learning curve for SAC looks like Figure 3.11. The parameters used for the LLR simulations in this chapter are given in Table 3.2.

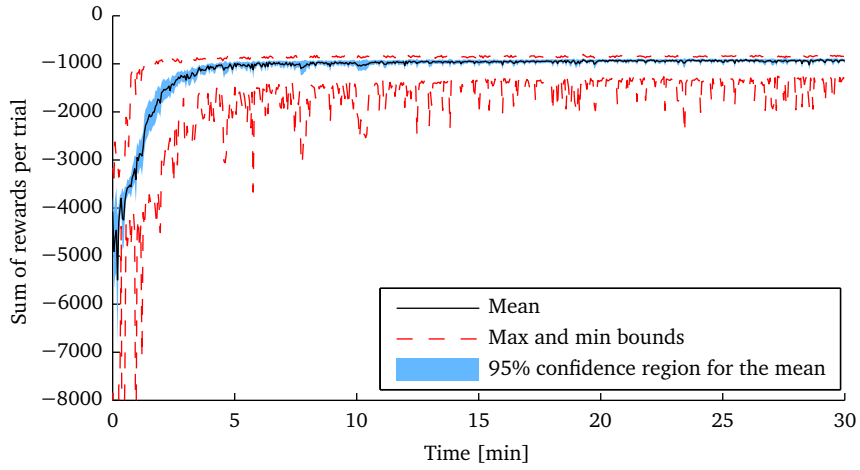


Figure 3.11 Results for the SAC algorithm using LLR. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

The most striking difference with the learning curves produced with tile coding and radial basis functions is obviously the absence of a dip in learning performance. Since LLR is non-parametric and starts with empty memories, no initial values of any kind have to be chosen, which means that also no wrong initial value can be chosen. Once the LLR memories start filling, the contents of the memories are always based on actual gathered experience and hence no sudden setback in learning performance should be expected.

Comparison of Different Function Approximators

To get an idea of the differences in performance when using various function approximators, the learning curves of Figures 3.6, 3.10 and 3.11 are plotted together in Figure 3.12. The absence of a learning dip with LLR clearly shows its benefits, as LLR comes out on top when looking at how fast

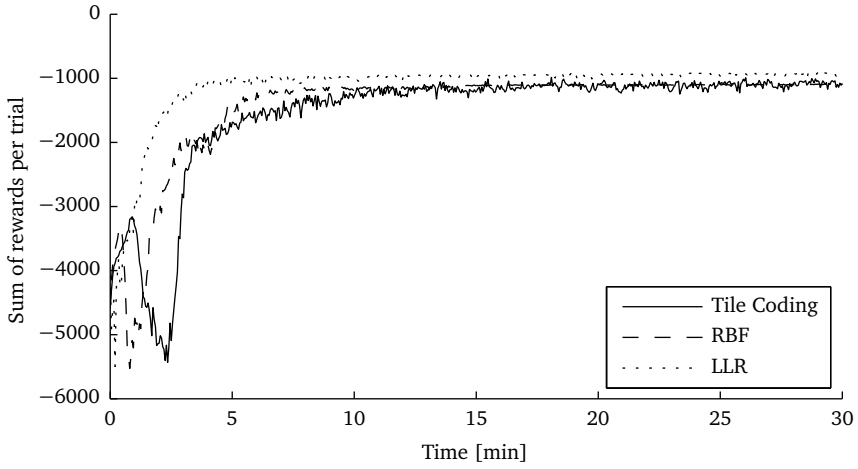


Figure 3.12 Results for the SAC algorithm using the three different function approximators.

the learning progresses. The RBF and tile coding approximators are only converging after just over 5 and 10 minutes, respectively, where LLR already reaches its maximum in about 3 minutes. Moreover, when looking at the final performance of the algorithm (which is the value of the learning curve at $t = 30\,\text{min}$), LLR also is outperforming the other two approximators.

3.6.2 Model Learning Actor-Critic

This section will describe the simulation results of MLAC on the pendulum swing-up task. Because the algorithm needs function approximators that are differentiable, tile coding can not be used and as a result only RBFs and LLR will be discussed.

Radial Basis Functions

Using the parameters from Table 3.1, the learning curve in Figure 3.13 was created. Again, the dip in learning performance at the start of the learning process is seen. This means that even though the model is helping the actor

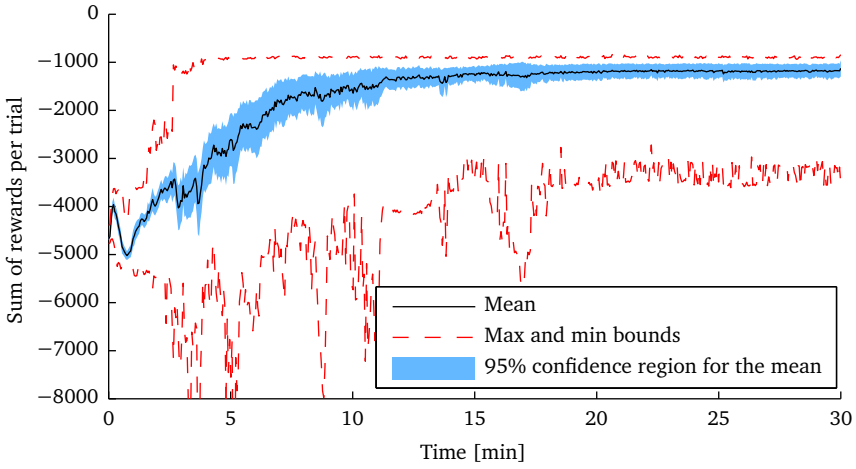


Figure 3.13 Results for the MLAC algorithm using RBFs. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

to update more efficiently towards an optimal policy, it can not overcome the difficulties caused by optimistic initialisation.

Local Linear Regression

The MLAC algorithm was applied using the parameter settings in Table 3.2, which produced the learning curve in Figure 3.14. As with SAC, the performance of MLAC is largely monotonically increasing when using LLR as the function approximator. Futhermore, the figure shows the effects of learning a model online. Where the combination of SAC and LLR converged in about 3 minutes, here the maximum is reached after about 2 min to 2.5 min. Figure 3.15 and Figure 3.16 show an example of how the actor and critic memories look after one full learning experiment with MLAC.

There is a lack of samples in the lower regions of Figure 3.15 and Figure 3.16, because none of the trials in this particular learning experiment generated a trajectory through that region.

The MLAC method converges fast and to a good solution of the swing-

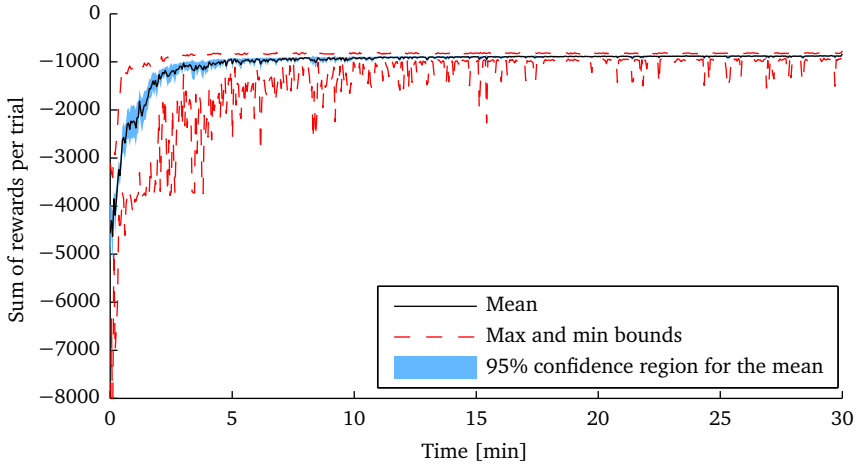


Figure 3.14 Results for the MLAC algorithm using LLR. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

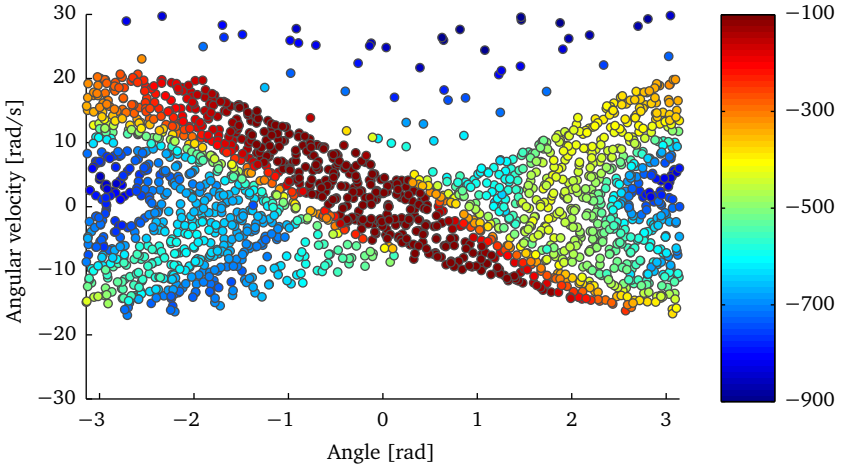


Figure 3.15 Final critic $V(x)$ for the MLAC algorithm after one learning experiment. Every point represents a sample $[x \mid V]$ in the critic memory M^{C}.

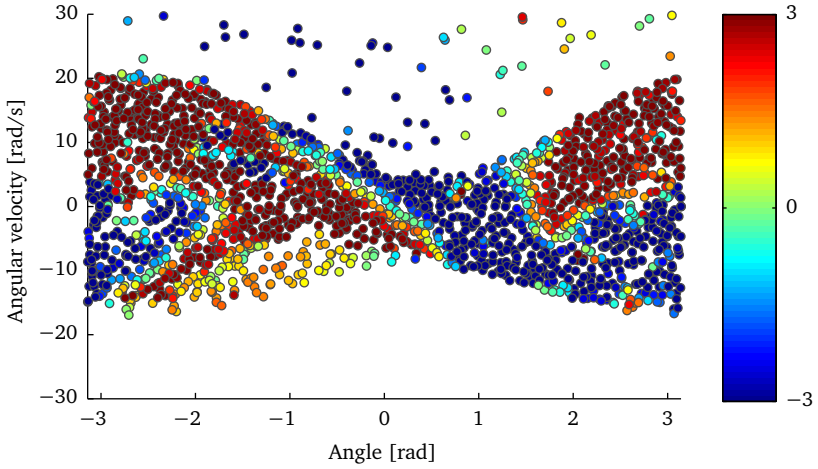


Figure 3.16 Final actor $\pi(x)$ for the MLAC algorithm after one learning experiment. Every point represents a sample $[x \mid u]$ in the actor memory M^{A}.

up task. The fast learning speed can be attributed to the characteristics of LLR. LLR gives a good quick estimate at the start of learning by inserting observations directly into the memory and also allows for broad generalisation over the states when the number of collected samples is still low. Finally, the update by Equation (3.9) (Equation (3.14) in the LLR case) is not stochastic in contrast to the update of SAC by Equation (3.1) which is typically based on the random exploration Δu. This means that MLAC is less dependent on the proper tuning of exploration parameters, such as the standard deviation used in the generation of random exploratory actions.

Comparison of Different Function Approximators

Figure 3.17 shows the two previously discussed learning curves in one plot. The figure is self-explanatory: LLR is far superior in both learning speed and performance when compared to the RBF implementation. Using LLR gives convergence in just over 3 minutes, whereas it takes over 10 minutes in the RBF case.

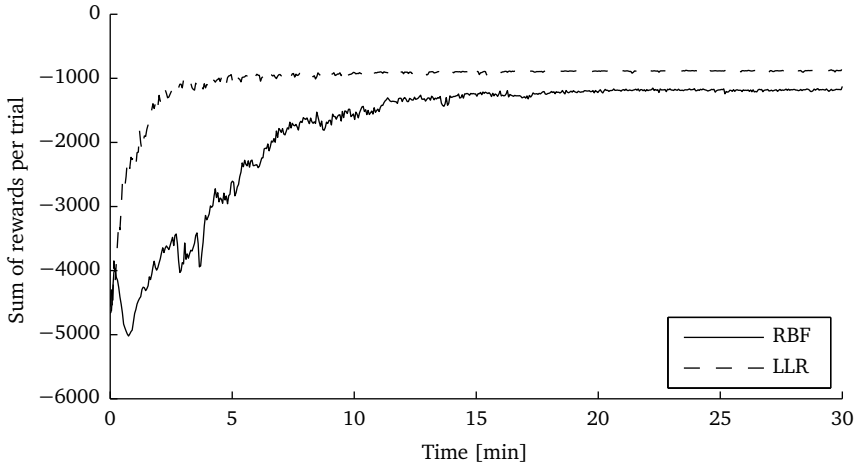


Figure 3.17 Results for the MLAC algorithm using the two different function approximators.

3.6.3 Reference Model Actor-Critic

Finally, the results of RMAC on the pendulum swing-up task is discussed. As with MLAC, only RBFs and LLR can be discussed as an implementation with tile coding does not exist.

Radial Basis Functions

The results of an implementation with RBFs is shown in Figure 3.18. A peculiar effect here is that the learning dip is still present, but not as explicit as with the other algorithms. A possible explanation for this is that the updates in RMAC are based on using the extreme values for the input, making the algorithm sweep the whole state space in a much shorter time than the other algorithms. This explanation is supported by Figure 3.20, which is still to be discussed, but already shows that the LLR memory samples are much more evenly spread across the whole state space than was the case with MLAC.

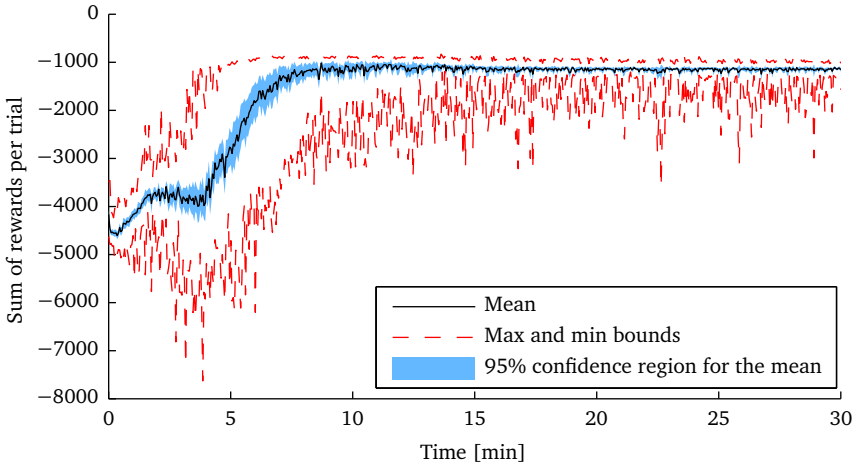


Figure 3.18 Results for the RMAC algorithm using RBFs. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

Local Linear Regression

As expected, the learning curve for RMAC in combination with LLR in Figure 3.19 shows monotonically increasing performance. Figure 3.20 shows the samples from the critic memory obtained after one representative learning experiment, whereas Figure 3.21 shows an example of memory samples for the reference model, by representing the mapping from states to their next respective desired states with arrows. Both figures show a large coverage of the state space[5], compared to Figures 3.15 and 3.16, confirming the statement made earlier that RMAC sweeps a large part of the state space.

The RMAC method combined with LLR converges very quickly and to a good solution for the pendulum problem. The main reason for the fast convergence is the fact that the method starts out by choosing the desired states $\hat{x}$ that resulted from the extremes of u. Desired states that result from the extremes of u also result in large values for u and makes the system explore

[5]The reference model in Figure 3.21 does not show all samples as this would make the figure less legible.

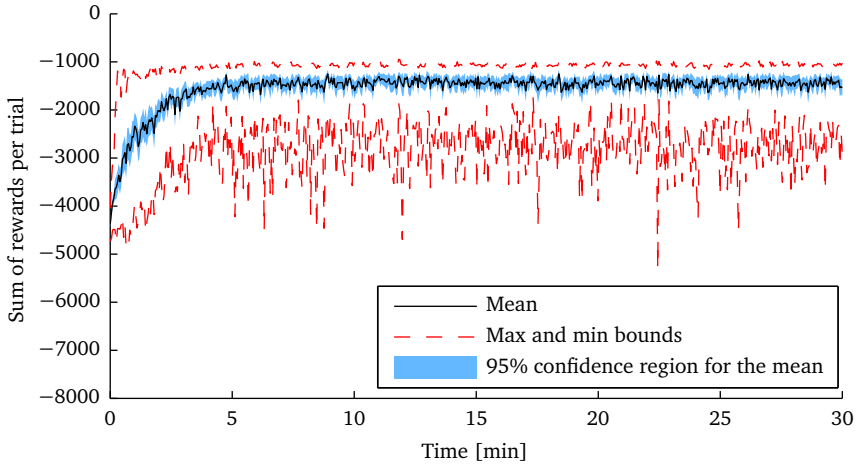


Figure 3.19 Results for the RMAC algorithm using LLR. The mean, max and min bounds and 95% confidence region are computed from 30 learning curves.

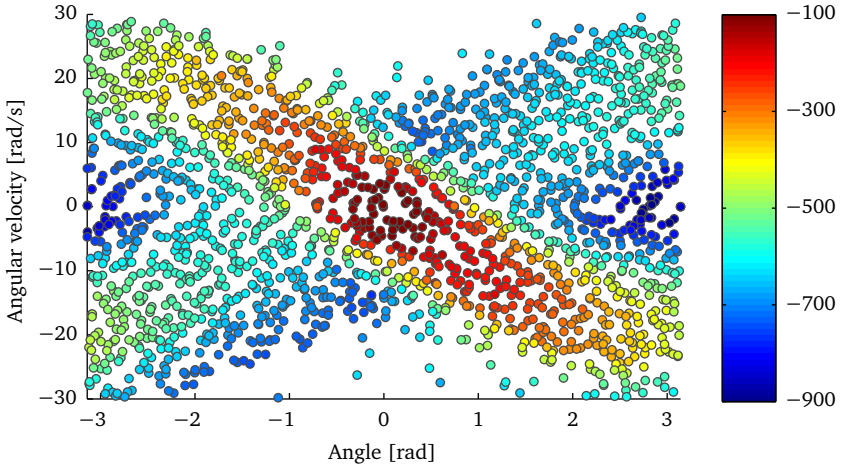


Figure 3.20 Final critic $V(x)$ for the RMAC algorithm after one learning experiment. Every point represents a sample $[x \mid V]$ in the critic memory M^{C}.

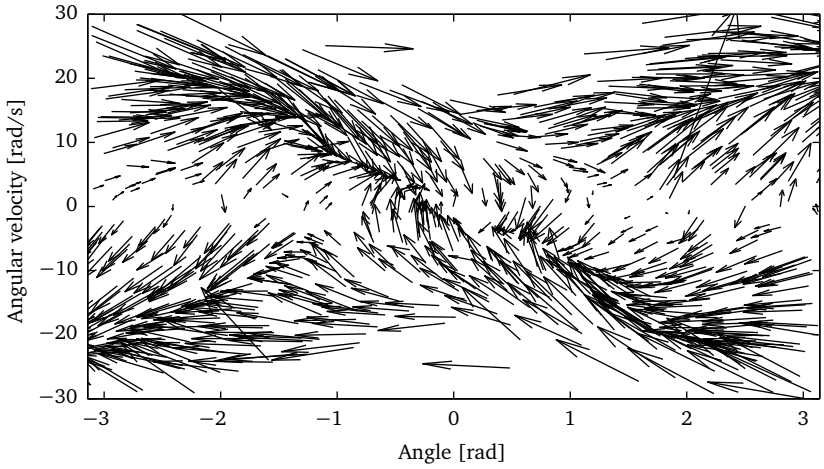


Figure 3.21 Final reference model $R(x)$ for the RMAC algorithm after one learning experiment. Every arrow represents a sample $[x \mid \hat{x}]$ in the reference model memory M^{R}. The arrow points to the desired next state $\hat{x}$.

a large part of the state space. This yields a fast initial estimate of the value function which is beneficial for the learning speed.

Comparison of Different Function Approximators

Although LLR is still outperforming the RBF implementation in terms of learning speed, the final policy obtained is not near as good, see Figure 3.22. This difference in final performance is likely to be caused by the updates to the reference model, using extreme values of the input. In the case of LLR, new samples based on these extreme values are *directly* put into the memory and the learning rate α_r only comes into play when existing samples are updated. In the case of RBFs, the reference model is trained using this new sample, which means that the learning rate α_r of the reference model already has a smoothing effect. This also explains why the RBF implementation is a tad slower than the LLR implementation: since the actions taken are not likely to be the extreme values anymore, the state space is not explored as quickly and hence it takes longer to build up enough experience across the whole space.

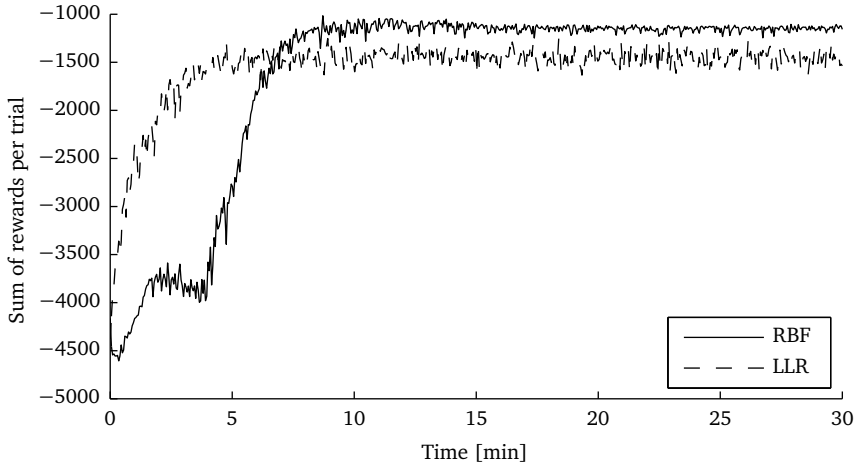
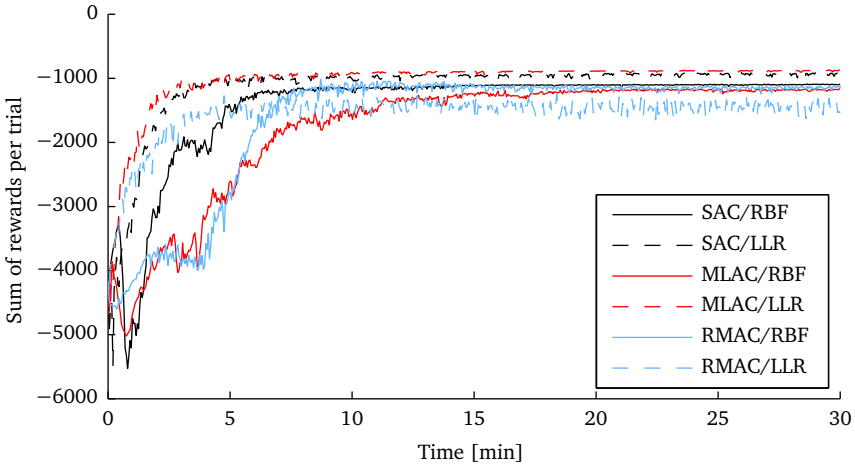


Figure 3.22 Results for the RMAC algorithm using the two different function approximators.

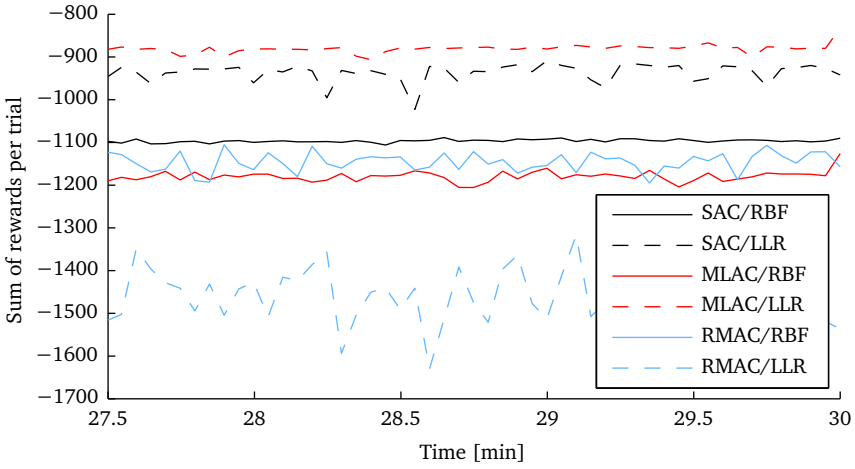
3.7 Discussion

Figure 3.23 shows all the learning curves from the previous sections together in one plot, with the exception of the tile coding implementation of SAC, as tile coding could not be used with MLAC and RMAC. The first plot shows the complete learning experiments, the second plot zooms in on the performance in the last 50 episodes.

It is clear from this picture that all methods learn the quickest when they use LLR as the function approximator. Furthermore, the MLAC and RMAC methods are performing better than SAC in terms of learning speed. When using RBFs, the MLAC and RMAC method do not perform better than SAC, which implies that they need a better function approximator to become truly powerful. In addition, the LLR implementations of SAC and MLAC also obtain a much better performance at the end of the learning experiment than their RBF counterparts. The RMAC/LLR combination shows a steep learning curve at the start, but does not reach the best solution. Interestingly, the RMAC method is the only method here that performs better when using RBFs as the function approximator.



(a) The complete learning curves of all algorithms and function approximators.



(b) The last 2.5 minutes of the learning curves.

Figure 3.23 Comparison of different algorithms and function approximators.

From the results in the previous section, local linear regression seems to be the approximator of choice when the learning should both be quick and deliver a good performance at the end of a learning experiment.

The MLAC and RMAC methods only really show their power when used in combination with local linear regression. It is the combination of both LLR and these new methods that will provide quick, stable and good learning. However, no tests were done using a mix of function approximators, e.g. RBFs for the actor and critic and LLR for the process model, which might yield even more powerful results. Another option is to try and combine the quick learning of RMAC/LLR with the good performance of MLAC/LLR, by starting with RMAC and switching to MLAC when the performance does not significantly increase anymore.

A crucial condition for the model learning algorithms to work is that the input saturation of a system should be dealt with when learning a process model. Simply ignoring this will not produce a well performing policy. This can be overcome by setting the process model's gradient with respect to the input to zero at the saturation bounds or by making sure that the actor can not produce output signals beyond the saturation bounds.

A persisting problem in dynamic programming and reinforcement learning is the proper tuning of the learning algorithms and, if applicable, function approximators. The quality of an RL algorithm should therefore not only be measured by looking at the performance of the policy it produces, but also by checking its robustness, i.e. how well it keeps performing when deviating from the set of optimally tuned parameters. This sensitivity analysis for the algorithms presented here is left for future work.

Although not exploited in this thesis, one benefit of the memory-based function approximators is that they can easily be initialised with samples obtained from other experiments. This makes it easy to incorporate prior knowledge on the process dynamics, a near optimal control policy or near optimal behaviour. For example, (partial) knowledge of the process could be incorporated in the process model's memory and the reference model memory in RMAC could be initialised with samples of desired closed-loop behaviour. This can be beneficial if the desired behaviour of the system is known but the control policy is yet unknown (which is often the case when supplying prior knowledge by imitation (Peters and Schaal, 2008a)). In both algorithms, the process model can be initialised with input/state/output samples of the

open loop system. This has the benefit that it is usually easy to conduct experiments that will generate these samples and that it is unnecessary to derive an analytical model of the system, as the samples are used to calculate locally linear models that are accurate enough in their neighbourhood.

LLR seems very promising for use in fast learning algorithms, but a few issues prevent it from being used to its full potential. The first issue is how to choose the correct input weighting (including the unit scaling of the inputs), which has a large influence on selecting the most relevant samples for regression. A second issue that must be investigated more closely is memory management: different ways of scoring samples in terms of age and redundancy and thus deciding when to remove certain samples from the memory will also influence the accuracy of the estimates generated by LLR.

The experiments now only compare the performance of MLAC and RMAC with a standard actor-critic algorithm. It should also be interesting to see how these methods compare to direct policy search methods and natural gradient based techniques. Finally, the approximation of the reachable subset $\mathcal{X}_R$ may not be sufficiently accurate for more complex control tasks. A more reliable calculation of reachable states is the main improvement that could be made to RMAC.

Chapter

4

Solutions to Finite Horizon Cost Problems Using Actor-Critic Reinforcement Learning

Up to this chapter, only tasks that are modeled as a Markov decision process with an infinite horizon cost function have been dealt with. In practice, however, it is sometimes desired to have a solution for the case where the cost function is defined over a finite horizon, which means that the optimal control problem will be time-varying and thus harder to solve. In this chapter, the algorithms presented in the Chapter 3 are adapted from the infinite horizon setting to the finite horizon setting and applies them to learning a task on a nonlinear system. Since this is still in the actor-critic reinforcement learning setting, no assumptions or knowledge about the system dynamics are needed. Simulations on the pendulum swing-up problem are carried out, showing that the presented actor-critic algorithms are capable of solving the difficult problem of time-varying optimal control.

4.1 Introduction

Most of the literature on reinforcement learning, including the work on actor-critic algorithms, discusses their use in Markov decision processes (MDPs) in an infinite horizon cost setting. On one hand, this is a reasonable choice when considering a controller for regulation or tracking that, after training or learning, is put into place and stays in operation for an indefinite amount of time, to stabilise a system for example. On the other hand, one can think of

control tasks that have to be performed within a given time frame, such as a pick and place task for a robotic manipulator or network flow and congestion problems as discussed by Bhatnagar and Abdulla (2008). For these problems, a description using MDPs with a finite horizon cost function is more suitable.

The main difference between infinite and finite horizon control is the terminal cost[1] constraint that is put in place in finite horizon control. As a result of this constraint, the control law used near the end of the horizon will, for example, favour actions that will prevent a large penalty of the terminal cost constraint, whereas at the start of the process, other actions might be optimal, since the threat of receiving a large penalty is not as imminent. In other words, optimality in finite horizon control requires a time-varying control law.

From an RL perspective, a time-varying control law poses the challenge of having to deal with yet another dimension while learning, being time, obviously contributing to the well-known curse of dimensionality (Powell, 2010) and hence making the learning tougher. Nevertheless, learning control might be a weapon of choice for finding the optimal control for (nonlinear) systems over a finite time horizon, as explicitly solving the Hamilton-Jacobi-Bellman equations that follow from this setting is difficult or even impossible (Cheng et al., 2007).

Existing work on learning optimal control for nonlinear systems in the finite horizon (or fixed-final-time) setting is mainly carried out in the approximate dynamic programming community. Quite often, stringent assumptions are posed on the system to be controlled, such as dynamics or transition probabilities that have to be fully known (Bertsekas and Tsitsiklis, 1996; Cheng et al., 2007); alternatively, a strategy that requires offline training of a model network is suggested (Park et al., 1996; Wang et al., 2012). Q-learning and actor-critic algorithms for use in finite horizon MDPs were developed before by Gosavi (2010) and Bhatnagar and Abdulla (2008), respectively, but these work for a finite state space and a compact or finite action space. Moreover, even though an explicit model of the system was assumed to be unknown, Bhatnagar and Abdulla (2008) assumed that transitions could be simulated somehow. Another approach to solving finite horizon Markov decision processes is to find a time-invariant policy (Furmston and Barber, 2011), hence resulting in a suboptimal policy.

[1] Terminal *cost* is used here, in keeping with existing literature. In the terminology of reinforcement learning, however, one could also see this as a terminal *reward*.

In this chapter, the actor-critic algorithms described in Chapter 3 are adapted to work online in a finite horizon setting, without requiring any kind of knowledge of the (nonlinear) system, nor the ability to simulate transitions throughout the whole state and action space. The algorithms produce time-varying policies. Furthermore, both state and action space are assumed to be continuous and hence the time-varying policies should allow for continuous actions.

The chapter continues with an introduction to the concept of MDP's in a finite horizon cost setting in Section 4.2. Section 4.3 discusses how actor-critic RL algorithms may be used to solve a finite horizon cost MDP and adapts the previously introduced actor-critic algorithms to work in the finite horizon setting. Simulations with the newly derived algorithms on a pendulum swing-up task will be discussed in Section 4.4 and the chapter ends with conclusions and recommendations for future work in Section 4.5.

4.2 Markov Decision Processes for the Finite Horizon Cost Setting

This section introduces the concepts of finite horizon Markov Decision Processes, partly based on the definitions given by Bäuerle and Rieder (2011), but combined with notations used throughout this thesis and ideas by Bhatnagar and Abdulla (2008).

A finite horizon MDP is a tuple $\langle X, U, f, \rho, \tau_N \rangle$, where X denotes the state space, U the action space, $f : X \times U \to X$ the (deterministic) state transition function, $\rho : X \times U \times X \to \mathbb{R}$ the reward function for system times $k = 0, 1, \ldots, N-1$ and $\tau_N : X \to \mathbb{R}$ the terminal reward of the system at time $k = N$.

The process to be controlled is described by the state transition function[2] f, which returns the state x_{k+1} that the process reaches from state x_k after applying action u_k. As in the infinite horizon case, after each transition to a state x_{k+1}, while the finite time horizon N has not been reached, the controller

[2]For simplicity, it is assumed here that the transition function f is still time-invariant. Hence, the use of a terminal cost function is the sole cause of a time-varying policy.

receives an immediate reward

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

which depends on the current state, the action taken and the next state. Note that for deterministic systems, the next state is fully determined by the current state and the action taken, making it possible to shorten this definition to

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

and this is the definition that will be used for the remainder of this chapter. For the finite horizon case, when the system reaches the final time instant $k = N$, a terminal reward $\tau_N(x_N)$ is received. The (terminal) reward functions ρ and τ_N are both assumed to be bounded.

The action u_k taken in a state x_k is determined by the policy π, i.e. the actor, that is time-dependent because of the finite horizon cost setting. This time-varying policy is denoted by $\pi_k : X \to U$, where the subscript k denotes the time instant at which the policy π_k is active.

The goal of the reinforcement learning agent is to find the N-stage policy $\pi = (\pi_0, \pi_1, \ldots, \pi_{N-1})$ which maximises the expected value of the cost function J, which is equal to expected value of the sum of rewards received when starting from an initial state $x_0 \in X$ drawn from an initial state distribution $x_0 \sim d_0(\cdot)$;

$$J(\pi) = \mathrm{E}\left\{ \tau_N(x_N) + \sum_{k=0}^{N-1} \rho(x_k, \pi_k(x_k)) \middle| d_0 \right\}. \tag{4.1}$$

This sum is also called the return. Even though a deterministic process and policy are assumed, the expectation is introduced because of the starting state distribution d_0. For the simulations in this chapter only one starting state is chosen, making it possible to drop the expected value.

For policy evaluation, the critic will have to estimate the cost-to-go function for a given N-stage policy π for any state x at some time instant k. The resulting estimate is the *value function*. The value function for a finite horizon problem is time-varying as well, as at every time step it carries the value for a state, assuming the use of $N - k$ remaining different policies. The time-varying value function is given at each time step k by (see e.g. Bertsekas and Tsitsiklis, 1996))

$$V_k^{\pi}(x) = \tau_N(x_N) + \sum_{j=k}^{N-1} \rho(x_j, \pi_j(x_j)), \tag{4.2}$$

where $x_k = x$. Note that the value function at time $k = N$ for any N-stage policy only depends on the state, as no further actions from a policy are required. Following Equation (4.2), the value function at the final time step is known to be

$$V_N^\pi(x) = \tau_N(x). \tag{4.3}$$

Expanding Equation (4.2), the Bellman equation for the finite horizon case is

$$V_k^\pi(x) = \rho(x, \pi_k(x)) + V_{k+1}^\pi(x'), \tag{4.4}$$

with $x' = f(x, \pi_k(x))$.

Optimality for the state value function V_k^π is governed by the Bellman *optimality* equation. Denoting the optimal state value function with $V_k^*(x)$, the Bellman optimality equation is

$$V_k^*(x) = \max_u \left\{ \rho(x,u) + V_{k+1}^*(x') \right\},$$

with $x' = f(x,u)$.

4.3 Actor-Critic RL for Finite Horizon MDPs

To find an optimal control strategy for a finite horizon MDP, adapted versions of the actor-critic algorithms used in the previous chapter must be used. This section discusses how this adaptation may be achieved. Before deriving the finite horizon version of these algorithms, first some ideas that will apply to all algorithms will be discussed.

4.3.1 Parameterising a Time-Varying Actor and Critic

For the same reasons as before in the infinite horizon case, the time-varying actor and critic will also be parameterised functions. The critic's parameterisation will be denoted with the parameter vector θ and basis functions $\phi(\cdot)$ and the the actor's parameterisation will be denoted with the parameter vector ϑ and basis functions $\psi(\cdot)$.

As discussed in Section 4.2, both the actor and the critic need to be time-varying if they want to be able to represent an optimal control strategy for a finite horizon MDP. The question then arises how the time-varying nature

of these functions should be parameterised. One can opt for the actor-critic representation[3] where the time dependence is put into the weights

$$V_k(x) = {\theta_k}^\top \phi(x) \qquad \pi_k(x) = {\vartheta_k}^\top \psi(x) \tag{4.5}$$

which is done, for example, by Cheng et al. (2007); Wang et al. (2012) or a respresentation where the time-variance is put into the basis functions

$$V_k(x) = \theta^\top \phi(x,k) \qquad \pi_k(x) = \vartheta^\top \psi(x,k). \tag{4.6}$$

which is done by Bhatnagar and Abdulla (2008). It should be noted that regardless of the chosen representation, the weights θ and ϑ are always time-varying during learning because of the update laws applied to them. In other words, the representations presented here depict the situation when all weights have converged.

The easiest way to extend existing infinite horizon reinforcement learning algorithms to finite horizon problems is by using the representation in Equation (4.6), as it really only requires the expansion of the state dimension with one extra dimension for time. The necessary changes in the algorithms themselves (temporal difference calculations, eligibility trace updates, etc.) will then follow naturally. Having time as part of the state also adds the advantage of automatic "time generalisation" to the learning: even though actor and critic are time-varying, it is not unreasonable to assume that these functions vary smoothly with time. By choosing this particular parameterisation, an update at one particular time instance will also update neighboring times, which should speed up learning.

Although the extension using Equation (4.6) is theoretically straightforward, the time dimension that is added to the state space is superfluous in some ways. Firstly, the input space of the basis functions is continuous, whereas time is usually a discretised variable in learning problems. This means the actor and critic are approximated over a continuous time axis and this might affect the quality of the discrete time control policy obtained. Secondly, the discrete-time dynamics of the time variable are known (a discrete-time integrator), posing the question if it really should be rendered unknown by making it part of the state. Both points show that simply considering time to be an element of state might actually overcomplicate the learning of the value function. In practice,

[3]Note that the superscript π is no longer present in the value function, as it may no longer represent the exact value function for a specific policy per se.

the learning speed achieved might therefore be negatively affected by using this approach, although the experiments at the end of this chapter do not show any significant slowdown compared to the experiments in the previous chapter.

Using the parameterisation of Equation (4.5) also has its advantages and drawbacks. One advantage is that the time axis is held discrete, which eliminates any approximation error over the time variable and may thus result in policies at every time step k that are optimal, as every time step has its own policy parameter which is independent of the policy parameters at other time steps. Furthermore, the complexity of the basis functions is reduced, since they only have the state as their input and in this sense should not be more complicated to choose than in the infinite time horizon case. An obvious disadvantage is that the number of parameters can grow very large as the time horizon N increases. Finally, generalisation across the time axis is lost if the parameters of different time steps are updated independently of each other, although this may be addressed, for example, by a clever updating technique similar to eligibility traces.

Since the step from an infinite horizon implementation to a finite horizon implementation is apparently smaller with the parameterisation in Equation (4.6), this chapter uses that particular parameterisation. The use of Equation (4.5) is shortly explored at the end of this chapter.

4.3.2 Standard Actor-Critic

Based on the implementation of the infinite horizon standard actor-critic (SAC) algorithm, this sections derives its finite horizon version by observing the changes needed specifically for this algorithm, when the parameterisation in Equation (4.6) is used.

Having extended the state with a time element, the implementation of the algorithm itself still needs to be changed to incorporate the effect of the terminal reward $\tau_N(x)$ in the updates. Equations (4.3) and (4.4) are the key to implement the needed change. The heuristic ideas behind the update rules are the same as in the infinite horizon case, but it is the temporal difference error that is being used for the update that requires attention. Since Equation (4.4) now contains time-varying value functions, this time-variance has to be reflected in the definition of the temporal difference. The new definition of the temporal difference error is obtained by subtracting the left-

hand side of Equation (4.4) from its right-hand side, giving

$$\delta_k = \rho(x_k, \pi_k(x_k)) + V_{k+1}^{\pi}(x_{k+1}) - V_k^{\pi}(x_k)$$

During learning, exploration needs to be added to the inputs generated by the current policy, such that the learning agent is able to find new, possibly better, policies. With the exploration at time k defined as Δu_k, and substitution of the parameterisation of the value function, the temporal difference used is

$$\delta_k \approx r_{k+1} + \theta^{\top}\left[\phi(x_{k+1}, k+1) - \phi(x_k, k)\right]$$

with $r_{k+1} = \rho(x_k, u_k)$ and $u_k = \pi_k(x_k) + \Delta u_k$. This definition of the temporal difference error will hold for time steps $k = 0, 1, \ldots, N-2$, but not for $k = N-1$, as this would include the approximation term for $V_N^{\pi}(x_N)$, for which the prior knowledge from Equation (4.3) should be plugged in, yielding

$$\delta_{N-1} = r_N + \tau_N(x_N) - \theta^{\top}\phi(x_{N-1}, N-1).$$

The update law for the critic itself remains the same. The resulting finite horizon standard actor-critic (SAC-FH) algorithm is given in Algorithm 4.[4]

4.3.3 Model Learning Actor-Critic

The finite horizon version of the Model Learning Actor-Critic algorithm (MLAC-FH) not only needs a different calculation of the temporal difference error, but the policy gradient used in MLAC also needs adjusting, as it depends on the value function which is now time-varying, implying that the policy gradient itself is now also time-varying. The exact change needed will be discussed next.

As a reminder, the idea behind the Model Learning Actor-Critic algorithm (MLAC) is to increase the learning speed by using each transition sample to learn a model of the process in an online fashion. The approximate process model $x' = \widehat{f}_{\zeta}(x, u)$ is parameterised by the parameter $\zeta \in \mathbb{R}^{r \cdot n}$, where r is the number of basis functions per element of process model output used for the approximation and n is the state dimension. The process model is learned online by using a simple gradient descent method on the modeling

[4] The symbol $\leftarrow$ is used here to indicate a *replacement* of an old parameter value with a new one. This is to prevent notational confusion with the time-varying parameter approach discussed earlier.

Algorithm 4 SAC-FH

Input: $\lambda, \alpha_a, \alpha_c$

Initialise function approximators
loop
  Reset eligibility: $z = 0$
  Set x_0
  for $k = 0$ **to** $N-1$ **do**
    Choose exploration $\Delta u_k \sim \mathcal{N}(0, \sigma^2)$
    Apply $u_k = \text{sat}\left\{\vartheta^\top \psi(x_k) + \Delta u_k\right\}$
    Measure x_{k+1}
    $r_{k+1} = \rho(x_k, u_k)$
    *// **Update actor and critic***
    if $k = N-1$ **then**
      $\delta_k = r_{k+1} + \tau_N(x_{k+1}) - \theta^\top \phi(x_k, k)$
    else
      $\delta_k = r_{k+1} + \theta^\top \left[\phi(x_{k+1}, k+1) - \phi(x_k, k)\right]$
    end if
    $\vartheta \leftarrow \vartheta + \alpha_a \delta_k \Delta u_k \psi(x_k, k)$
    $z \leftarrow \lambda z + \phi(x_k, k)$
    $\theta \leftarrow \theta + \alpha_c \delta_k z$
  end for
end loop

error between the process model output $\widehat{f}_\zeta(x_k, u_k)$ and the actual system output x_{k+1}.

Substituting the process model into Equation (4.4), the Bellman equation becomes

$$V_\theta(x_k, k) = \rho(x_k, \pi_\vartheta(x_k, k)) + V_\theta(\widehat{f}_\zeta(x_k, \pi_\vartheta(x_k, k), k+1). \tag{4.7}$$

This equation shows how the value function for a specific state x and time step k can be influenced by manipulating the policy parameter ϑ.

Given that the cost-to-go function at any time k is modeled by the time-dependent value function $V(x, k)$, the gradient of the cost-to-go function, i.e. the policy gradient, can be found by taking the derivative of Equation (4.7)

with respect to the policy parameter ϑ, resulting in[5]

$$\nabla_\vartheta V(x_k,k) \approx \nabla_u \rho(x_k,\tilde{u}_k)^\top \nabla_\vartheta \pi_\vartheta(x_k,k) + \nabla_x V(\tilde{x}_{k+1},k+1)^\top \nabla_u \widehat{f}_\zeta(x_k,\tilde{u}_k) \nabla_\vartheta \pi_\vartheta(x_k,k),$$

with

$$\begin{aligned} \tilde{x}_{k+1} &= \widehat{f}_\zeta(x_k,\tilde{u}_k) \\ \tilde{u}_k &= \pi_\vartheta(x_k,k) \end{aligned}$$

Putting the parameterisation of the policy into place, this yields the policy gradient

$$\nabla_\vartheta V(x_k,k) \approx \left\{ \nabla_u \rho(x_k,\tilde{u}_k)^\top + \nabla_x V(\tilde{x}_{k+1},k+1)^\top \nabla_u \widehat{f}_\zeta(x_k,\tilde{u}_k) \right\} \psi(x_k,k)$$

Using this policy gradient, the policy parameter ϑ can be moved in a direction that will produce a better value function for a given state and time.

With the temporal difference update on the critic being the same as in the SAC-FH case, the algorithm for MLAC-FH is as shown in Algorithm 5.

4.3.4 Reference Model Actor-Critic

To make Reference Model Actor-Critic suitable for finite horizon cost functions, it is only necessary to add a time variable to the domains of the critic and the reference model to make them time-varying functions. This means that the time-dependent critic is defined the same way as with SAC-FH and MLAC-FH. For the reference model, the input is now the current state and time, since the desired next state can (and will likely) be different depending on the current time instant. The output is still the state of the system, but *excluding time*. Making time part of the reference model output would imply a redundant modeling of information, since a system at time k can only move to time instant $k+1$ after selecting one input. The rest of the algorithm, e.g. the way in which the set of reachable states is defined and how the models are updated, remains unchanged.

With these slight modifications in place, the pseudocode for the RMAC-FH algorithm is given in Algorithm 6.

[5]The derivation of this gradient is similar to the derivation presented in Section 3.3.2 and hence is not repeated here.

Algorithm 5 MLAC-FH

Input: $\lambda, \alpha_a, \alpha_c, \alpha_p$

Initialise function approximators
loop
  Reset eligibility: $z = 0$
  Set x_0
  for $k = 0$ **to** $N - 1$ **do**
    Choose exploration $\Delta u_k \sim \mathcal{N}(0, \sigma^2)$
    Apply $u_k = \text{sat}\left\{\vartheta^\top \psi(x_k) + \Delta u_k\right\}$
    Measure x_{k+1}
    $r_{k+1} = \rho(x_k, u_k)$
    // Update process model
    $\zeta \leftarrow \zeta + \alpha_p (x_{k+1} - \widehat{f}_\zeta(x_k, u_k)) \nabla_\zeta \widehat{f}_\zeta(x_k, u_k)$
    // Update actor
    $\tilde{u}_k = \text{sat}\left\{\vartheta^\top \psi(x_k)\right\}$
    $\tilde{x}_k = \widehat{f}_\zeta(x_k, \tilde{u}_k)$
    $\vartheta \leftarrow \vartheta + \alpha_a \left\{\nabla_u \rho(x_k, \tilde{u}_k)^\top + \nabla_x V(\tilde{x}_{k+1}, k+1)^\top \nabla_u \widehat{f}_\zeta(x_k, \tilde{u}_k)\right\} \psi(x_k, k)$
    // Update critic
    if $k = N - 1$ **then**
      $\delta_k = r_{k+1} + \tau_N(x_{k+1}) - \theta^\top \phi(x_k, k)$
    else
      $\delta_k = r_{k+1} + \theta^\top \left[\phi(x_{k+1}, k+1) - \phi(x_k, k)\right]$
    end if
    $z \leftarrow \lambda z + \phi(x_k, k)$
    $\theta \leftarrow \theta + \alpha_c \delta_k z$
  end for
end loop

4.4 Simulation Results

The finite horizon actor-critic algorithms of Section 4.3 have been applied to the problem of swinging up an inverted pendulum, as described in Appendix A.1.

The task is the same as before: starting from the pointing down position,

Algorithm 6 RMAC-FH

Input: $\lambda, \alpha_a, \alpha_c, \alpha_p$

Initialise function approximators
loop
  Reset eligibility: $z = 0$
  Set x_0
  for $k = 0$ **to** $N-1$ **do**
    Choose exploration $\Delta u_k \sim \mathcal{N}(0, \sigma^2)$
    Apply $u_k \leftarrow \text{sat}\{u_k + \Delta u_k\}$
    Measure x_{k+1}
    $r_{k+1} = \rho(x_k, u_k)$
    $\widehat{x}_{k+2} \leftarrow R_{\eta_k}(x_{k+1}, k+1)$
    $u_{k+1} \leftarrow \widehat{f}^{-1}_{\zeta_k}(x_{k+1}, \widehat{x}_{k+2})$
    // Update process model
    $\zeta \leftarrow \zeta + \alpha_p(x_{k+1} - \widehat{f}_\zeta(x_k, u_k))\nabla_\zeta \widehat{f}_\zeta(x_k, u_k)$
    // Update critic
    if $k = N-1$ **then**
      $\delta_k = r_{k+1} + \tau_N(x_{k+1}) - \theta^\top \phi(x_k, k)$
    else
      $\delta_k = r_{k+1} + \theta^\top \left[\phi(x_{k+1}, k+1) - \phi(x_k, k)\right]$
    end if
    $z \leftarrow \lambda z + \phi(x_k, k)$
    $\theta \leftarrow \theta + \alpha_c \delta_k z$
    // Update reference model
    Select best reachable state x_r from (x_k, k)
    $\eta_{k+1} \leftarrow \eta_k + \alpha_r(x_r - \widehat{x}_{k+1})\nabla_\eta R_{\eta_k}(x_k)$
  end for
end loop

the pendulum must be swung up to the upright position. Again, only limited actuation is allowed, such that it is impossible to complete the task by directly moving from the pointing-down position to the upright position in one direction.

The function approximator used for the actor, critic and process model is a network of Gaussian radial basis functions (RBFs), with a number of 9 RBFs

per dimension, centered evenly throughout the state space, where it is assumed that the position $\varphi \in [-\pi, \pi]$, the angular velocity $\dot{\varphi} \in [-8\pi, 8\pi]$ and it is further known that $k \in \{0, 1, \ldots, 99\}$, since the sampling time for the 3-second horizon is chosen to be 0.03 s, giving 100 time steps in total. The above implies that a total of 729 RBFs are used in the time-varying actor and critic as they both have three dimensions (angle, angular velocity and time) and 81 RBFs in the process model, which only has two dimensions (angle and angular velocity) as the process itself is time-invariant.

The reward functions used are

$$\rho(x, u) = -x^\top Q x - u^2 \qquad \tau_N(x) = -x^\top Q x$$

with

$$x = \begin{bmatrix} \varphi \\ \dot{\varphi} \\ k \end{bmatrix} \qquad Q = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Note that the last row and column of Q are set to zero, meaning that no time-based penalty is given.

For both experiments, a full learning experiment runs for 30 minutes of simulated time. The experiment starts with all approximators (actor, critic and process model) initialised at zero. One experiment consists of 600 episodes, each with a duration of 3 seconds. At the start of an episode, the state of the system is reset to the pointing down position $x_0 = [\pi\ 0]^\top$ and the eligibility trace is erased, but the actor, critic and process model are kept intact, such that the next episodes can benefit from the learning experiences in previous episodes. For each episode, the return defined in Equation (4.1) can be calculated, which means that for each experiment 600 returns are calculated. Plotting these returns versus the episodes results in a *learning curve*, showing how the return is improving with each episode. The next sections will use these learning curves as a measure of learning speed. Note that the learning curves include the stochastic effects caused by exploration during the episodes.

4.4.1 Finite Horizon Standard Actor-Critic

Figure 4.1a shows a learning curve for the SAC-FH algorithm. To account for stochastic effects introduced by exploration, the graph is an average of

20 different learning curves, all generated using the same set of learning parameters.

The learning parameters $\alpha_a = 0.05$, $\alpha_c = 0.2$ and the widths of the Gaussian RBF grid (0.55) were tuned for best end performance, by iterating over a grid of parameters and choosing the parameter set that generated the highest average return over the 20 experiments in the last 100 episodes. The number of RBFs used per dimension was held fixed to 9.

From this figure, it can be seen that after learning for about a minute, a dip occurs in the return. Similar to the results in the infinite horizon case, this is likely to be a result of the optimistic initialisation ($\theta = 0$) of the value function. Once a more reasonable value function has been learned, the return gradually increases and stabilises after roughly 15 minutes.

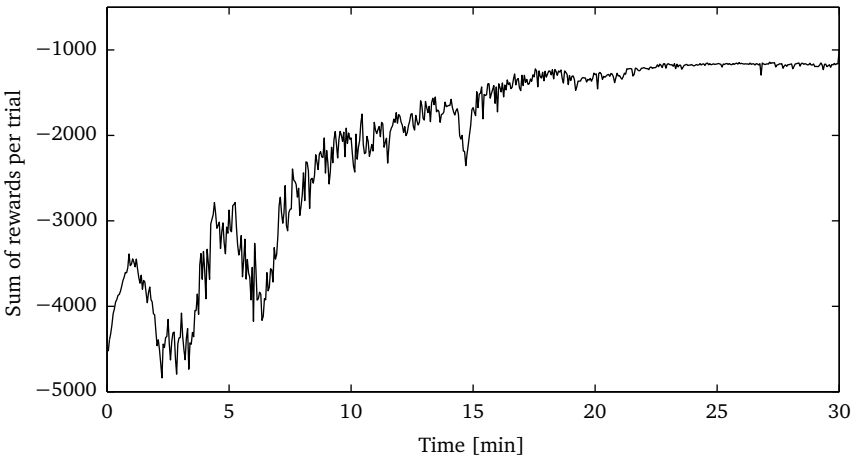
Figure 4.1b shows the typical time response of the system with a policy that was learned using SAC-FH. For this figure, the final policy (after 600 episodes) was used with the exploration switched off.

The policy generated by SAC-FH clearly shows how it forces the pendulum to pick up momentum by applying the extrema of possible inputs at the start to make the pendulum swing back and forth. Around 0.5 s, the input switches sign again to brake the pendulum and keep it in the upright position.

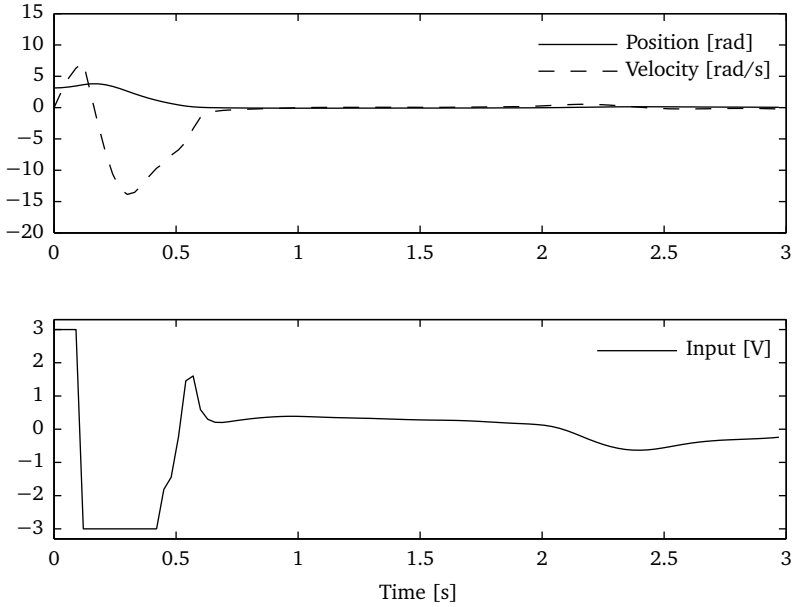
4.4.2 Finite Horizon Model Learning Actor-Critic

After tuning, the learning parameters for the MLAC-FH algorithm were set to $\alpha_a = 0.15$, $\alpha_c = 0.4$, $\alpha_p = 0.8$. The number of RBFs in each dimension were again held fixed at 9, but the widths of the RBFs were set to 0.55, 0.65 and 0.75 for the actor, the critic and the process model, respectively. The learning curve for MLAC-FH is shown in Figure 4.2a. It clearly suffers from a dip in the return too, but once the process model has learned a fair part of the transition function $f(x,u)$ after about 3 minutes, the learning curve becomes much steeper than the curve for SAC-FH around that same time. Similarly to the infinite horizion case, the addition of a learned process model boosts the learning speed.

The time response of the policy produced by MLAC-FH in Figure 4.2b is similar to the one produced by SAC-FH. Apart from having learned the mirrored solution, the only clear difference is that the input signals is a bit less smooth. Nevertheless, it can not be concluded that SAC-FH learns smoother

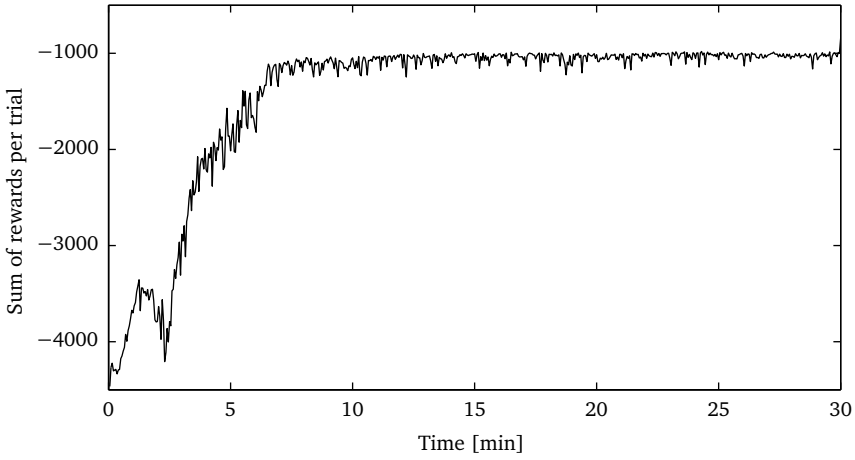


(a) Learning curve

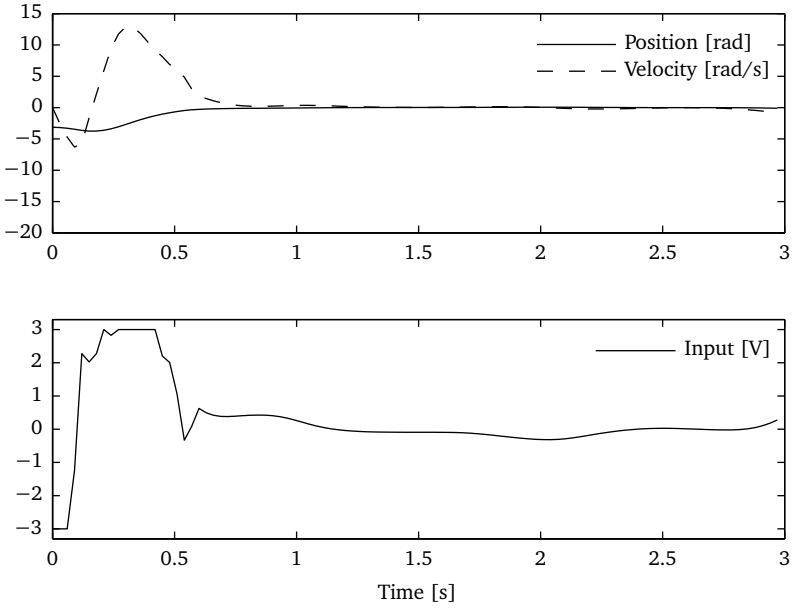


(b) Time response

Figure 4.1 Results for SAC-FH using tuned parameter sets.

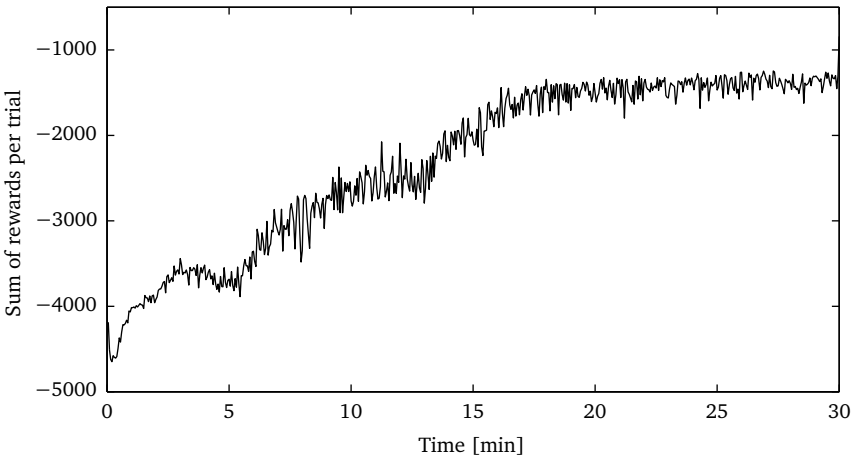


(a) Learning curve

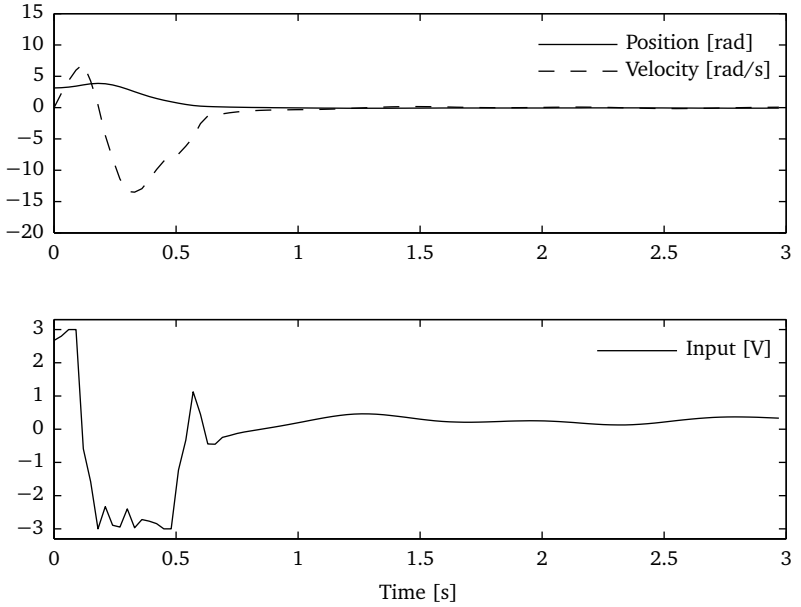


(b) Time response

Figure 4.2 Results for MLAC-FH using tuned parameter sets.



(a) Learning curve



(b) Time response

Figure 4.3 Results for RMAC-FH using tuned parameter sets.

Table 4.1 Averages μ and sample standard deviations s of the accumulated rewards during the last episode of the SAC-FH and MLAC-FH experiments.

	μ	s
SAC-FH	−995.7016	199.4336
MLAC-FH	−835.3521	55.3719
RMAC-FH	−840.7689	37.9602

policies than MLAC-FH, as the given time responses of both algorithms are mere examples of what a typical policy produced by the algorithms can look like.

4.4.3 Finite Horizon Reference Model Actor-Critic

Figure 4.3a shows the learning curve for RMAC-FH, produced by using the tuned learning parameters $\alpha_r = 0.6$, $\alpha_c = 0.2$, $\alpha_r = 0.6$. The number of RBFs in each dimension were again held fixed at 9 and the widths of the RBFs were set to 0.7 for the reference model and the critic and 0.6 for the process model. Although no sharp dips are observed, the algorithm clearly is not learning as fast as SAC-FH and MLAC-FH. The most likely culprit is the reference model, since the way in which the process model is defined and updated is exactly the same as with MLAC-FH. The fact that the optimisation procedure explained in Section 3.4 now has to use a reference model which that has an extra time dimension in its input space is slowing down the learning speed considerably.

Figure 4.3b shows an example of a time response produced by RMAC-FH. The only significant difference with the time responses seen before with SAC-FH and MLAC-FH is that the part between $t = 0.2\,\mathrm{s}$ and $t = 0.5\,\mathrm{s}$ is not smooth, which is likely to be caused by the use of extreme values of inputs when calculating the desired next state.

When looking at the average and sample standard devation of the return in the final episode for SAC-FH, MLAC-FH and RMAC-FH given in Table 4.1, the conclusion is that the model learning methods clearly win over the standard actor-critic method. The higher average reward means that they learn better solutions to the problem and the low sample standard deviations show that they also learn these solutions in a more consistent way. Moreover, MLAC-FH has shown to be a very fast learning method. The reason why the values in

Table 4.1 do not seem to correspond to Figures 4.1a, 4.2a and 4.3a is that the last episode is carried out *without* exploration, in contrast to the other episodes, which are dominantly seen in the figure.

4.5 Discussion

This chapter introduced finite horizon adaptations of the actor-critic algorithms in Chapter 3, in order to learn optimal control solutions for a finite horizon task on a nonlinear system. Results from simulation confirm the successful adaptation of the algorithms to the finite horizon case. As with the infinite horizon versions introduced in Chapter 3, the use of a learned process model significantly enhances the quality of the obtained policy and in the case of MLAC-FH also the learning speed. This can be seen from Figure 4.4, where the learning curves are put together in one plot.

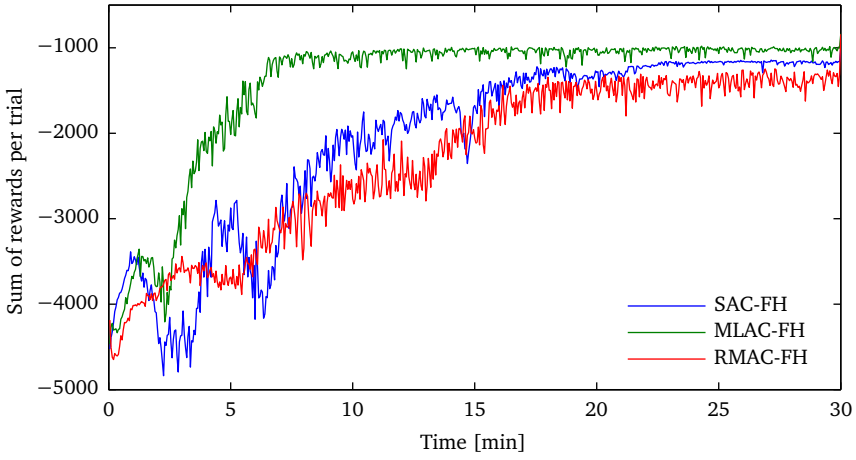


Figure 4.4 Learning curves for SAC-FH and MLAC-FH.

Chapter

5

Simulations with a Two-Link Manipulator

To further test the applicability of the actor-critic algorithms presented in this thesis, this chapter discusses simulations of the algorithms on a two-link manipulator. The extra difficulty introduced here is that the system now has multiple inputs, whereas the previous chapters only dealt with single input systems. As such, the algorithms now have to learn a vector-valued policy, in order to control the system. Moreover, the order of the system is now four instead of two in the case of the inverted pendulum, meaning that both actor and critic have to cover a domain that is much larger.

After first discussing the setup of the manipulator and the effects the higher order system has on the model learning methods, the experiment is divided into two parts. In both parts, the goal is to make the dynamical system reach an equilibrium state. In the first part, the two-link manipulator is suspended vertically and the the task is to make the system learn how to reach the trivial equilibrium point, i.e. the point in which the potential energy in the system is reduced to a minimum. The second part uses the two-link manipulator in a horizontal plane, which implies that gravity is no longer of influence, and the task is to steer the system to any reference state.

5.1 Simulation Setup

The experimental setup used in this chapter is a manipulator, consisting of two links and two frictionless rotational joints. One of the joints has a fixed position, the other connects both links. The system has two degrees of freedom and is fully actuated by two motors driving the joints. A complete description, including a table with the physical constants used, and a picture of this system is given in Appendix A.2.

5.2 Consequences for Model Learning Methods

Now that a fourth order system is considered, the domains of the actor and critic, which for both is the state space, obviously become much larger. Moreover, the learned process model and reference model will not only have larger domains, but also larger codomains, as they have a state (which now has four elements) as their output. More specifically, for a fourth order system with two control inputs, the learned process model now has six input arguments and an output of four elements. In the inverted pendulum case, this was three input arguments and two outputs. A learned reference model now has four inputs as well as four outputs, whereas for the inverted pendulum this was two inputs and two outputs. Applying dynamic programming and reinforcement learning to higher-order problems is a tough exercise, because of the curse of dimensionality. For example: if ten basis functions per dimension are used for the learned process model of the inverted pendulum, this means that the process will be modelled by a thousand basis functions in total. For the two-link manipulator, the doubled number of input arguments means that *a million* basis functions are needed for the process model if the same resolution in each dimension is to be kept. This exponential growth of the number of required basis functions and corresponding parameters to estimate is definitely a problem for parametric function approximators. A direct consequence is that the simulation times of MLAC and RMAC using radial basis functions grew dramatically, making them practically infeasible to use. Therefore, the test cases are only performed with the standard actor-critic algorithm using both types of function approximators and the model-learning actor-critic algorithm with only the LLR function approximator. For RMAC, the simulations turned out to be computationally too heavy even when LLR was the function approximator used.

For non-parametric function approximators, it is not quite clear if doubling the number of input arguments would also imply that, in the case of LLR for example, the number of samples kept in the memory would need to be squared. The test cases in this chapter will provide more insight into this subject. The learning parameters used in the simulations for this chapter are all listed at the end of this chapter.

5.3 Case I: Learn to Inject Proper Damping

In this first test case, the two-link manipulator is suspended in the vertical plane, which means that its motion is influenced by the force of gravity. With no friction or damping present in the system, a non-zero initial position of the manipulator will result in an endless motion if the system is left uncontrolled, so $u_i = 0$ for $i = 1, 2$. This motion is illustrated in Figure 5.1.

The objective in this test case is to bring the system to the state which in practice would be a trivial equilibrium: the state in which the angles of both links are zero, making the complete setup point downwards, i.e. where the potential energy in the system is equal to zero. This requires the controller to inject damping into the system, such that it will come to a standstill in the downward position.

The state x is defined as

$$x = \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \dot{\varphi}_1 \\ \dot{\varphi}_2 \end{bmatrix}$$

with φ_i the angle of link i and $\dot{\varphi}_i$ its angular velocity. The reward function used in this test case is similar to the one used for the inverted pendulum. The angle and angular velocity of both links as well as both inputs u_i are weighted the same way as before, i.e.

$$\rho(x, u) = -x^\top Q x - u^\top P u \tag{5.1}$$

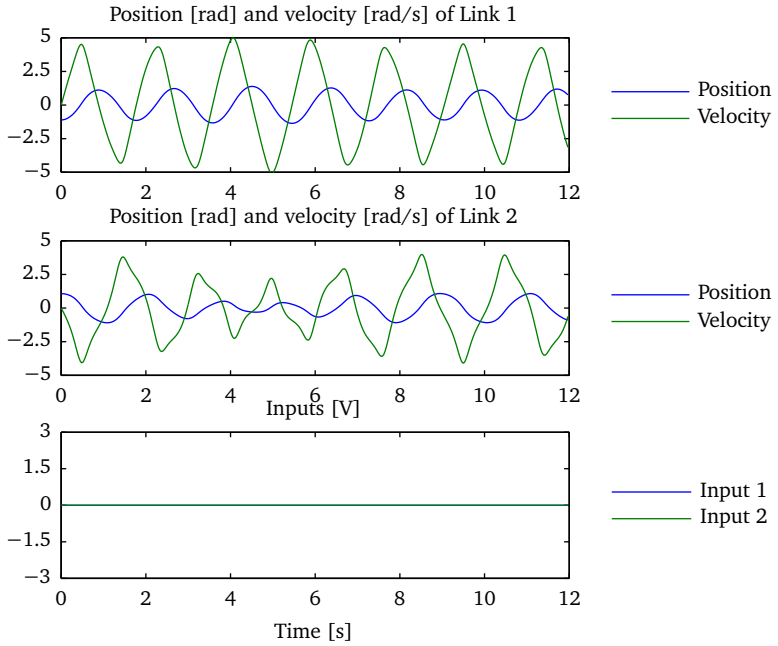


Figure 5.1 The uncontrolled motion of the two-link manipulator suspended in the vertical plane.

with

$$Q = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The standard actor-critic method SAC and MLAC and are applied to task described above, using both RBFs and LLR as approximators in the case of SAC and only LLR as the approximator in the case of MLAC. The algorithms run for 60 minutes of simulated time, consisting of 600 consecutive trials with each trial lasting 6 seconds, which is enough to make the manipulator come to a standstill. Every trial begins in with the initial state $x_0 = [-1.1\ 1\ 0\ 0]^\top$. A *learning experiment* is defined as one complete run of 600 consecutive trials.

The sum of rewards received per trial is plotted over the time which results in a learning curve. This procedure is repeated for 20 complete learning experiments to get an estimate of the average learning speed.

Tuning of all algorithm/approximator combinations was done by iterating over a grid of parameters, which included parameters of the algorithm (i.e. the learning rates) as well as parameters of the function approximator (neighborhood sizes in case of LLR and widths of RBFs, for example). For all experiments, the sampling time was set to 0.03 s, the reward discount rate γ to 0.97 and the decay rate of the eligibility trace to $\lambda = 0.65$. Exploration is done every time step by randomly perturbing the policy with normally distributed zero mean white noise with standard deviation $\sigma = 1$.

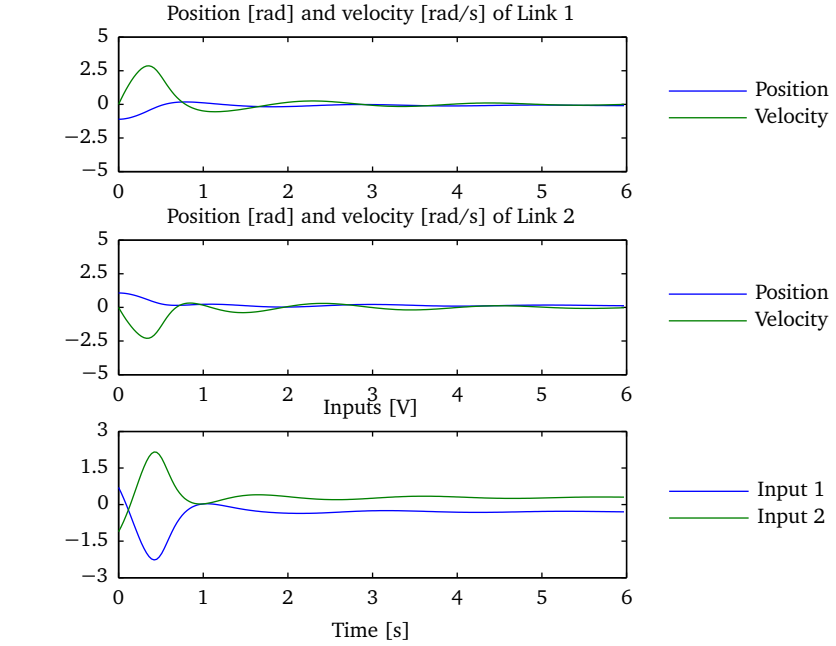
The next sections will describe the results of the simulation, ordered by algorithm. All the results shown are from simulations that used an optimal set of parameters after tuning them over a large grid of parameter values. For each algorithm, the simulation results are discussed separately for each type of function approximator.

5.3.1 Standard Actor-Critic

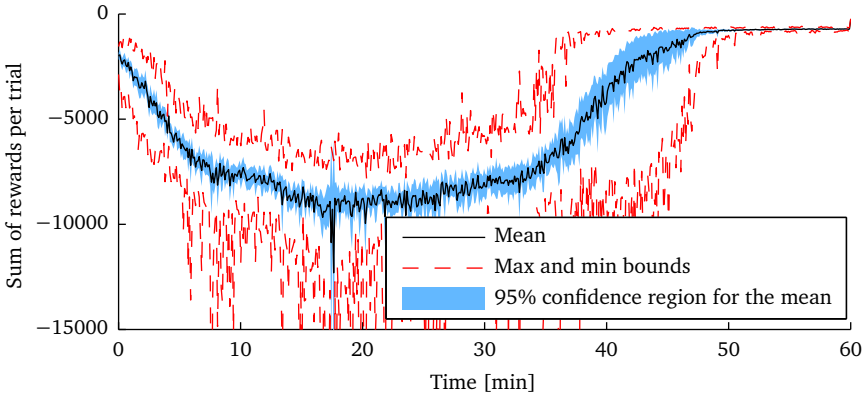
Figure 5.2 shows the results of learning a controller with the SAC algorithm, using RBFs as the function approximator. The tuned learning rates for the actor and critic are $\alpha_a = 0.25$ and $\alpha_c = 0.4$, respectively. The number of RBFs in each dimension is 9 with a width of 0.9.

From Figure 5.2a, it is clear that the algorithm produces a smooth policy that has clearly learned to steer the system in a direction opposite to its current motion, as desired. Looking closer at the inputs, it is clear that they are still non-zero by the end of the trial, indicating that the position $x = 0$ is not reached perfectly.

The learning curve in Figure 5.2b shows the familiar picture of a learning curve with a large dip in the learning performance. The dip is now more pronounced as the policy does not recover as quickly as before. The most obvious reason for this is that the state/action space is now much larger and the whole space is still initialised optimistically. As a result, the learning algorithm needs more interaction with the system to get rid of the optimistic initial values and to obtain a more realistic value function and policy for the whole state space.

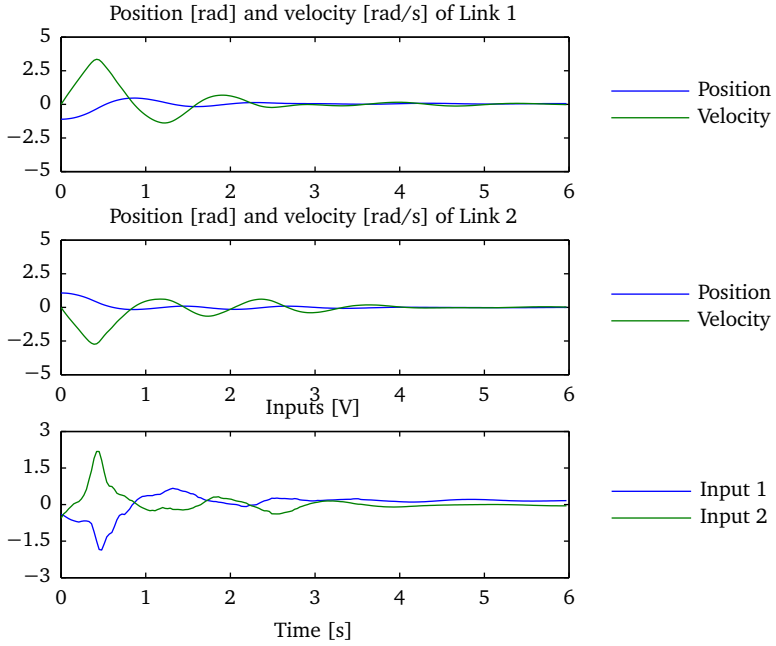


(a) Time response

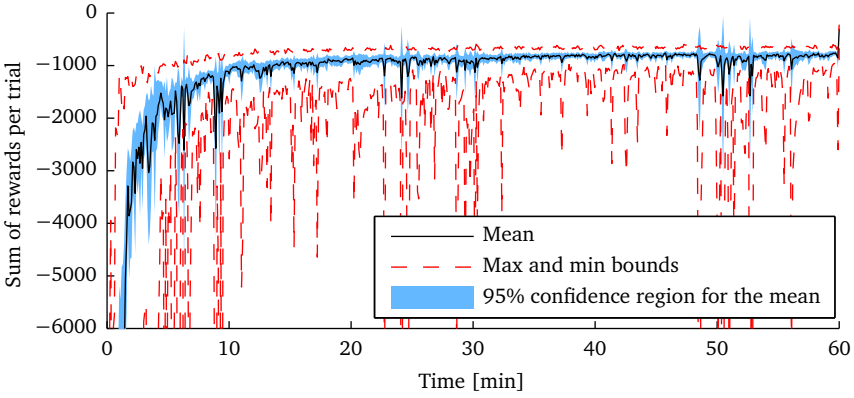


(b) Learning curve

Figure 5.2 Results for SAC/RBF using tuned parameter sets on Case I.

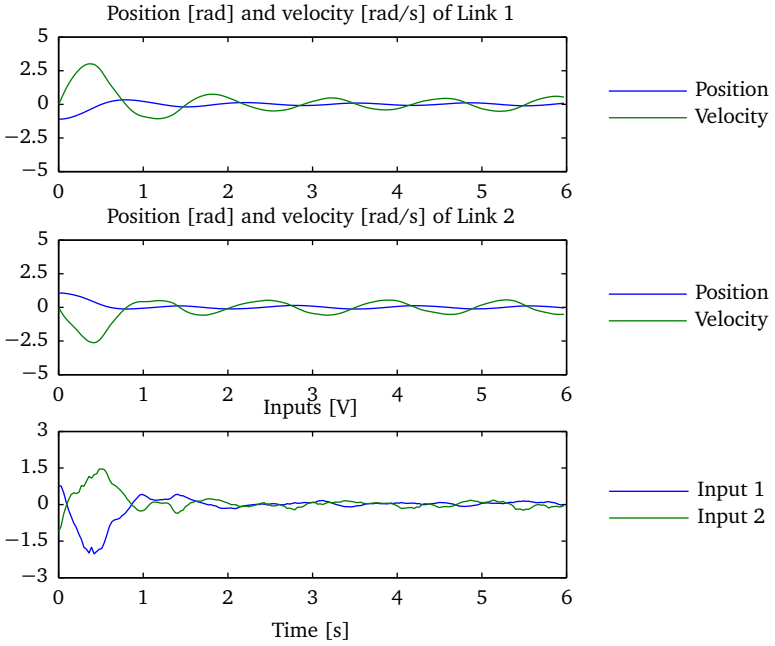


(a) Time response

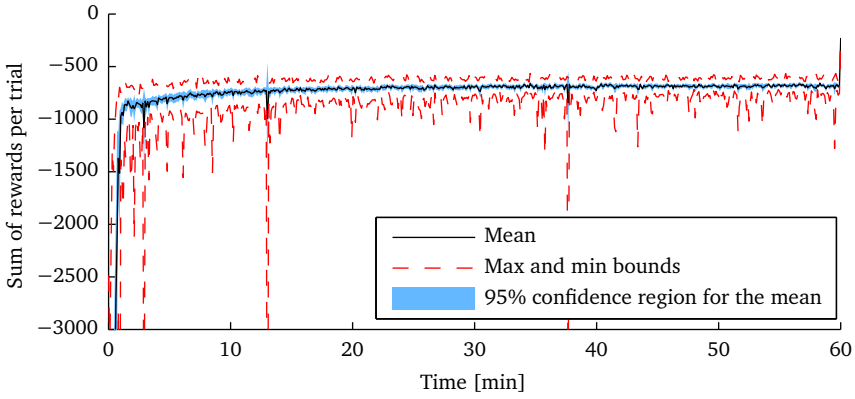


(b) Learning curve

Figure 5.3 Results for SAC/LLR using tuned parameter sets on Case I.



(a) Time response



(b) Learning curve

Figure 5.4 Results for MLAC/LLR using tuned parameter sets on Case I.

Figure 5.3 shows the results of learning a controller with the SAC algorithm, now using LLR as the function approximator. The learning rate for the actor is now much lower ($\alpha_a = 0.05$) while the learning rate for the critic is the same as in the RBF case ($\alpha_c = 0.4$). For the function approximator, both the actor and critic LLR memories contain 6000 samples and 30 nearest neighbours are used for the estimation of a value. The algorithm has still learned to steer the system in a direction opposite to its current motion (Figure 5.3a), but the final policy is not as smooth as with RBFs. On the other hand, the system now does reach the desired state more closely.

The path through which this final policy is reached is also more satisfactory. Figure 5.3b shows that a dip in the learning behaviour is no longer present (analogously to the learning behaviour seen in the previous chapters) and the confidence region is also smaller, indicating a more stable way of learning. Moreover, note that the scale of the reward axis is much smaller.

5.3.2 Model Learning Actor-Critic

Figure 5.4a shows that the MLAC algorithm is capable of injecting some damping into the system, but unfortunately can not get the system in a steady state, as it keeps oscillating slightly around the origin. The learning rates for the actor and critic here are $\alpha_a = 0.15$ and $\alpha_c = 0.35$, respectively. The LLR memories for the actor and critic again contain 6000 samples, with 30 nearest neighbours used for approximations. The LLR memory for the process model contains only 600 samples and uses 25 nearest neighbours for an approximation.

In Figure 5.4b, the confidence region reduced even further in size, indicating an even more stable way of learning, also supported by the narrow distance between the maximum and minimum learning results of all the experiments.

Figure 5.5 shows a graph of the learning curves for Case I for all three algorithms used. There is a dramatic difference in the performance of the algorithms using LLR as their function approximator and the performance of the SAC algorithm using RBFs. Model learning is again showing its advantage over the other algorithm, with MLAC/LLR already reaching a policy close to its final result in less than 3 minutes of simulated time, whereas SAC/LLR reaches a similar result after 10 minutes and SAC/RBF only reaches it after more than 45 minutes.

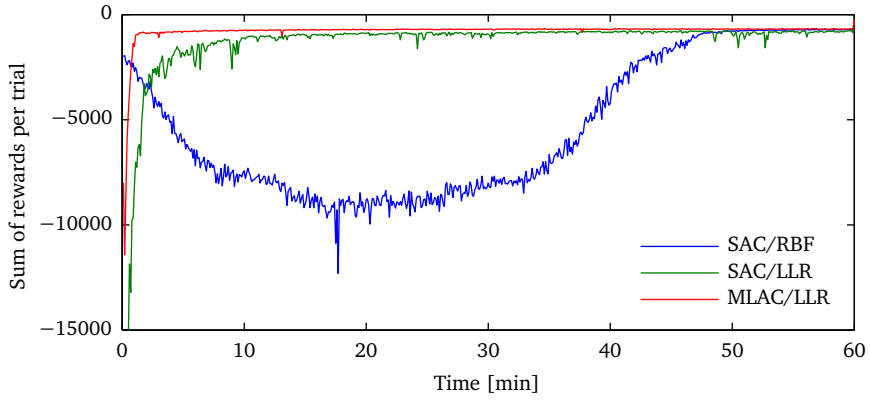


Figure 5.5 Learning curves for Case I. The time axis represents the *simulated* time.

5.4 Case II: Learn to Find a Nontrivial Equilibrium

As a second test case, the two-link manipulator is now lying in the horizontal plane, effectively cancelling out the effects of gravity on the system, i.e. setting $G(\varphi)$ in the equations of motion equal to zero. The task now is to go from the initial state $x_0 = [-3\ 1\ 0\ 0]^\top$ to a reference state $x_r = [1.1\ \ -0.5\ 0\ 0]^\top$. There is still no damping or friction present in the system. In order to carry out the task succesfully, the controller has to learn how to both accelerate and decelerate the links such that the desired end position is reached.

The setup of the experiments and the learning parameters such as the discount factor γ, the decay rate λ of the eligibility traces are the same as in the previous case. The reward function used here is almost the same as before, but instead of using the state x in the reward function, the error e between the current state and the desired state, defined as

$$e = \begin{bmatrix} \text{wrap}(\varphi_1 - \varphi_{1,r}) \\ \text{wrap}(\varphi_2 - \varphi_{2,r}) \\ \dot{\varphi}_1 - \dot{\varphi}_{1,r} \\ \dot{\varphi}_2 - \dot{\varphi}_{2,r} \end{bmatrix}$$

is used, where the variables with the r subscript indicate the reference. Note that the error in the angles is wrapped, such that the difference between the

current angle and the reference angle always lies in the range $[-\pi, \pi)$. The reward function then becomes

$$\rho(x, u) = -e^\top Q e - u^\top P u, \tag{5.2}$$

with Q and P defined as before in Case I.

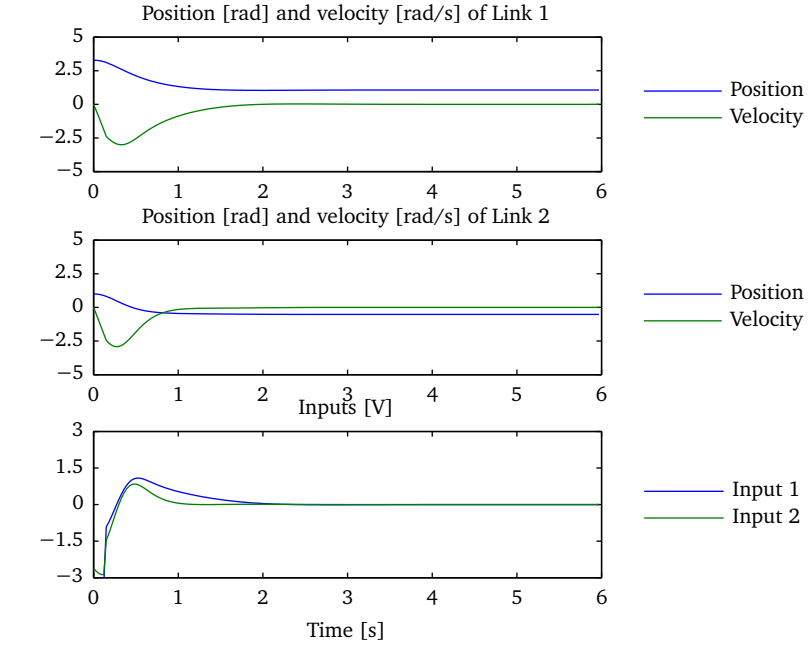
5.4.1 Standard Actor-Critic

The result of learning a policy for the second test case with the standard actor-critic algorithm using radial basis functions as a function approximator for the actor and critic is shown in Figure 5.6. The tuned learning rates here are $\alpha_a = 0.1$ for the actor and $\alpha_c = 0.3$ for the critic. The number of RBFs in each dimension is still 9, but the intersection height of the RBFs is now set at 0.7.

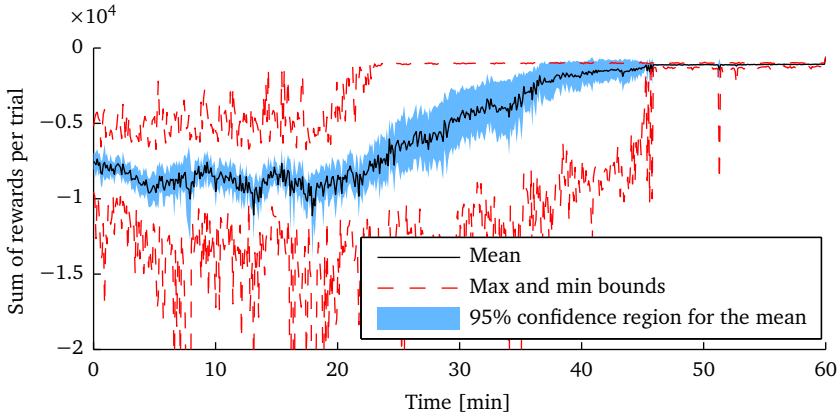
The quality of the policy is striking (Figure 5.6a). The final policy reaches the desired state very smoothly within two seconds. The policy's smoothness is caused by the fact that the function approximator is inherently smooth. The speed of learning depicted in Figure 5.6b is again somewhat disappointing, again needing at least 40 minutes to reach a policy close to the end result. One peculiarity here is that after about 45 minutes, the confidence region is collapsing, meaning all the different experiments have learned policies which are very close in terms of the sum of collected rewards per trial they produce. This behaviour was also seen in Figure 5.2b, albeit that the collapse of the confidence region and the max and min bounds was less pronounced there. The reason for the sudden collapse shown here is unknown.

The effects of using a non-parametric LLR approximator in the standard actor-critic algorithm are demonstrated in Figure 5.6. As with Case I, the learning rate of the actor is $\alpha_a = 0.05$, but the critic learning rate is slightly lower at $\alpha_c = 0.35$. The number of samples in the LLR memories for both the actor and the critic is now only 5000, whereas tuning in Case I prescribed 6000 samples. Although the task itself is slightly more difficult, the critic also contains an implicit model of the process and since gravity is not considered here, the model—and therefore the critic—are less complex, which explains the reduction in the number of necessary samples. The number of neighbours used in approximations is still 30 for both the actor and critic.

Similar to the first test case, the time response in Figure 5.7a is again a bit more erratic, but still achieves the task of reaching the reference state, albeit that it now needs three seconds to reach a steady state. The learning curve

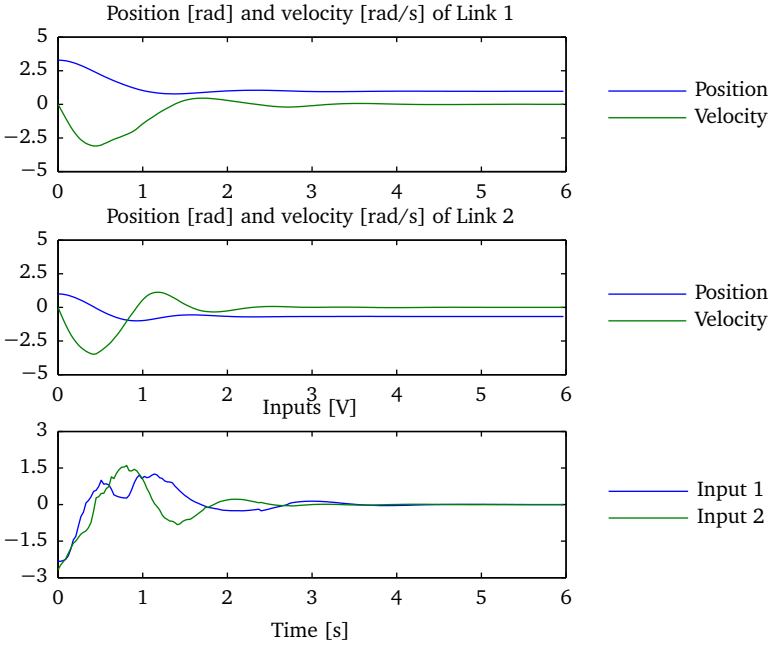


(a) Time response

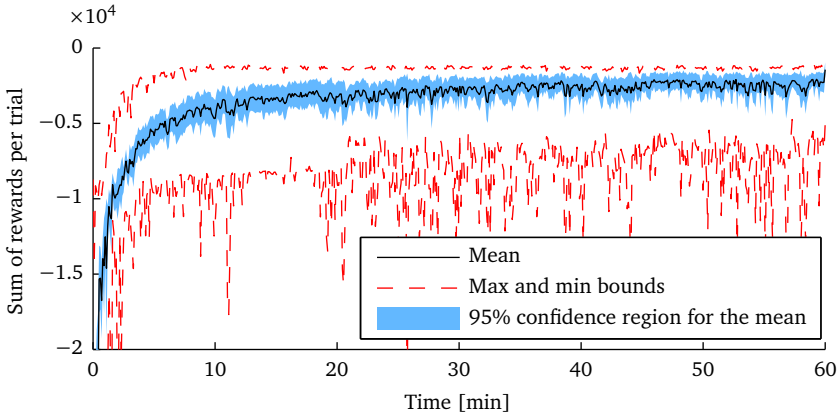


(b) Learning curve

Figure 5.6 Results for SAC/RBF using tuned parameter sets on Case II.



(a) Time response



(b) Learning curve

Figure 5.7 Results for SAC/LLR using tuned parameter sets on Case II.

in Figure 5.6b is also meeting expectations: the quality of the policy steadily increases without a dip and converges quite quickly, with a confidence region for the mean which is again a bit smaller than in the RBF case, except for the last part. The confidence region is not collapsing now, because LLR keeps adding and removing samples to the memories of the actor, critic and process model. As exploration is not decaying towards the end of an experiment, the samples added to the memories can vary greatly in each experiment, in turn having a big influence on the function approximation and, as a result, on the learned policy.

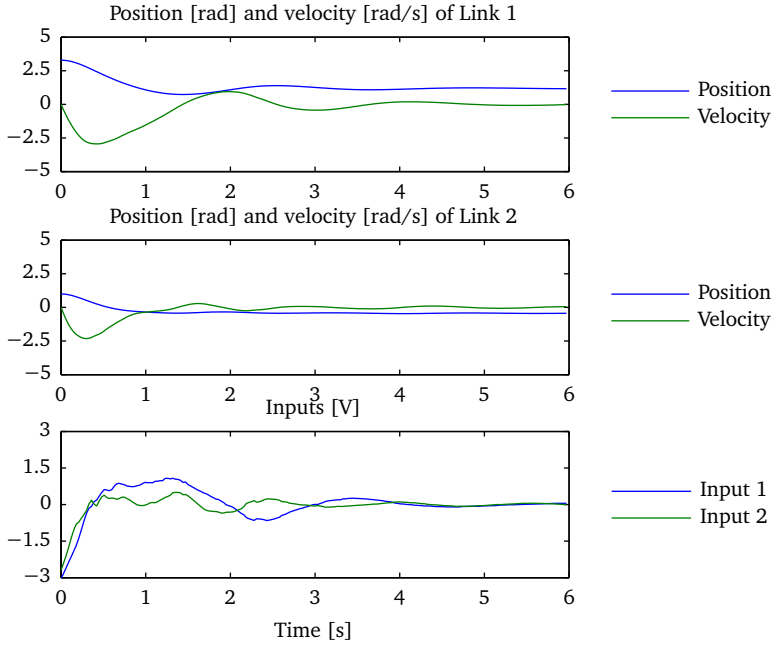
5.4.2 Model Learning Actor-Critic

Figure 5.8 shows the time response of the MLAC algorithm using LLR as the function approximator. The learning rates changed slightly to $\alpha_a = 0.1$ and $\alpha_c = 0.4$ and the number of samples in the actor and critic LLR memories increased slightly to 7000 in both cases. The number of nearest neighbours used for estimation of values is now 20 instead of the 30 in Case I. The number of samples needed in the LLR memory of the process model is now 500 instead of 600, which again underpins the fact that the model indeed is simpler in this test case. The number of nearest neighbours used in the process model is 15.

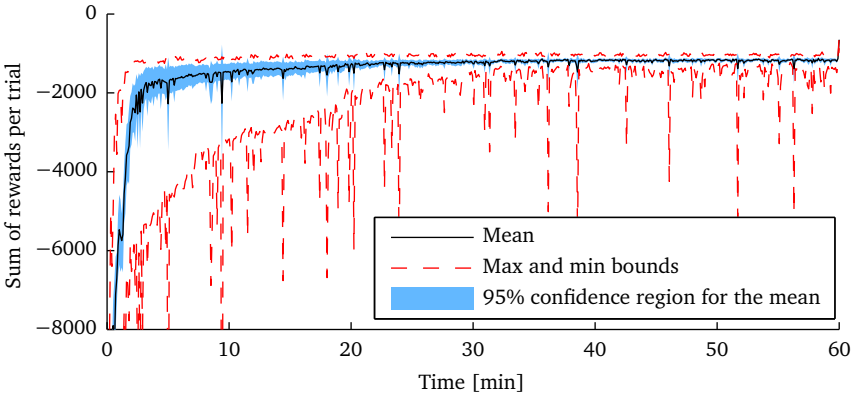
In terms of time-optimality, the policy clearly has degraded a bit (Figure 5.8a), but nevertheless the system does reach the desired steady state by the end of the trial. Figure 5.8b again shows a very quick and steady learning behaviour, converging in less than 5 minutes.

Although the combination of MLAC/LLR seems to produce the worst policy of all three combinations, Figure 5.9 reveals that in terms of collected rewards it is outperforming SAC/LLR and is only slightly behind SAC/RBF. Furthermore, MLAC/LLR reaches this final policy a lot sooner than both other methods. After just five minutes, the MLAC/LLR policy is already close to its final policy and way ahead of the other two combinations, which need at least another 20 or 40 minutes, for SAC/LLR and SAC/RBF respectively, to learn a policy that is equivalent with respect to the return they yield.

Nevertheless, a case could clearly be made that MLAC/LLR is producing the worst policy in terms of the time response. Hence, a reasonable statement to make here is that the somewhat poor quality of the MLAC/LLR policy is due to the definition of the reward function. Apparently, the reward function



(a) Time response



(b) Learning curve

Figure 5.8 Results for MLAC/LLR using tuned parameter sets on Case II.

was not capable of distinguishing a very good policy from a mediocre one, when considering only the time response. This stresses the importance of a well-chosen reward function in order to learn successfully in terms of collected rewards *and* time response.

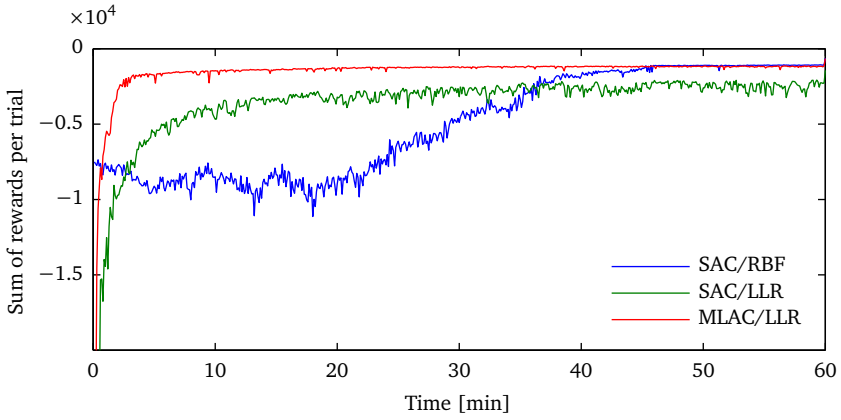


Figure 5.9 Learning curves for Case II. The time axis represents the *simulated* time.

5.5 Discussion

This chapter demonstrated the use of some of the actor-critic algorithms described in this thesis to a two-link manipulator. A perhaps obvious, but important point made at the start of this chapter is that model learning methods lose their applicability in higher order systems. This is due to the fact that (co)domains of the functions involved (the process model and possibly a reference model) get too large to be learned efficiently. This is not to say that MLAC and RMAC do not work at all in this setting. It is their computational complexity that makes simulation times rise too high for rigorous tuning to be done, prohibiting to see if there would be a suitable choice of parameters that would yield good results. It is also not true that a more simple algorithm like SAC would necessarily outperform the model learning methods in terms of simulation time: the evaluation of thousands of radial basis functions at each time step is something that will have a considerable effect too.

Good policies were retrieved with SAC using both RBFs and LLR as function approximators. With MLAC, success within reasonable simulation time was only possible using the non-parametric LLR function approximator. For the RMAC algorithm it turns out that even with LLR, the extra effort of having to learn an extra (reference) model is simply too much and RMAC was therefore not considered in this chapter.

Looking at Figures 5.5 and 5.9, it is obvious that the learning speed pattern is the same for both test cases. The standard actor-critic algorithm is without a doubt the slowest to learn a policy that performs reasonably well. Again, although not as pronounced as in the previous chapters, a slight dip in the learning performance can be seen between 10 and 30 minutes of simulated time which is caused by the optimistic initialisation of the RBFs. Using LLR instead of RBFs as the function approximator again overcomes this problem and the learning curve changes significantly, learning a good policy in about 25-50% of the time it takes with RBFs. Adding model learning, i.e. using MLAC, improves the learning behaviour even more, again reducing the time to learn an acceptable policy considerably.

The quality of the policies learned using the LLR approximator, both with and without model learning, is not as good as the quality of the policy learned by SAC/RBF. As explained earlier, the smoothness of the policy that comes by design when using RBFs is a favourable factor here. To get a smoother policy using LLR, one would probably have to choose a larger number of neighbours used in the approximation, although this will undoubtedly degrade the performance, given that experiments were actually done using bigger sets of neighbours, but without showing better a better accumulation of rewards in the final episode. The fact that the return on the last episode of MLAC/LLR in Figure 5.9 is about the same as that of SAC/RBF indicates that the chosen reward function apparently is unable to distinguish a policy of a good quality from a policy with lesser quality in terms of time response.

Chapter

6

Learning Rate Free RL Using a Value-Gradient Based Policy

The learning speed of RL algorithms heavily depends on the learning rate parameter, which is difficult to tune. In this chapter, a sample-efficient, learning-rate-free version of the Value-Gradient Based Policy (VGBP) algorithm is presented. The main difference between VGBP and other frequently used RL algorithms, such as SARSA, is that in VGBP the learning agent has a direct access to the reward function, rather than just the immediate reward values. Furthermore, the agent learns a process model (see Chapter 3). The combination of direct access to the reward function and a learned process model enables the algorithm to select control actions by optimising over the right-hand side of the Bellman equation. Fast learning convergence in simulations and experiments with the underactuated pendulum swing-up task is demonstrated. In addition, experimental results for a more complex 2-DOF robotic manipulator are discussed.

6.1 Introduction

The first chapters of this thesis already described that one of the main disadvantages of RL is the slow learning caused by inefficient use of samples (Wawrzyński, 2009). Numerous methods have been devised to improve sample efficiency, like eligibility traces (Sutton and Barto, 1998), experience replay (Adam et al., 2011; Wawrzyński, 2009) and methods that learn a

process model (Sutton, 1990). Model learning RL algorithms are related to approximate dynamic programming (Powell, 2010). In Dyna (Sutton, 1990), the process model is used to increase the learning speed of the value function by simulating interactions with the system. The model learning actor-critic (MLAC) and reference model actor-critic (RMAC) algorithms described in Chapter 3 use a learned process model in efficient actor updates that lead to faster learning. MLAC uses a model solely in the actor update. Like in Heuristic Dynamic Programming (HDP) (Si et al., 2004), the model is used to estimate the policy gradient. The actor is updated with a gradient ascent update rule. In RMAC the model is used to train a reference model, which contains a mapping from states to desired next states. The action needed to go to the desired next state is obtained through a local inverse of the learned process model.

The success of experience replay and model learning RL methods gives rise to the question if there is more information that can be provided to the agent to enable it to take better decisions after a shorter learning period. In the standard RL framework both the reward function and the system are considered to be part of the environment (Sutton and Barto, 1998). However, in most cases the reward function is designed by an engineer and therefore can be easily provided to the learning agent. With the reward function and a learned process model the agent has complete knowledge of the environment, which enables it to select optimal actions in a more effective way. An RL method that selects its actions with this full view is the Value-Gradient Based Policy (VGBP) algorithm (Doya, 2000). It selects the action by optimising the right-hand side of the Bellman equation. Because only the process model and the critic have to be learned and not an explicit actor, it is a critic-only method. VGBP has a number of favourable properties. By the efficient use of the available information it achieves fast learning. It is a critic-only method that is able to generate a smooth continuous policy that is (very close to) the optimal signal. When the system dynamics are linear and the reward function is quadratic, the agent generates the same actions as a linear quadratic regulator (Doya, 2000). Furthermore, prior knowledge about the system dynamics can be incorporated by initialising the process model.

The contributions of this chapter are the following. A novel variant of the Value-Gradient Based Policy algorithm is introduced, which is based on Least Squares Temporal Difference (LSTD) learning (Boyan, 2002) for the critic. Thanks to the use of LSTD, the VGBP algorithm no longer requires the critic learning rate and therefore becomes easier to tune. The learning performance

of this algorithm is compared with the SARSA algorithm (Sutton and Barto, 1998) and the model based MLAC algorithm of Section 3.3, to see the benefits of the incorporation of the reward function knowledge into the agent. SARSA is probably the most well-known critic-only method and this method is used to compare VGBP to general on-policy critic-only methods. Both MLAC and VGBP use the process model to compute the gradient of the return with respect to the policy parameters. Because of this similarity, a comparison between MLAC and VGBP provides more insight in the use of a process model and a reward function in continuous reinforcement learning. Different function approximation techniques are applied to VGBP and it is shown that it not only achieves quick learning in simulation, but also works very well in real-time experiments with two benchmarks: the pendulum swing-up and a 2-DOF manipulator.

The remainder of the chapter is structured as follows. The SARSA algorithm used in this chapter for comparison is described in Section 6.2. In Section 6.3, the learning-rate-free version of the VGBP algorithm is introduced. The performance of VGBP is tested and compared with SARSA and MLAC on two systems in Section 6.4. Section 6.5 concludes the chapter. The function approximators used for the critic and the process model are described in Section 3.5.

6.2 SARSA

In this section, SARSA, an algorithm comparable to VGBP is introduced. This method learns the value function of the currently used policy, which means it is an *on-policy* algorithm. SARSA selects actions based on the Q-function. The Q-function is learned by minimising the temporal difference (TD) error

$$\delta_{k+1} = r_{k+1} + \gamma Q_{\theta_k}(x_{k+1}, u_{k+1}) - Q_{\theta_k}(x_k, u_k), \tag{6.1}$$

which is simply the difference between the left and right hand side of the Bellman Equation. The name SARSA is an acronym (State-Action-Reward-State-Action) for the tuple $(x_k, u_k, r_{k+1}, x_{k+1}, u_{k+1})$ used in the update equation. The Q-function parameters are updated by means of the gradient descent formula

$$\theta_{k+1} = \theta_k + \alpha_k \delta_{k+1} \frac{\partial Q_{\theta_k}(x_k, u_k)}{\partial \theta}, \tag{6.2}$$

where $\alpha_k > 0$ is the learning rate. To guarantee convergence the learning rate has to satisfy the Robbins-Monro conditions (Robbins and Monro, 1951)

$$\alpha_k > 0 \quad \forall k \qquad \sum_k \alpha_k = \infty \qquad \sum_k \alpha_k^2 < \infty. \tag{6.3}$$

Furthermore, all state-action pairs need to be visited infinitely often. In theory, this is ensured by sometimes taking exploratory actions instead of greedy actions. A simple exploration method is *ϵ-greedy action selection*: with probability $1-\epsilon$, where $\epsilon \in (0,1)$, the greedy action is taken and with probability ϵ a random action is chosen from the action space U. Approximate SARSA converges with probability 1 if the exploration term ϵ decays to zero and the policy satisfies certain regularity conditions (Melo et al., 2008).
The update equations (6.1) and (6.2) assign the current reward r_{k+1} only to the latest transition. However, the current reward is the result of an entire trajectory. Learning can be sped up by using *eligibility traces* $z \in \mathbb{R}^p$ that allow the current reward r_{k+1} to update the Q-values of the recently visited states. The update equation with eligibility traces is

$$z_k = \lambda\gamma z_{k-1} + \frac{\partial Q_\theta(x_k,u_k)}{\partial\theta}, \tag{6.4}$$

$$\theta_{k+1} = \theta_k + \alpha_k \delta_{k+1} z_k, \tag{6.5}$$

where $\lambda \in [0,1]$ is the trace decay parameter and $z_0 = 0$.

6.3 Value-Gradient Based Policy Algorithm

Critic-only algorithms select the greedy action u by finding the maximum

$$\pi(x) = \arg\max_u Q(x,u).$$

This function which is optimized over is the left-hand side of

$$Q^\pi(x,u) = \rho(x,u) + \gamma V^\pi(x').$$

The right-hand side of this equation can be used to select the action if both the reward function and the transition model are known. In most cases the reward

function is known and can therefore be provided to the agent. Given a process model $\widehat{f}$ learnt from data, the optimal action is then found by solving

$$u = \arg\max_{u'} \left(\rho\left(x, u'\right) + \gamma V_\theta \left(\widehat{f}(x, u') \right) \right). \tag{6.6}$$

Because $\rho(x,u)$, $V_\theta(x)$ and $f(x,u)$ are in general highly nonlinear, the optimisation problem in Equation (6.6) can be hard to solve. If V_θ is smooth enough to be approximated by first order Taylor series and the process model is approximately linear in the control action, the Q-function can be approximated by

$$\begin{aligned} Q_\theta(x,u) &= \rho(x,u) + \gamma V_\theta(x') && (6.7)\\ &\approx \rho(x,u) + \gamma V_\theta(x) + \gamma \frac{\partial V_\theta(x)}{\partial x}(x' - x) && (6.8)\\ &\approx \rho(x,u) + \gamma V_\theta(x) + \gamma \frac{\partial V_\theta(x)}{\partial x} \frac{\partial \widehat{f}(x,u)}{\partial u} u. && (6.9) \end{aligned}$$

The nonlinearity in the reward function remains. If the reward function is concave, a unique solution can be easily found by setting the derivative of Equation (6.9) with respect to the action u equal to zero, yielding

$$-\frac{\partial \rho(x,u)}{\partial u} = \gamma \frac{\partial V_\theta(x)}{\partial x} \frac{\partial \widehat{f}(x,u)}{\partial u}. \tag{6.10}$$

To guarantee exploration, a zero-mean noise term Δu is added to the action. As the computed action including exploration may exceed the boundaries of the action space U it needs to be clipped with

$$u \leftarrow \mathrm{sat}(u + \Delta u) \tag{6.11}$$

where sat is an appropriate saturation function.

6.3.1 Process Model Parametrisation

The process model $x' = \widehat{f}(x,u)$ is approximated by using LLR. This approximator is chosen because it can learn a locally accurate process model from just a few observations. Samples $\zeta_i^P = [x_i^\top \; u_i^\top \mid {x'}_i^\top]^\top$ of the state transitions are stored in the memory M^P. The transition due to action u_k in state x_k is predicted with

$$x_{k+1} = \begin{bmatrix} \beta_x^P & \beta_u^P & \beta_b^P \end{bmatrix} \begin{bmatrix} x_k \\ u_k \\ 1 \end{bmatrix}. \tag{6.12}$$

The superscript P denotes the process model parameter and the subscripts denote the input variable which the parameter multiplies. The derivative of the process model needed in Equation (6.10) is then simply $\frac{\partial \widehat{f}(x,u)}{\partial u} = \beta_u^P$.

6.3.2 Critic Parametrisation

VGBP can use any critic function that is differentiable with respect to the state. In this chapter two critic approximators are used: the LLR critic of Section 3.5.2 and the RBF approximator. The LLR critic collects the samples $\zeta_i^C = [x_i^\top \mid V_i]^\top$ with $i = 1, \ldots, N^C$ in the critic memory M^C. The value of a state is computed using the local affine model

$$V_\theta(x) = \begin{bmatrix} \beta_x^C & \beta_b^C \end{bmatrix} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}. \tag{6.13}$$

with $\theta = [\beta_x^C \ \beta_b^C]$. For the local affine value function approximation, the right hand side of Equation (6.10) reduces to $\gamma \beta_x^C \beta_u^P$.

The equations of the RBF critic are given in Section 3.5.1. The derivative of the value function with respect to the state is given by

$$\frac{\partial V_\theta(x)}{\partial x} = \theta^\top \frac{\partial \phi(x)}{\partial x}. \tag{6.14}$$

With a diagonal scaling matrix B as used in this chapter,

$$\frac{\partial \phi_i(x)}{\partial x} = -\phi_i(x) \left[B^{-1}(x - c_i) + \frac{\sum_j \frac{\partial \bar{\phi}_j(x)}{\partial x}}{\sum_j \bar{\phi}_j(x)} \right]. \tag{6.15}$$

Because the RBF approximator is linear in its parameters, the parameter vector θ can also be estimated by least squares. The Least Squares Temporal Difference (LSTD) algorithm (Boyan, 2002) is given by

$$z_{k+1} = \lambda \gamma z_k + \phi(x_k), \tag{6.16}$$
$$A_{k+1} = A_k + z_k (\phi(x_k) - \gamma \phi(x_{k+1}))^\top, \tag{6.17}$$
$$b_{k+1} = b_k + z_k r_{k+1}, \tag{6.18}$$
$$\theta_{k+1} = A_{k+1}^{-1} b_{k+1}, \tag{6.19}$$

where $z_0 = \phi(x_0)$, $b_0 = 0$ and A_0 is chosen as a small invertible matrix. The advantages of the LSTD critic are its high learning speed and the fact that there is no learning rate or schedule to be tuned. A disadvantage is the increased computational costs. The full algorithm for VGBP using LLR process model and a generic critic is given in Algorithm 7.

Algorithm 7 Value-Gradient Based Policy with LLR process model

Require: : γ, λ, σ

Initialise $k = 0$, x_0, θ, M^P, $z_k = 0$
Draw action $a \sim \mathcal{N}(0, \sigma)$ and clip it if necessary
for every step $k = 0, 1, 2, \ldots$ **do**
    Measure x_{k+1} and compute reward r_{k+1}
    *// **Update critic***
    $\delta_{k+1} \leftarrow r_{k+1} + \gamma V(x_{k+1}) - V(x_k)$
    $z_k = \lambda\gamma z_k + \phi(x_k)$
    Update critic
    *// **Update process model***
    Insert $[x_k^\top \; u_k^\top \mid x_{k+1}^\top]^\top$ in M^P
    *// **Choose action***
    Obtain X and Y from M^P for x_{k+1}
    $\beta^P = YX^\top(XX^\top)^{-1}$
    Find u_{k+1} by solving $-\frac{\partial \rho(x_{k+1}, u_{k+1})}{\partial a} = \gamma \frac{\partial V_\theta(x_{k+1})}{\partial x} \beta_a^P$
    Draw $\Delta u_{k+1} \sim \mathcal{N}(0, \sigma)$
    $u_{k+1} \leftarrow \text{sat}(u_{k+1} + \Delta u_{k+1})$
    Apply action u_{k+1}
end for

The critic update equations in Algorithm 7 apply directly to the RBF (or similar parametric) approximator. For the LLR approximator update, refer to Algorithm 2 in Chapter 3.

6.4 Simulation and Experimental Results

The working of VGBP is verified for the pendulum swing-up task, both in simulation and on an experimental setup. This task is a standard benchmark

in RL (Buşoniu et al., 2010). The learning behaviour of VGBP in a multidimensional action space is also tested on a two-link robotic manipulator.

6.4.1 Underactuated Pendulum Swing-Up

A full description of the inverted pendulum setup is given in Appendix A.1. The goal is to swing a pendulum up from a down-pointing position to the upright position as quickly as possible and keep the pendulum upright by applying an appropriate voltage to the DC-motor. The action space is limited to $a \in [-3, 3]$ V, which makes it impossible to bring the pendulum directly from the initial state to its upright position. Instead the agent needs to learn to build up momentum by swinging the pendulum back and forth.

Learning Experiment Description

The learning parameters used are listed in Table 6.1.

Table 6.1 Learning parameters for the underactuated pendulum experiment.

RL parameter	Symbol	Value	Units
sampling time	T_s	0.03	s
discount factor	γ	0.97	
exploration noise	σ^2	1	V
maximal action	$u_{\max}$	3	V
minimal action	$u_{\min}$	-3	V

The reward is calculated using the following quadratic function

$$\rho(x_k, u_k) = -x_k^\top Q x_k - u_k^\top P u_k, \tag{6.20}$$

with

$$Q = \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix} \qquad P = 1.$$

A learning experiment consists of 200 trials. Each trial is 3 seconds long and after the trial, the pendulum is reset to its downward position $[\pi\ 0]^\top$. A sampling time of 0.03 s is used. All simulations and experiments are repeated 20 times to acquire an average of the learning behaviour as well as a confidence interval for the mean.

Model Learning Methods

Because MLAC and VGBP differ only in the action selection, the result of the new algorithm can be clearly illustrated by using in both algorithms the same process model and critic approximator (LLR). The parameters are stated in Table 6.2.

Table 6.2 Agent parameters for simulated pendulum swing-up for MLAC and VGBP with LLR approximators.

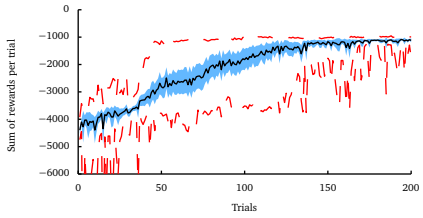
Parameter		Actor	Critic	Model
learning rate	α	0.04	0.1	
trace decay factor	λ	0	0.65	
memory size	N	2000	2000	2500
nearest neighbors	K	9	20	30
input scaling	$\mathrm{diag}(W)$	[1 0.1]	[1 0.1]	[1 0.1 1]

Given the reward function (6.20), the desired action for VGBP is given by

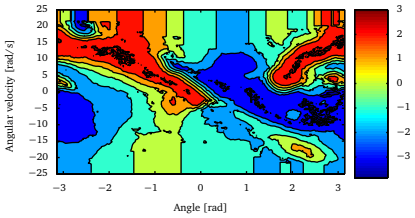
$$u = \frac{1}{2}\gamma P^{-1}\frac{\partial V_\theta(x)}{\partial x}\beta_u^P. \tag{6.21}$$

An exploratory action is applied for both MLAC and VGBP once every three time steps, where $\Delta u \sim \mathcal{N}(0,1)$. This exploration strategy generates large exploratory actions, which force the agent out of the known part of the state space. At the same time, the policy is not dominated by the exploration noise and the agent gets time to correct unsuccessful exploratory actions and adapt the critic.

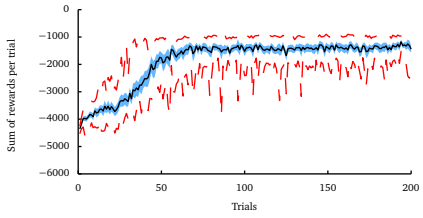
The learning behaviour including confidence bounds of MLAC is shown in Figure 6.1a. The learning converges in approximately 9 minutes of interaction time. The confidence region for the mean is very narrow, indicating that a steady swing-up is achieved each time. The learned policy is given in Figure 6.1b. The results of VGBP are given in Figure 6.1c. After 25 trials the learning speed increases, because VGBP has learned to bring the pendulum near its top position, but not yet how to stabilise it there. The algorithm converges after 3 minutes of learning. The final policy is slightly worse than that of MLAC, as VGBP needs two or three swings to get the pendulum upright. MLAC only needs two. The policy learned is shown in Figure 6.1d.



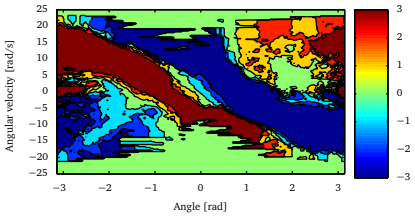
(a) Results of MLAC.



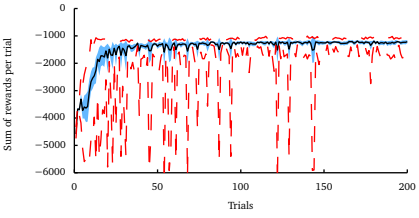
(b) Typical policy learned by MLAC



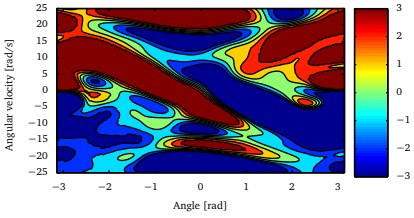
(c) Results of VGBP-LLR



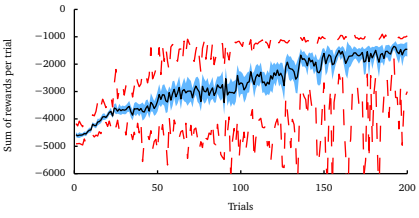
(d) Typical policy learned by VGBP-LLR



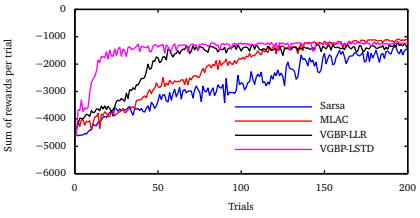
(e) Results of VGBP-LSTD



(f) Typical policy learned by VGBP-LSTD



(g) Results of SARSA



(h) Average learning behaviour of SARSA, MLAC VGBP-LLR and VGBP-LSTD

Figure 6.1 Results of 20 simulations of MLAC, VGBP-LLR, VGBP-LSTD and SARSA on the inverted pendulum swing-up task and typical learned policies.

SARSA

To see how the learning behaviour of VGBP relates to critic-only methods, SARSA is applied to the inverted pendulum. The critic uses a set of 400 RBFs with their centers placed on a 20×20 equidistant grid in the Cartesian product space of the angle domain $[-\pi, \pi]$ rad and the angular velocity domain $[-25, 25]$ rad/s. All RBFs have the same scaling matrix B. The scaling matrix is chosen such that the value of the non-normalised BF is 0.5 at the point where neighboring BFs intersect. The action space of SARSA is discrete: $U_{\text{SARSA}} = \{-3, 0, 3\}$. Each action has the same set of BFs, denoted by $\phi_b(x) \in \mathbb{R}^{400}$. This results in a parameter vector $\theta \in \mathbb{R}^p$ with $p = 3 \cdot 400 = 1200$.
The actions are selected with ϵ-greedy action selection, where ϵ is determined by

$$\epsilon = \max\left(0.1, 0.96^{\text{trial}-1}\right). \tag{6.22}$$

The decay factor 0.96 and the minimum exploration probability were found by manual tuning. The critic learning rate is set to $\alpha_c = 0.5$ and the trace decay factor to $\lambda = 0.85$. The results are shown in Figure 6.1g. SARSA needs more system interaction than the model learning algorithms, but its final performance is slightly better than that of VGBP-LLR.

LSTD Critic

To remove the dependency of VGBP on the critic learning rate, LSTD is applied. The same set of 400 equidistant BFs is used as for SARSA. The initial A matrix is set to $A = 0.5I$ and the b vector to $b_0 = 0$. This gives the value function a bias towards 0, which is an optimistic initialisation, since all rewards are negative. This stimulates exploration in the early phase of learning and prevents overfitting. The eligibility trace decay factor is set to $\lambda = 0.3$. With this relatively low value for the trace decay factor λ, the optimistic initialisation is retained longer than for bigger values of λ. Tests indicated that this is an appropriate compromise between stimulating exploration and propagating the reward to recently visited states.

The critic parameters are updated at the end of every trial, which gives steadier behaviour than when the critic is updated at each time step. The reason is that thanks to the use of LSTD, the value function, and hence the policy, can change considerably in a single update step. This leads to a noisy policy at early stages of learning. The learning progress is depicted in Figure 6.1e. A swing-up is learned in less than a minute of system interaction.

A typical trajectory generated with a learned policy is shown in Figure 6.2. The agent pulls the pendulum to one side by applying the maximum voltage. When the maximum angle is reached the agent reverses the control action to swing the pendulum to the top. Because the control action is cheap compared to the penalty on the angle, the agent adds more energy to the system than strictly needed for a swing-up. This is done to leave the expensive downwards position as quickly as possible. The additional energy is dissipated by the damping in the final stage of the swing-up after 0.7 s.

In Figure 6.1h the average learning behaviour of all the above methods is compared. As can be seen VGBP does not only learn faster with an LSTD-critic, but also finds a better, more steady policy. The quality of the learned policies of VGBP-LSTD and SARSA is similar. The final policy of MLAC is the best.

Experimental Results on the Physical Setup

Since in simulation the VGBP with the LSTD-critic performed the best, it was applied to the setup, using exactly the same settings as in simulation. Because the setup measures only the angle, the system model is used as a state observer. The action is determined with the predicted state of the model. This makes the time delay between the state measurement and action execution minimal. Figure 6.3 shows that the learning with VGBP-LSTD converges in 1.5 minutes. The results for the setup and the simulation are quite similar, with two main differences. On the setup, VGBP needs only two swings to get the pendulum upright. Although the number of swings needed to get the pendulum upright is smaller on the physical setup, the return does not increase, because there are some unmodeled dynamics in the setup. It takes slightly longer to complete a swing-up than in simulation. The other difference is that the spikes in the min-bound disappear on the setup after 30 trials, and not in the simulation results (Figure 6.1e). This is probably caused by unmodeled friction. This makes the movements a fraction smoother on the setup than in simulation. The smoother behaviour leads to less spikes in the return.

6.4.2 Robotic Manipulator

In this section VGBP learns to control a two-link manipulator to its center position. Both joints are actuated.

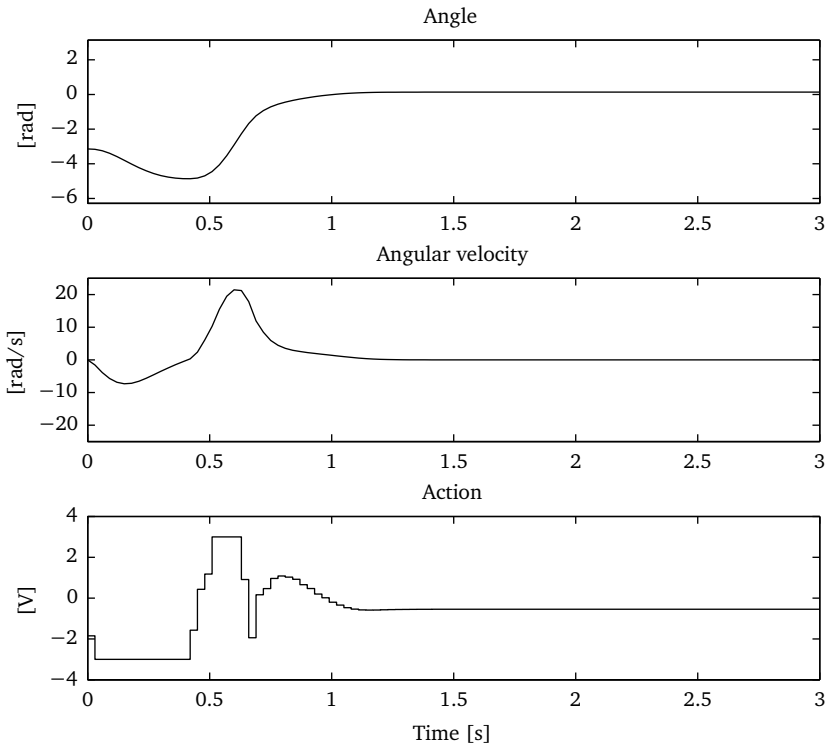


Figure 6.2 Trajectory of the swing-up of VGBP-LSTD in simulation.

Process Description

This setup is a standard 2-DOF manipulator operating in a horizontal plane, see the picture and schematic in Appendix A.2. The model parameters used in this particular case are listed in Table 6.3.

The system has four states, the two angles and two angular velocities: $x = [\varphi_1 \ \dot{\varphi}_1 \ \varphi_2 \ \dot{\varphi}_2]^\top$. The action u consists of the torques applied in the joints $u = [u_1 \ u_2]^\top$. For safety reasons, the absolute maximum action signal for both links is set to 0.1. Due to mechanical constraints, the operating range is set for both links to be -1.2 rad to 1.2 rad. Difficulties are the very high friction in the

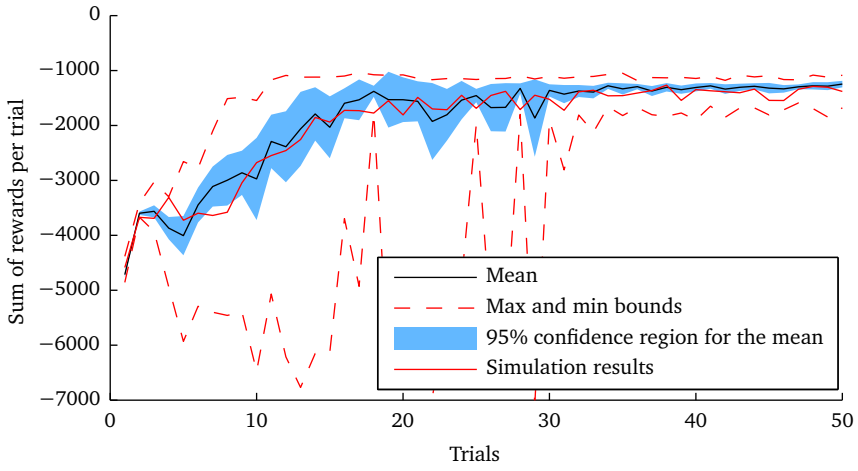


Figure 6.3 Performance of VGBP in 20 real-time experiments on the pendulum setup.

Table 6.3 Parameters of the robotic manipulator.

Model parameter	**Symbol**	**link 1**	**link 2**	**Units**
length	l	0.1	0.1	m
mass	m	0.125	0.05	kg
center of mass	m_c	0.04	0.06	m
inertia	I	0.074	$1.2 \cdot 10^4$	$\mathrm{kg\,m^2}$
RL parameter	**Symbol**	**Value**		**Units**
sampling time	T_s	0.01		s
discount factor	γ	0.97		
exploration noise	σ	0.05		
action space	U	[-0.1,0.1]	[-0.1,0.1]	N m

links and the coupling between the states. For this task the quadratic reward function in Equation (6.20) is used with

$$Q = \begin{bmatrix} 200 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 50 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad P = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

In case the manipulator is steered outside the operating range a reward of $-4 \cdot 10^4$ is given and the trial is terminated. This value has been chosen as an extremely large penalty to discourage the agent from leaving the operating range. The sampling time is set to 0.05 s. The learning procedure consists of sequences of four trials with the following initial states:

$$S_0 = \left\{ \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ -1 \\ 0 \end{bmatrix} \right\}$$

These states are the farthest from the goal state. In this manner an estimate is made for the lower bound of the return when starting from a random initial state. Each trial lasts for at most 10 seconds.

Learning Parameters

The value function approximator employs RBFs with their centers placed on an equidistant grid over the state space. This grid is defined for both links over -1.2 rad to 1.2 rad and for the angular velocities over -0.8 rad/s to 0.8 rad/s. The number of BFs along each dimension are [5 3 5 3], giving a total of 225 BFs. The B matrix is chosen in the same way as in Section 6.4.1. The critic is updated with LSTD(λ) with the trace decay factor $\lambda = 0.3$. The parameter vector θ is updated after the completion of a trial. This proved to give the highest learning speed, since information over a larger part of the state space is taken into account. The complete set of learning parameters are listed in Table 6.4.

Simulation and Experimental Results

Figure 6.5 shows the average learning curve computed from 20 learning experiments. A typical trajectory is shown in Figure 6.4.

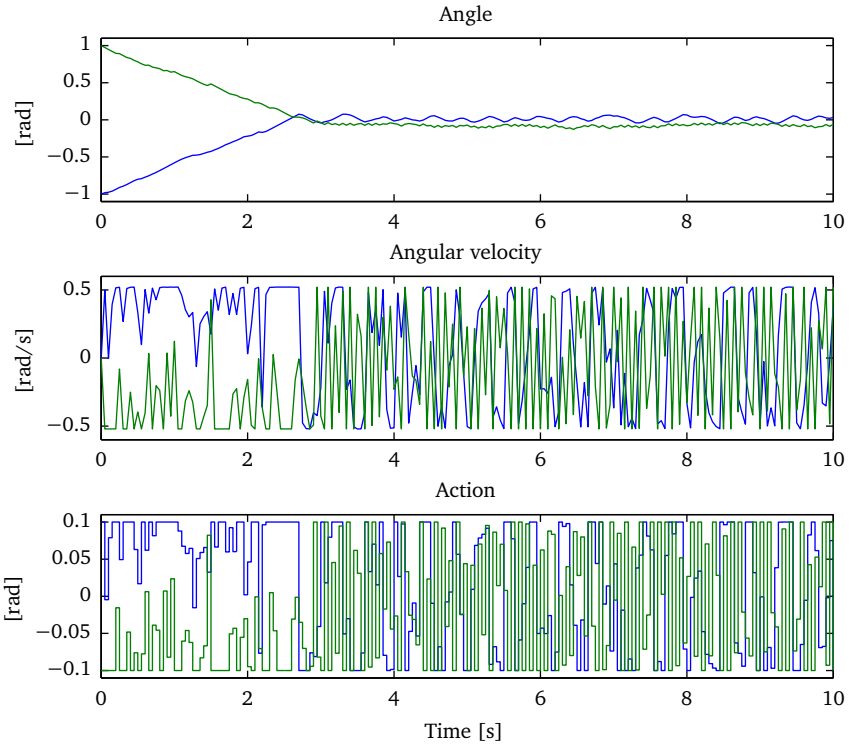


Figure 6.4 Typical experimental result learned on the manipulator. The solid blue line represents the first link and the green curve represents the second link.

Table 6.4 RL parameters of VGBP for the robotic manipulator.

Process Model		
trace decay factor	λ	0
memory size	N^P	5000
nearest neighbors	KP	80
input scaling	$\text{diag}(W)$	$\begin{bmatrix}1 & 0.05 & 1 & 0.05 & 5 & 5\end{bmatrix}$
Critic		
trace decay factor	λ	0.3
basis functions	N_{BF}	$\begin{bmatrix}5 & 3 & 5 & 3\end{bmatrix}$
angles	φ_1; φ_2	$[-1.2, 1.2]$; $[-1.2, 1.2]$ rad
angular velocities	$\dot{\varphi}_1$; $\dot{\varphi}_2$	$[-0.8, 0.8]$; $[-0.8, 0.8]$ rad/s
initial A-matrix	A_0	0.5I

The algorithm is able to steer the arm close to its center position. Due to the very high friction and the long sampling time the assumption made in Equation (6.9) that the value function is linear with respect to the action is violated near the center position. This causes a slight overshoot at that position.

The low minimum bound in Figure 6.5 is the result of VGBP occasionally steering the arm out of the feasible region. This is caused by the optimistic initialisation. Through exploratory actions the agent reaches unexplored parts of the state space. Because unseen states have a higher value than the known states, the agent leaves its known area and heads towards unexplored (terminal) states. This can be prevented by giving the known terminal states a very low initial value. The experiment shows that VGBP can be easily applied to multidimensional learning problems, finding a proper policy in a short amount of time.

6.5 Discussion

VGBP learns the value function and uses its derivative in the action selection. For concave reward functions the action is found by solving an optimisation problem. In tasks where extreme actions are required for large parts of the state space, like with the robotic manipulator, this proves to be very efficient, because the maximum action is taken when the gradient is steep and points

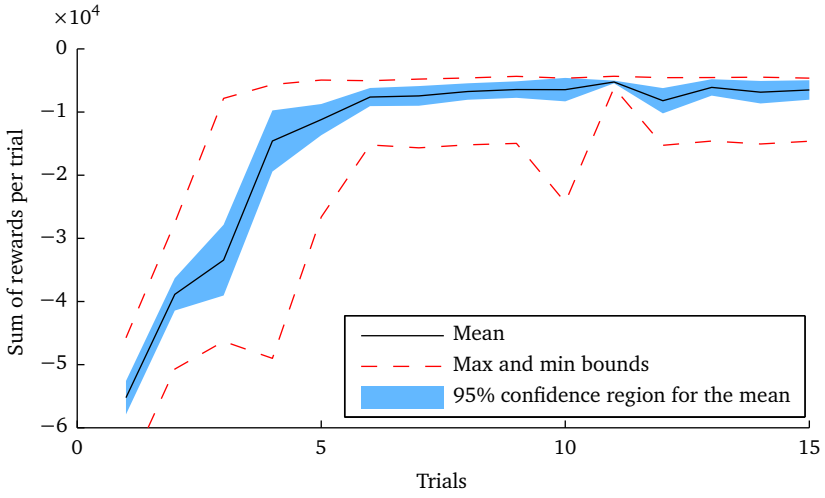


Figure 6.5 Results of VGBP for the 2-DOF manipulator (average of 20 simulations).

approximately in the right direction. With an optimistic initialisation this results in rapid exploration of the state space, leading to quick learning. A disadvantage is that the variance in the value function estimates is amplified in the gradient and therefore in the action selection. This resulted in a less stable final policy in simulations of the inverted pendulum swing-up; the agent needed two or three swings to get the pendulum upright. Other algorithms needed the same number of swings each time.

Furthermore, updating the critic at the end of a trial yielded better performance than updating the critic at each sample, which can result in overfitting and consequently in suboptimal policies. Other algorithms that use the gradient of the value function are MLAC and HDP methods. These methods suffer less from the variance in the value function because its influence is reduced by the actor update rule.

This chapter introduced a learning-rate-free version of VGBP. This method selects its actions via a direct optimisation over the right-hand side of the Bellman equation. In contrast to standard RL methods, the reward function must be available to the agent. By using the reward function along with a learned process model, the agent can predict the result of its actions. It has

been shown that the access to the reward function enables VGBP to learn faster than an actor-critic algorithm which also uses a model to obtain the same value gradient. The quality of the policy is dependent on the critic and process model approximation. Furthermore, results show that the framework works for several critic approximators, both parametric and nonparametric. VGBP-LSTD achieved the highest learning speed, both in simulations and in experiments with physical systems. In addition, no learning rates need to be tuned. In this chapter, LLR is used as a function approximator for the process model. It is interesting to investigate if the model can also be used to train the critic like in the Dyna algorithms. The two learning tasks in this chapter have a quadratic reward function. Further research is needed to find out if the good learning properties also hold for problems with more complex reward structures. As addressed in Section 6.4.2, the algorithm has a tendency to explore unseen states, including bad terminal states. It is worth investigating how the value function could be optimally initialised using knowledge of the reward function. This allows safe implementation of VGBP in learning tasks on mechanical systems with physical limitations.

Chapter

7

Conclusions and Recommendations

This thesis presented several algorithms to increase the learning speed of actor-critic reinforcement learning algorithms, by means of model learning and the explicit use of a known reward function. This chapter summarises the main conclusions of the thesis and provides possible directions for future research.

Although this thesis mainly revolves around actor-critic reinforcement learning, it is by no means a statement that actor-critic methods are always preferred. When applying reinforcement learning to a certain problem, knowing a priori whether a critic-only, actor-only or actor-critic algorithm will yield the best control policy is virtually impossible. Nevertheless, a few guidelines towards choosing an appropriate method can be provided. If, for example, it is necessary for the control policy to produce actions in a continuous space, critic-only algorithms are no longer an option, as calculating a control law would require solving a possibly non-convex optimisation problem over a continuous action space. Conversely, when the controller only needs to generate actions in a (small) countable, finite space, it makes sense to use critic-only methods, as these algorithms can solve RL problems by enumeration. Using a critic-only method also overcomes the problem of high-variance gradients in actor-only methods and the introduction of more tuning parameters (such as extra learning rates) in actor-critic methods.

Choosing between actor-only and actor-critic methods is more straight-

forward. If the problem is modeled by a (quasi-)stationary Markov decision process (MDP), actor-critic methods should provide policy gradients with lower variance than actor-only methods. Actor-only methods are however more resilient to fast changing non-stationary environments, in which a critic would be incapable of keeping up with the time-varying nature of the process and would not provide useful information to the actor, cancelling the advantages of using actor-critic algorithms. In summary, actor-critic algorithms are most sensibly used in a (quasi-)stationary setting with a continuous state and action space.

Once the choice for actor-critic has been made, there is the issue of choosing the right function approximator features for the actor and critic. Several actor-critic algorithms use the exact same set of features for both the actor and the critic, while the policy gradient theorem indicates that it is best to first choose a parameterisation for the actor, after which *compatible features* for the critic can be derived. In this sense, the use of compatible features is beneficial as it *lessens* the burden of choosing a separate parameterisation for the value function. Adding state-dependent features to the value function on top of the compatible features remains an important task as this is the only way to further reduce the variance in the policy gradient estimates. How to choose these additional features remains a difficult problem.

Even when making use of compatible features, choosing a good parameterisation for the policy in the first place still remains an important issue as it highly influences the performance after learning. Choosing this parameterisation does seem less difficult than for the value function, as in practice it is easier to get an idea what shape the policy has than the corresponding value function.

7.1 Conclusions

One of the conditions for successful application of reinforcement learning in practical applications is that learning should be quick, i.e. the number of trials needed should be limited. In the introduction of this thesis it was mentioned that classical model-based control techniques usually rely on a model of the system being available, which means a lot of time is spent on deriving such a model and introducing modelling errors along the way, which get even worse after linearisation. As a first general conclusion, it is reasonable to state that

only learning controllers can really overcome the issue of modelling errors. The time which is normally spent on deriving and linearising models now has to go into finding a reward function that describes the control task properly and also allows for actually learning that task quickly. As such, using a learning controller does not necessarily mean a suitable controller will be found quicker. Moreover, even with a perfect reward function, tuning for optimal learning and/or function approximation parameters can take a lot of time. Still, even when a system model is relatively easy to derive, learning control definitely has its perks as it could, for example, help in dealing with modelling errors, improving the usually linear control law given by classical control theory by introducing non-linearity into it. Therefore, it remains important to find ways of learning (to improve) a policy in a quick and reliable fashion.

This thesis presented several results that show how online learning of a model may improve learning speed in Chapters 3, 4 and 5. Additionally, the influence of using different function approximators was discussed. The algorithm developed in Chapter 6 tries to improve the learning speed even more by making explicit use of full knowledge of the reward function.

The most important conclusions of the research presented in this thesis are:

- The model learning actor-critic (MLAC) and reference model actor-critic (RMAC) algorithms have shown good performance in terms of both learning speed and performance of the final policy. However, these methods only really show their power when used in combination with local linear regression. When using radial basis functions, the MLAC and RMAC method do not perform better than the standard actor-critic (SAC) algorithm. It is the combination of both local linear regression (LLR) and these new methods that will provide quick, stable and good learning. The advantage of LLR over radial basis functions (RBFs) is that no initialisation is needed, eliminating the possibility of choosing an initial value for the samples that turns out to have an adverse effect on the learning. This resulted in policies being learnt in less than 50 percent of the time it takes when using RBFs in the same algorithm, although the quality of the policies in terms of smoothness may be quite different. This can be attributed to the choice of the reward function.

  In Chapter 3, the LLR implementations of SAC and MLAC obtain a much better performance at the end of the learning experiment than their RBF counterparts. RMAC was the only exception to this. This reinforces the

statement made in the previous point that the use of a non-parametric function approximator like LLR can also cause faster and better learning.

- Model learning can also be succesfully supplied to problems which are set in the finite horizon cost setting. The use of a learned process model significantly enhances the quality of the obtained policy and in the case of finite horizon MLAC (MLAC-FH) also the learning speed.

- The experiments on the two-link manipulator in Chapter 5 showed that the increase in learning speed when using model learning and/or LLR as a function approximator is even more dramatic than in the inverted pendulum case of the preceding chapters. Policies were learnt in less than 20 percent of the time it takes when using the standard actor-critic algorithm with RBFs as the function approximator. The quality of the policies were quite different, though, which again can be attributed to the choice of the reward function.

- Despite the nice advantages of having a learned model available, the process model (MLAC/RMAC) and reference model (RMAC) will suffer strongly from the curse of dimensionality, especially when RBFs are used as the function approximator. For the two-link manipulator problem, using a model learning algorithm in combination with RBFs turned out to be practically infeasible because of the increased simulation time. This is likely to be caused by both the increased computational intensity of the problem (i.e. the number of function evaluations to perform) as well as the increased complexity of the model that has to be learnt.

- A crucial condition for the model learning algorithms to work is that the input saturation of a system should be dealt with when learning a process model. Simply ignoring the input bounds will not produce a well performing policy. This can be overcome by either setting the process model's gradient with respect to the input to zero at the saturation bounds, such that an actor update cannot push the actor's output outside the saturation bounds (as done while using RBFs), or by making sure that the actor's output is manually saturated (as done while using LLR).

- Providing access to the reward function enables the value-gradient based policy (VGBP) algorithm to learn faster than an actor-critic algorithm which also uses a model to obtain the same value gradient, but only uses the values of the instantaneous rewards it receives. VGBP using the least-squares temporal difference (LSTD) achieved the highest learning

speed, both in simulations and in experiments with physical systems. In addition, no learning rates need to be tuned when using the LSTD implementation. Note that this does not necessarily mean that VGBP(-LSTD) also provides the best *policy*.

7.2 Directions for Future Research

While carrying out the research for this thesis, a number of open issues that relate to model learning, the algorithms presented in this thesis, or even reinforcement learning in general were identified. These are grouped accordingly and listed below.

7.2.1 Reinforcement Learning

- Tuning a controller, regardless of the method used, can be a daunting task. Although a learning controller alleviates the task of having to model a system, the number of parameters involved in a learning controller (number of basis functions, the parameters that can be set for the basis functions themselves, learning rates, discount rates for rewards and eligibility traces, etc.) does not lessen the burden of tuning at all. A lot can be gained if there would be a more intelligent, systematic way of tuning these parameters, rather than searching over a grid of parameter values.

- Complementing the previous point, the quality of an RL algorithm should not only be measured by looking at the performance of the policy it produces and/or the speed at which it does so, but also by checking its robustness, i.e. a sensitivity analysis should be carried out on the learning parameters of the algorithm to see how well it keeps performing when deviating from the set of optimally tuned parameters.

- Most learning controllers derive a (static) state-feedback policy. Little or no work is done on learning dynamic policies, even though dynamic controllers are amongst the most popular ones in the model-based control design world.

- Although this thesis focused on gradient-based algorithms and how to estimate this gradient, it should be noted that it is not only the quality

of the gradient estimate that influences the speed of learning. Balancing the exploration and exploitation of a policy and choosing good learning rate schedules also have a large effect on this, although more recently expectation-maximisation (EM) methods that work without learning rates have been proposed (Kober and Peters, 2011; Vlassis et al., 2009).

- The learning tasks in this thesis all have a quadratic reward function. Further research is needed to find out if the good learning properties also hold for problems with more complex reward structures.
- Given that explicit knowledge of the reward function is usually available, it is worth investigating how this could help in choosing a better initialisation of the value function.

7.2.2 Model Learning

- The experiments now only compare the performance of MLAC and RMAC with a standard actor-critic algorithm. It is worthwhile investigating how these methods compare to direct policy search methods and natural gradient based techniques.
- RMAC did not perform as well as MLAC in most of the experiments. A likely reason for this is that the convex hull approximation of the reachable subset $\mathcal{X}_R$ may not be sufficiently accurate for more complex control tasks. A more reliable calculation of reachable states is a main improvement that could be made to RMAC.
- As MLAC and RMAC are built on the same principle of building a process model, one could try and combine the quick learning of RMAC/LLR seen in Chapter 3 with the good performance of MLAC/LLR, by starting with RMAC and switching to MLAC when the performance does not significantly increase anymore.

7.2.3 Function Approximation

- While experimenting with parametric and non-parametric function approximators, no tests were done using a mix of function approximators, such as RBFs for the actor and critic and LLR for the process model, which might yield even more powerful results.

- This thesis has shown that LLR is very promising for use in fast learning algorithms, but a few issues prevent it from being used to its full potential. The first issue is how to choose the correct input weighting (including the unit scaling of the inputs), which has a large influence on selecting the most relevant samples for use in the least-squares regression.
- A second LLR related issue that should be investigated more closely is memory management: different ways of scoring samples in terms of age and redundancy and thus deciding when to remove certain samples from the memory will influence the accuracy of the estimates generated by LLR and hence the learning speed of the algorithms it is used in.
- A benefit of the memory-based function approximators, which is not exploited in this thesis, is that they can easily be initialised with samples obtained from other experiments. This makes it easy to incorporate prior knowledge on the process dynamics, a near optimal control policy or near optimal behaviour. For example, (partial) knowledge of the process could be incorporated in the process model's memory and the reference model memory in RMAC could be initialised with samples of desired closed-loop behaviour. This can be beneficial if the desired behaviour of the system is known but the control policy is yet unknown, which is often the case when supplying prior knowledge by imitation (Peters and Schaal, 2008a). In both algorithms, the process model can be initialised with input/state/output samples of the open loop system. This has the benefit that it is usually easy to conduct experiments that will generate these samples and that it is unnecessary to derive an analytical model of the system, as the samples are used to calculate locally linear models that are accurate enough in their neighbourhood.

Appendix

A

Experimental Setups

This appendix describes the two experimental setups used in this thesis. The first experimental setup is the inverted pendulum, used in Chapters 3 and 4. The second setup is a two-link manipulator, used in Chapters 5 and 6.

A.1 Inverted Pendulum

The inverted pendulum consists of an electric motor, which actuates a (weightless) link with length l. At the end of the link, a (point) mass M is attached. A picture of this system is shown in Figure A.1.

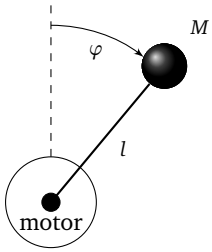


Figure A.1 The inverted pendulum setup.

The equation of motion of this system is

$$J\ddot{\varphi} \quad = \quad Mgl\sin(\varphi) - \left(b + \tfrac{K^2}{R}\right)\dot{\varphi} + \tfrac{K}{R}u$$

where φ is the angle of the pendulum measured from the upright position. The model parameters are given in Table A.1.

Table A.1 Inverted pendulum model parameters

Model parameter	Symbol	Value	Units
Pendulum inertia	J	$1.91 \cdot 10^{-4}$	$\mathrm{kg\,m^2}$
Pendulum mass	M	$5.50 \cdot 10^{-2}$	kg
Gravity	g	9.81	$\mathrm{m/s^2}$
Pendulum length	l	$4.20 \cdot 10^{-2}$	m
Damping	b	$3 \cdot 10^{-6}$	N m s/rad
Torque constant	K	$5.36 \cdot 10^{-2}$	N m/A
Rotor resistance	R	9.50	Ω

A.2 Two-Link Manipulator

The two-link manipulator is shown in Figure A.2. Both links have different lengths, masses and moments of inertia (see Table A.2). The first link has end points q_0 and q_1, where q_0 is held fixed. The second link has end points q_1 and q_2, where the revolute joint at q_1 is an electric motor. Another electric motor is at q_0. The system has no internal damping.

The equation of motion of this system is

$$M(\varphi)\ddot{\varphi} + C(\varphi,\dot{\varphi})\dot{\varphi} + G(\varphi) = u$$

with $\varphi = \left[\varphi_1\,\varphi_2\right]^\top$, $u = \left[u_1\,u_2\right]^\top$ and matrices

$$M(\varphi) = \begin{bmatrix} P_1 + P_2 + 2P_3\cos\varphi_2 & P_2 + P_3\cos\varphi_2 \\ P_2 + P_3\cos\varphi_2 & P_2 \end{bmatrix}$$

$$C(\varphi,\dot{\varphi}) = \begin{bmatrix} -P_3\dot{\varphi}_2\sin\varphi_2 & -P_3(\dot{\varphi}_1 + \dot{\varphi}_2)\sin\varphi_2 \\ P_3\dot{\varphi}_1\sin\varphi_2 & 0 \end{bmatrix}$$

$$G(\varphi) = \begin{bmatrix} gl_1m_1\sin\varphi_1 + gl_1m_2\sin\varphi_1 \\ gl_2m_2\sin\varphi_2 \end{bmatrix}$$

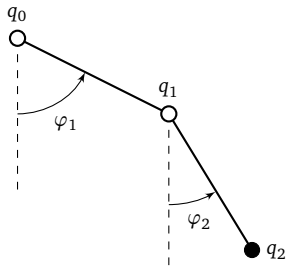


Figure A.2 The two-link manipulator setup.

Table A.2 Parameters of the robotic manipulator used in Chapter 5.

Model parameter	**Symbol**	**link 1**	**link 2**	**Units**
Length	l	0.3825	0.1	m
Mass	m	0.809	0.852	kg
Center of mass	c	0.07	0.2450	m
Inertia	I	0.1449	0.1635	$\mathrm{kg\,m^2}$
Gravity	g	9.81		$\mathrm{m/s^2}$

where $P_1 = m_1 c_1^2 + m_2 l_1^2 + I_1$, $P_2 = m_2 c_2^2 + I_2$ and $P_3 = m_2 l_1 c_2$.

References

Adam, S., L. Buşoniu and R. Babuška (2011). Experience Replay for Real-Time Reinforcement Learning Control. *IEEE Transactions on Systems, Man and Cybernetics—Part C: Applications and Reviews* 42:2, pp. 201–212.

Aleksandrov, V. M., V. I. Sysoyev and V. V. Shemeneva (1968). Stochastic Optimization. *Engineering Cybernetics* 5, pp. 11–16.

Amari, S. I. (1998). Natural Gradient Works Efficiently in Learning. *Neural Computation* 10:2, pp. 251–276.

Amari, S. I. and S. C. Douglas (1998). Why Natural Gradient? *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. Seattle, USA, pp. 1213–1216.

Arruda, R. L. S. de and F. J. Von Zuben (2011). A Neural Architecture to Address Reinforcement Learning Problems. *Proceedings of International Joint Conference on Neural Networks*. San Jose, California, USA, pp. 2930–2935.

Atkeson, C. G. and S. Schaal (1997). Robot Learning From Demonstration. *Proceedings of the 14th International Conference on Machine Learning*. Nashville, Tennessee, USA, pp. 12–20.

Bagnell, J. A. and J. Schneider (2003a). Covariant Policy Search. *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Acapulco, Mexico, pp. 1019–1024.

Bagnell, J. A. and J. Schneider (2003b). *Policy Search in Kernel Hilbert Space*. Tech. rep. 80. Carnegie Mellon University.

Baird, L. (1995). Residual Algorithms: Reinforcement Learning with Function Approximation. *Proceedings of the 12th International Conference on Machine Learning*. Tahoe City, USA, pp. 30–37.

Baird, L. and A. Moore (1999). Gradient Descent for General Reinforcement Learning. *Advances in Neural Information Processing Systems 11*. MIT Press.

Barto, A. G., R. S. Sutton and C. W. Anderson (1983). Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man, and Cybernetics* 13:5, pp. 834–846.

Bäuerle, N. and U. Rieder (2011). *Markov Decision Processes with Applications to Finance*. Universitext. Springer-Verlag Berlin Heidelberg.

Baxter, J. and P. L. Bartlett (2001). Infinite-Horizon Policy-Gradient Estimation. *Journal of Artificial Intelligence Research* 15, pp. 319–350.

Benbrahim, H., J. Doleac, J. A. Franklin and O. Selfridge (1992). Real-Time Learning: a Ball on a Beam. *Proceedings of the International Joint Conference on Neural Networks*. Baltimore, USA, pp. 98–103.

Benbrahim, H. and J. A. Franklin (1997). Biped Dynamic Walking Using Reinforcement Learning. *Robotics and Autonomous Systems* 22:3, pp. 283–302.

Bentley, J. L. and J. H. Friedman (1979). Data Structures for Range Searching. *ACM Computing Surveys (CSUR)* 11:4, pp. 397–409.

Berenji, H. R. and D. Vengerov (2003). A Convergent Actor-Critic-Based FRL Algorithm with Application to Power Management of Wireless Transmitters. *IEEE Transactions on Fuzzy Systems* 11:4, pp. 478–485.

Bertsekas, D. P. (2007). *Dynamic Programming and Optimal Control*. 3rd ed. Vol. 2. Athena Scientific.

Bertsekas, D. P. and J. N. Tsitsiklis (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bhatnagar, S. (2010). An Actor-Critic Algorithm with Function Approximation for Discounted Cost Constrained Markov Decision Processes. *Systems & Control Letters* 59:12, pp. 760–766.

Bhatnagar, S. and M. S. Abdulla (2008). Simulation-Based Optimization Algorithms for Finite-Horizon Markov Decision Processes. *Simulation* 84:12, pp. 577–600.

Bhatnagar, S., R. S. Sutton, M. Ghavamzadeh and M. Lee (2008). Incremental Natural Actor-Critic Algorithms. *Advances in Neural Information Processing Systems 20*, pp. 105–112.

Bhatnagar, S., R. S. Sutton, M. Ghavamzadeh and M. Lee (2009). Natural actor-critic algorithms. *Automatica* 45:11, pp. 2471–2482.

Borkar, V. S. (1997). Stochastic Approximation with Two Time Scales. *Systems & Control Letters* 29:5, pp. 291–294.

Borkar, V. S. (2001). A Sensitivity Formula for Risk-Sensitive Cost and the Actor-Critic Algorithm. *Systems & Control Letters* 44:5, pp. 339–346.

Boyan, J. A. (2002). Technical Update: Least-Squares Temporal Difference Learning. *Machine Learning* 49:2-3, pp. 233–246.

Bradtke, S. J. and A. G. Barto (1996). Linear Least-Squares Algorithms for Temporal Difference Learning. *Machine Learning* 22:1-3, pp. 33–57.

Bradtke, S. J., B. E. Ydstie and A. G. Barto (1994). Adaptive Linear Quadratic Control Using Policy Iteration. *Proceedings of the American Control Conference*. Baltimore, Maryland, USA, pp. 3475–3479.

Buşoniu, L., R. Babuška, B. De Schutter and D. Ernst (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Automation and Control Engineering Series. CRC Press.

Cheng, T., F. L. Lewis and M. Abu-Khalaf (2007). Fixed-Final-Time-Constrained Optimal Control of Nonlinear Systems Using Neural Network HJB Approach. *IEEE Transactions on Neural Networks* 18:6, pp. 1725–1737.

Cheng, Y. H., J. Q. Yi and D. B. Zhao (2004). Application of Actor-Critic Learning to Adaptive State Space Construction. *Proceedings of the Third International Conference on Machine Learning and Cybernetics*. Shanghai, China, pp. 2985–2990.

Deisenroth, M. P. and C. E. Rasmussen (2011). PILCO: A Model-Based and Data-Efficient Approach to Policy Search. *Proceedings of the 28th International Conference on Machine Learning*. Bellevue, Washington, USA, pp. 465–472.

Doya, K. (2000). Reinforcement Learning in Continuous Time and Space. *Neural Computation* 12:1, pp. 219–245.

Dyer, P. and S. R. McReynolds (1970). *The Computation and Theory of Optimal Control*. Ed. by R. Bellman. Vol. 65. Mathematics in Science and Engineering. Academic Press, Inc.

El-Fakdi, A., M. Carreras and E. Galceran (2010). Two Steps Natural Actor Critic Learning for Underwater Cable Tracking. *Proceedings of the IEEE International Conference on Robotics and Automation*. Anchorage, Alaska, USA, pp. 2267–2272.

Forbes, J. and D. Andre (2002). Representations for Learning Control Policies. *Proceedings of the ICML-2002 Workshop on Development of Representations*. Sydney, Australia, pp. 7–14.

Franklin, G. F., J. D. Powell and A. Emami-Naeini (2002). *Feedback Control of Dynamic Systems*. 4th ed. Athens, Greece: Prentice Hall.

Furmston, T. and D. Barber (2011). Lagrange Dual Decomposition for Finite Horizon Markov Decision Processes. *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases*. Athens, Greece: Springer-Verlag, pp. 487–502.

Gabel, T. and M. Riedmiller (2005). CBR for State Value Function Approximation in Reinforcement Learning. *Proceedings of the 6th International Conference on Case-Based Reasoning*. Chicago, Illinois, USA: Springer Berlin Heidelberg, pp. 206–221.

Girgin, S. and P. Preux (2008). Basis Expansion in Natural Actor Critic Methods. *Lecture Notes in Artificial Intelligence 5323*. Springer-Verlag Berlin Heidelberg, pp. 110–123.

Glynn, P. W. (1987). Likelihood Ratio Gradient Estimation: An Overview. *Proceedings of the 1987 Winter Simulation Conference*. Atlanta, Georgia, USA: ACM Press, pp. 366–375.

Gordon, G. J. (1995). Stable Function Approximation in Dynamic Programming. *Proceedings of the 12th International Conference on Machine Learning*. Tahoe City, USA, pp. 261–268.

Gosavi, A. (2009). Reinforcement Learning: A Tutorial Survey and Recent Advances. *INFORMS Journal on Computing* 21:2, pp. 178–192.

Gosavi, A. (2010). Finite Horizon Markov Control with One-Step Variance Penalties. *Proceedings of the 48th Annual Allerton Conference on Communication, Control, and Computing*. Allerton, Illinois, USA, pp. 1355–1359.

Gullapalli, V. (1990). A Stochastic Reinforcement Learning Algorithm for Learning Real-Valued Functions. *Neural Networks* 3:6, pp. 671–692.

Gullapalli, V. (1993). Learning Control Under Extreme Uncertainty. *Advances in Neural Information Processing Systems 5*. Ed. by S. J. Hanson, J. D. Cowan and C. L. Giles. Morgan Kaufmann Publishers, pp. 327–334.

Hanselmann, T., L. Noakes and A. Zaknich (2007). Continuous-Time Adaptive Critics. *IEEE Transactions on Neural Networks* 18:3, pp. 631–647.

Hasdorff, L. (1976). *Gradient Optimization and Nonlinear Control*. New York: John Wiley & Sons, Inc.

Jacobson, D. H. and D. Q. Mayne (1970). *Differential Dynamic Programming*. Vol. 24. Modern Analytic and Computational Methods in Science and Mathematics. New York: American Elsevier Publishing Company, Inc.

Kaelbling, L. P., M. L. Littman and A. W. Moore (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4, pp. 237–285.

Kakade, S. (2001). A Natural Policy Gradient. *Advances in Neural Information Processing Systems 14*. MIT Press, pp. 1531–1538.

Kersting, K. and K. Driessens (2008). Non-Parametric Policy Gradients: A Unified Treatment of Propositional and Relational Domains. *Proceedings of the 25th International Conference on Machine Learning*. Helsinki, Finland, pp. 456–463.

Khansari-Zadeh, S. M. and A. Billard (2011). Learning Stable Nonlinear Dynamical Systems with Gaussian Mixture Models. *IEEE Transactions on Robotics* 27:5, pp. 943–957.

Kim, B., J. Park, S. Park and S. Kang (2010). Impedance Learning for Robotic Contact Tasks Using Natural Actor-Critic Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics* 40:2, pp. 433–443.

Kimura, H. (2008). Natural Gradient Actor-Critic Algorithms Using Random Rectangular Coarse Coding. *Proceedings of the SICE Annual Conference*. Chofu, Japan, pp. 2027–2034.

Kimura, H., T. Yamashita and S. Kobayashi (2001). Reinforcement Learning of Walking Behavior for a Four-Legged Robot. *Proceedings of the 40th IEEE Conference on Decision and Control*. Orlando, Florida, USA, pp. 411–416.

Kober, J. and J. Peters (2011). Policy search for motor primitives in robotics. *Machine Learning* 84:1-2, pp. 171–203.

Konda, V. R. and J. N. Tsitsiklis (2003). On Actor-Critic Algorithms. *SIAM Journal on Control and Optimization* 42:4, pp. 1143–1166.

Konda, V. R. and V. S. Borkar (1999). Actor-Critic–Type Learning Algorithms for Markov Decision Processes. *SIAM Journal on Control and Optimization* 38:1, pp. 94–123.

Kuvayev, L. and R. S. Sutton (1996). Model-Based Reinforcement Learning with an Approximate, Learned Model. *Proceedings of the 9th Yale Workshop on Adaptive and Learning Systems*, pp. 101–105.

Lazaric, A., M. Ghavamzadeh and R. Munos (2010). Finite-Sample Analysis of LSTD. *Proceedings of the 27th International Conference on Machine Learning*. Haifa, Israel, pp. 615–622.

Lee, C. C. (1990). Fuzzy Logic in Control Systems: Fuzzy Logic Controller — Part I. *IEEE Transactions on Systems, Man, and Cybernetics* 20:2, pp. 404–418.

Lewis, F. L. and D. Vrabie (2009). Reinforcement learning and adaptive dynamic programming for feedback control. *IEEE Circuits and Systems Magazine* 9:3, pp. 32–50.

Li, C. G., M. Wang, Z. G. Sun, Z. F. Zhang and F. Y. Lin (2009). Urban Traffic Signal Learning Control Using Fuzzy Actor-Critic Methods. *Proceedings of the Fifth International Conference on Natural Computation*. Tianjin, China, pp. 368–372.

Li, C. G., M. Wang and Q. N. Yuan (2008). A Multi-Agent Reinforcement Learning Using Actor-Critic Methods. *Proceedings of the Seventh International Conference on Machine Learning and Cybernetics*. Kunming, China, pp. 878–882.

Lin, L. J. (1992). Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning* 8, pp. 293–321.

Maei, H. R., C. Szepesvári, S. Bhatnagar and R. S. Sutton (2010). Toward Off-Policy Learning Control with Function Approximation. *Proceedings of the 27th International Conference on Machine Learning*. Haifa, Israel: Rowman & Littlefield Pub Inc.

Melo, F. S. and M. Lopes (2008). Fitted Natural Actor-Critic: A New Algorithm for Continuous State-Action MDPs. *Proceedings of the European conference on Machine Learning and Knowledge Discovery in Databases*. Antwerp, Belgium, pp. 66–81.

Melo, F. S., S. P. Meyn and M. I. Ribeiro (2008). An Analysis of Reinforcement Learning with Function Approximation. *Proceedings of the 25th International Conference on Machine Learning*. Helsinki, Finland, pp. 664–671.

Moore, A. W. and C. G. Atkeson (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time. *Machine Learning* 13, pp. 103–130.

Morimura, T., E. Uchibe, J. Yoshimoto and K. Doya (2008). A New Natural Policy Gradient by Stationary Distribution Metric. *Lecture Notes in Artificial Intelligence 5212*. Ed. by W. Daelemans, B. Goethals and K. Morik. Springer-Verlag Berlin Heidelberg, pp. 82–97.

Morimura, T., E. Uchibe, J. Yoshimoto and K. Doya (2009). A Generalized Natural Actor-Critic Algorithm. *Advances in Neural Information Processing Systems 22*. MIT Press, pp. 1312–1320.

Morimura, T., E. Uchibe, J. Yoshimoto, J. Peters and K. Doya (2010). Derivatives of Logarithmic Stationary Distributions for Policy Gradient Reinforcement Learning. *Neural Computation* 22:2, pp. 342–376.

Nakamura, Y., T. Mori, M.-a. Sato and S. Ishii (2007). Reinforcement Learning for a Biped Robot Based on a CPG-Actor-Critic Method. *Neural Networks* 20, pp. 723–735.

Neumann, G. and J. Peters (2009). Fitted Q-Iteration by Advantage Weighted Regression. *Advances in Neural Information Processing Systems 22 (NIPS 2008)*.

Ng, A. Y., A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger and E. Liang (2006). Autonomous Inverted Helicopter Flight via Reinforcement Learning. Experimental Robotics IX, pp. 363–372.

Niedzwiedz, C., I. Elhanany, Z. Liu and S. Livingston (2008). A Consolidated Actor-Critic Model with Function Approximation for High-Dimensional POMDPs. *AAAI 2008 Workshop for Advancement in POMDP Solvers*. Chicago, Illinois, USA, pp. 37–42.

Park, J., J. Kim and D. Kang (2005). An RLS-Based Natural Actor-Critic Algorithm for Locomotion of a Two-Linked Robot Arm. *Lecture Notes on Artificial Intelligence 3801*. Springer-Verlag Berlin Heidelberg, pp. 65–72.

Park, Y. M., M. S. Choi and K. Y. Lee (1996). An Optimal Tracking Neuro-Controller for Nonlinear Dynamic Systems. *IEEE Transactions on Neural Networks* 7:5.

Paschalidis, I. C., K. Li and R. M. Estanjini (2009). An Actor-Critic Method Using Least Squares Temporal Difference Learning. *Proceedings of the Joint 48th IEEE Congress on Decision and Control and 28th Chinese Control Conference*. Shanghai, China, pp. 2564–2569.

Pazis, J. and M. G. Lagoudakis (2011). Reinforcement Learning in Multidimensional Continuous Action Spaces. *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pp. 97–104.

Pennesi, P. and I. C. Paschalidis (2010). A Distributed Actor-Critic Algorithm and Applications to Mobile Sensor Network Coordination Problems. *IEEE Transactions on Automatic Control* 55:2, pp. 492–497.

Peters, J., K. Mülling and Y. Altün (2010). Relative Entropy Policy Search. *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. Atlanta, Georgia, USA, pp. 1607–1612.

Peters, J. and S. Schaal (2008a). Natural Actor-Critic. *Neurocomputing* 71, pp. 1180–1190.

Peters, J. and S. Schaal (2008b). Reinforcement Learning of Motor Skills with Policy Gradients. *Neural Networks* 21:4, pp. 682–697.

Peters, J., S. Vijayakumar and S. Schaal (2003). Reinforcement Learning for Humanoid Robotics. *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots*. Karlsruhe, Germany.

Powell, W. B. (2010). *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. 2nd ed. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc.

Raju, C., Y. Narahari and K. Ravikumar (2003). Reinforcement Learning Applications in Dynamic Pricing of Retail Markets. *IEEE International Conference on E-Commerce*. Newport Beach, California, USA, pp. 339–346.

Richter, S., D. Aberdeen and J. Yu (2007). Natural Actor-Critic for Road Traffic Optimisation. *Advances in Neural Information Processing Systems 19*. Ed. by B. Schölkopf, J. Platt and T. Hoffman. MIT Press, pp. 1169–1176.

Riedmiller, M., J. Peters and S. Schaal (2007). Evaluation of Policy Gradient Methods and Variants on the Cart-Pole Benchmark. *Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*. Honolulu, USA, pp. 254–261.

Robbins, H. and S. Monro (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics* 22:3, pp. 400–407.

Rummery, G. A. and M. Niranjan (1994). *On-Line Q-Learning Using Connectionist Systems*. Tech. rep. CUED/F-INFENG/TR 166. Cambridge University.

Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* 3:3, pp. 211–229.

Schoknecht, R. (2003). Optimality of Reinforcement Learning Algorithms with Linear Function Approximation. *Advances in Neural Information Processing Systems 15*. MIT Press, pp. 1555–1562.

Si, J., A. Barto, W. Powell and D. Wunsch (2004). *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press.

Skogestad, S. and I. Postlethwaite (2008). *Multivariable Feedback Control - Analysis and Design*. 2nd ed. John Wiley & Sons, Inc.

Spall, J. C. (1992). Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation. *IEEE Transactions on Automatic Control* 37:3, pp. 332–341.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning* 3, pp. 9–44.

Sutton, R. S. (1990). Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. *Proceedings of the Seventh International Conference on Machine Learning*. San Mateo, California, USA, pp. 216–224.

Sutton, R. S. (1992). Reinforcement Learning Architectures. *Proceedings of the International Symposium on Neural Information Processing*, pp. 211–216.

Sutton, R. S. and A. G. Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Sutton, R. S., H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári and E. Wiewiora (2009). Fast Gradient-Descent Methods for Temporal-Difference Learning with Linear Function Approximation. *Proceedings of the 26th Annual International Conference on Machine Learning*. Montreal, Canada: ACM.

Sutton, R. S., D. McAllester, S. Singh and Y. Mansour (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems 12*. MIT Press, pp. 1057–1063.

Szepesvári, C. (2010). Algorithms for Reinforcement Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers.

Tsitsiklis, J. N. and B. Van Roy (1996). Feature-Based Methods for Large Scale Dynamic Programming. *Recent Advances in Reinforcement Learning*. Ed. by L. P. Kaelbling. Kluwer Academic Publishers, pp. 59–94.

Tsitsiklis, J. N. and B. Van Roy (1997). An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control* 42:5, pp. 674–690.

Tsitsiklis, J. N. and B. Van Roy (1999). Average Cost Temporal-Difference Learning. *Automatica* 35:11, pp. 1799–1808.

Usaha, W. and J. A. Barria (2007). Reinforcement Learning for Resource Allocation in LEO Satellite Networks. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics* 37:3, pp. 515–527.

Vamvoudakis, K. G. and F. L. Lewis (2010). Online Actor-Critic Algorithm to Solve the Continuous-Time Infinite Horizon Optimal Control Problem. *Automatica* 46:5, pp. 878–888.

Vengerov, D., N. Bambos and H. R. Berenji (2005). A Fuzzy Reinforcement Learning Approach to Power Control in Wireless Transmitters. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics* 35:4, pp. 768–778.

Vlassis, N., M. Toussaint, G. Kontes and S. Piperidis (2009). Learning Model-Free Robot Control by a Monte Carlo EM Algorithm. *Autonomous Robots* 27:2, pp. 123–130.

Wang, D., D. Liu and H. Li (2012). Finite-Horizon Neural Optimal Tracking Control for a Class of Nonlinear Systems with Unknown Dynamics. *Proceedings of the 10th World Congress on Intelligent Control and Automation*. Beijing, China: IEEE, pp. 138–143.

Wang, X. S., Y. H. Cheng and J. Q. Yi (2007). A Fuzzy Actor-Critic Reinforcement Learning Network. *Information Sciences* 177, pp. 3764–3781.

Watkins, C. J. C. H. and P. Dayan (1992). Q-Learning. *Machine Learning* 8, pp. 279–292.

Watkins, C. J. C. H. (1989). Learning from Delayed Rewards. PhD thesis. King's College, University of Cambridge.

Wawrzyński, P. (2009). Real-Time Reinforcement Learning by Sequential Actor–Critics and Experience Replay. *Neural Networks* 22:10, pp. 1484–1497.

Wettschereck, D., D. W. Aha and T. Mohri (1997). A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms. *Artificial Intelligence Review* 11, pp. 273–314.

Williams, J. L., J. W. Fisher III and A. S. Willsky (2006). Importance Sampling Actor-Critic Algorithms. *Proceedings of the 2006 American Control Conference*. Minneapolis, Minnesota, USA: IEEE, pp. 1625–1630.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning* 8, pp. 229–256.

Wilson, D. R. and T. R. Martinez (2000). Reduction Techniques for Instance-Based Learning Algorithms. *Machine Learning* 38, pp. 257–286.

Witten, I. H. (1977). An Adaptive Optimal Controller for Discrete-Time Markov Environments. *Information and Control* 34, pp. 286–295.

Glossary

The glossary consists of a summary of the mathematical symbols and abbreviations used throughout this thesis.

List of symbols and notations

Standard math symbols

$\mathbb{R}$	set of real numbers
E	expectation
P	probability
μ	mean
σ	standard deviation
s	sample standard deviation
$\mathcal{N}(\mu, \sigma)$	normal distribution
$\sim$	draw from distribution
$x^\top$	transpose of x
$\|x\|$	norm of x
H	Hessian
$\nabla_x f$ or $\frac{\partial f}{\partial x}$	Gradient of function f with respect to x
$\widetilde{\nabla}_x f$	Natural gradient of function f with respect to x
$G(\vartheta)$	Riemannian metric tensor with respect to ϑ
F	Fisher information matrix
L	Lagrangian
μ	Lagrangian multiplier

MDPs and RL

X	state space
U	action space
x	state
u	action
r	instantaneous reward
f	transition function
ρ	reward function
τ_N	terminal reward function
k	discrete time
N	number of time steps in finite horizon
γ	reward discount factor
z	eligibility trace
λ	eligibility trace discount factor
π	policy
V	state value function
Q	state-action value function
A	advantage function
J	cost
d	state distribution
n	size of state vector
Δ	exploration
α	learning rate
δ	temporal difference (error)

Approximate RL

ψ	policy feature vector
ϑ	policy parameter vector
p	size of policy parameter vector
ϕ	value function feature vector
θ	value function parameter vector
q	size of value parameter vector
ψ	compatible feature vector
w	compatible feature parameter
M	LLR memory
s_i	sample i in LLR memory

$\mathcal{K}(x)$	set of indices i in LLR memory defining the neighbourhood of x.

MLAC / RMAC

$\widehat{f}$	process model
ζ	process model parameter vector
r	number of process model parameters per element of output
R	reference model
η	reference model parameter vector
s	number of reference model parameters per element of output
$\mathcal{R}$	reachable set of states

List of abbreviations

The abbreviations used throughout the thesis are listed here in lexicographical order.

ACSMDP	actor-critic semi-Markov decision algorithm
ACFRL	actor-critic fuzzy reinforcement learning
BF	basis function
CACM	Consolidated Actor-Critic Model
DOF	degrees of freedom
EM	expectation-maximization
eNAC	episodic natural actor-critic
FACRLN	Fuzzy Actor-Critic Reinforcement Learning Network
FIM	Fisher information matrix
gNAC	generalized natural actor-critic
gNG	generalized natural gradient
HDP	Heuristic Dynamic Programming
IPA	infinitesimal perturbation analysis
LLR	local linear regression
LQG	linear quadratic Gaussian
LQR	linear quadratic regulator
LSTD	least-squares temporal difference
MDP	Markov decision process
MLAC	model learning actor-critic

NAC	natural actor-critic
OPI	optimistic policy iteration
PID	proportional-integral-derivative
RBF	radial basis function
REINFORCE	REward Increment = Non-negative Factor × Offset Reinforcement × Characteristic Eligibility
REPS	relative entropy policy search
RL	reinforcement learning
RMAC	reference model actor-critic
SAC	standard actor-critic
SARSA	state-action-reward-state-action
SPSA	simultaneous perturbation stochastic approximation
TD	temporal difference
VGBP	value-gradient based policy

Publications by the Author

Journal Papers

Depraetere, B., M. Liu, G. Pinte, I. Grondman and R. Babuška (2014). Comparison of Model-Free and Model-Based Methods for Time Optimal Hit Control of a Badminton Robot. *Mechatronics* 24:8, pp. 1021–1030.

Grondman, I., L. Buşoniu, G. A. Lopes and R. Babuška (2012a). A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients. *IEEE Transactions on Systems, Man and Cybernetics—Part C: Applications and Reviews* 42:6, pp. 1291–1307.

Grondman, I., M. Vaandrager, L. Buşoniu, R. Babuška and E. Schuitema (2012b). Efficient Model Learning Methods for Actor-Critic Control. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics* 42:3, pp. 591–602.

Rooijen, J. C. van, I. Grondman and R. Babuška (2014). Learning Rate Free Reinforcement Learning for Real-Time Motion Control Using a Value-Gradient Based Policy. *Mechatronics* 24:8, pp. 966–974.

Conference Papers

Grondman, I., L. Buşoniu and R. Babuška (2012). Model Learning Actor–Critic Algorithms: Performance Evaluation in a Motion Control Task. *Proceedings*

of the 51st IEEE Conference on Decision and Control. Maui, Hawaii, USA, pp. 5272–5277.

Grondman, I., M. Vaandrager, L. Buşoniu, R. Babuška and E. Schuitema (2011). Actor-Critic Control with Reference Model Learning. *Proceedings of the IFAC 18th World Congress*. Milan, Italy, pp. 14723–14728.

Grondman, I., H. Xu, S. Jagannathan and R. Babuška (2013). Solutions to Finite Horizon Cost Problems Using Actor-Critic Reinforcement Learning. *Proceedings of the 2013 International Joint Conference on Neural Networks*. Dallas, Texas, USA, pp. 325–331.

Liu, M., B. Depraetere, G. Pinte, I. Grondman and R. Babuška (2013). Model-Free and Model-Based Time-Optimal Control of a Badminton Robot. *Proceedings of the 9th Asian Control Conference*. Istanbul, Turkey.

Summary

Classical control theory requires a model to be derived for a system, before any control design can take place. This can be a hard, time-consuming process if the system is complex. Moreover, there is no way of escaping modelling errors. As an alternative approach, there is the possibility of having the system *learn* a controller by itself while it is in operation or offline. Reinforcement learning (RL) is such a framework in which an agent (or controller) optimises its behaviour by interacting with its environment[1]. After taking an action in some state, the agent receives a scalar reward from the environment, which gives the agent an indication of the quality of that action. The function that indicates the action to take in a certain state is called the policy. The main goal of the agent is to find a policy that maximises the total accumulated reward during the course of interaction, also called the return. By following a given policy and processing the rewards, the agent can build estimates of the return (the value function) and use this information to improve its policy.

For continuous state and action spaces, the use of function approximators is a necessity and a commonly used type of RL algorithms for these continuous spaces is the actor-critic algorithm, in which two independent function approximators take the role of the policy (the actor) and the value function (the critic).

A main challenge in RL is to use the information gathered during the interaction as efficiently as possible, such that an optimal policy may be reached in a short amount of time. The majority of RL algorithms at each time step measure the state, choose an action corresponding to this state, measure the next state, the corresponding reward and update a value function

[1] In this thesis, the system to be controlled is seen as the environment of the controller.

(and possibly a separate policy). As such, the only source of information used for learning at each time step is the last transition sample. Algorithms that store these transitions separately and reuse them to enlarge the efficiency of the algorithm, e.g. the "experience replay" algorithms and Sutton's Dyna algorithm, have been around for a while, but even more efficiency in learning can be achieved by

- Searching for a relation between the collected transition samples and trying to predict the system's behaviour from this by interpolation and/or extrapolation.
- Incorporating prior knowledge, e.g. about the system or the desired closed-loop behaviour.
- Using the reward function that can often explicitly be made available to the learning agent.

This thesis describes and explores the use of the above principles, keeping the objective of improving the learning speed in mind.

Online model learning for RL

Two new model learning actor-critic learning algorithms are introduced in this thesis: model learning actor-critic (MLAC) and reference model actor-critic (RMAC). Both have in common that they learn a full-state model of the system to be controlled, which is then used to make one-step predictions about the states the system will end up in if a certain input is applied. If available, the function approximator used for this process model can be pre-trained with prior knowledge about the system.

Model learning actor-critic

Many standard reinforcement learning algorithms are inefficient in their use of measured data. Once a transition sample—containing the previous state, the action taken, the subsequent state and the reward—has been used to update the actor and critic, it is thrown away and never reused in future updates. To overcome this problem, several techniques have been proposed to remember and reuse measured data, such as experience replay and prioritised sweeping. A drawback of these methods is that they require storage of all the samples

gathered, making them memory intensive and computationally heavy. Dyna architectures combine reinforcement learning with the concept of planning, by learning a model of the process or environment online and using this model to generate experiences from which the critic (and thus the actor) can be updated. This results in more frequent updates and hence quicker learning.

In MLAC, the learned process model is not used to generate experiences. Instead, the process model is used directly in the policy gradient, aiming to get faster convergence of learning without increasing the number of updates for the actor and/or critic.

Having a learned process model available simplifies the update of the actor, as it allows to predict what the next state of the system will be, given some applied input. The value function then provides information on the value of that next state. Hence, calculating the optimal input to the system reduces to an optimization problem, in which the objective function is the composition of the process model and the value function and the decision variable is the applied input or action.

Reference model actor-critic

RMAC is different from the typical actor–critic methods in the sense that it does not learn an explicit mapping from state to action. Instead of an explicit actor/policy, RMAC learns a reference model that represents a desired behaviour of the system, based on the value function. Similar to MLAC, this algorithm learns a process model to facilitate one-step predictions about the system. The difference with respect to MLAC is that the explicit actor is now replaced by a composition of the learned reference model with the inverse of the learned process model to calculate an action.

A reference model provides a means for the storage of demonstration data, if it is available. Some learning algorithms benefit from having the desired behaviour or task demonstrated to them. This can be done, for example, by a human manually moving a robot arm in such a way that a target task is performed. The demonstrated trajectory is then stored as a sequence of (sampled) states and it is exactly this type of information that can be stored in a reference model.

The applicability of the model learning actor-critic algorithms has been verified with simulation experiments on an inverted pendulum and a two-link

manipulator. For the inverted pendulum, experiments have been carried out in both an infinite and finite horizon setting. Results of these experiments show that the use of local linear regression as a function approximator and/or the addition of model learning can reduce the time of learning an acceptable control policy to about 20 to 50 per cent of the time it takes with a completely model-free actor-critic algorithm using a parametric function approximator.

Using reward function knowledge

Another way of making learning more efficient, is to make the reward function directly accessible to the learning agent. Classic reinforcement learning theory assumes that the reward function is part of the agent's environment and therefore unknown. The learning agent only gathers information about rewards on a per-sample basis. For quite a lot of problems and especially the problems addressed in this thesis, though, the reward function is usually designed by an engineer. Hence, an explicit expression representing the reward function is available and as such can directly be used by the learning agent.

The final algorithm presented in this thesis makes use of the explicit knowledge of the reward function and also learns a process model online. This enables the algorithm to select control actions by optimizing over the right-hand side of the Bellman equation. Fast learning convergence in simulations and experiments with the underactuated pendulum swing-up task is demonstrated and additionally, experimental results for a more complex 2-DOF robotic manipulator task are presented. It was shown that access to the reward function enables the Value-Gradient Based Policy (VGBP) algorithm to learn faster than an actor-critic algorithm which also uses a model to obtain the same value gradient. Nevertheless, the quality of the learned policy is dependent on the critic and process model approximation and may still be outperformed by other algorithms.

Samenvatting

Klassieke regeltheorie vereist het afleiden van een model voor een systeem, voordat enig ontwerp van een regelaar kan plaatsvinden. Dit kan een moeilijk, tijdrovend proces zijn als het een complex systeem betreft. Bovendien is er niet te ontsnappen aan modelleringsfouten. Als alternatieve aanpak is er de mogelijkheid om het systeem zelf een regelaar te laten *leren* terwijl het in bedrijf is (online) of niet (offline). Reinforcement learning (RL) is zo'n raamwerk waarin een agent (of regelaar) zijn gedrag optimaliseert door middel van interactie met zijn omgeving[2]. Na het nemen van een actie, ontvangt de agent een scalaire beloning van de omgeving, die een indicatie geeft van de kwaliteit van die actie. De functie die bepaalt welke actie er genomen wordt in een bepaalde toestand wordt de strategie genoemd. Het hoofddoel van de agent is om een strategie te vinden die de totale opbrengst van geaccumuleerde beloningen tijdens de interactie maximaliseert. Door een gegeven strategie te volgen en de verkregen beloningen te verwerken, kan de agent schattingen maken van deze opbrengst (de waardefunctie) en deze gebruiken om zijn strategie te verbeteren.

Voor continue toestands- en actieruimten is het gebruik van functiebenaderingen een noodzaak en een veelgebruikt type RL algoritme voor deze continue ruimtes is het acteur-criticus algoritme, waarin twee onafhankelijke functiebenaderingen de rol op zich nemen van de strategie (de acteur) en de waardefunctie (de criticus).

Een grote uitdaging in RL is om de informatie die tijdens de interactie is verkregen zo efficiënt mogelijk te gebruiken, zodat een optimale strategie binnen een korte tijdsduur bereikt kan worden. De meerderheid van RL

[2] In dit proefschrift wordt het te regelen systeem gezien als de omgeving van de regelaar.

algoritmen meet op elk tijdsinterval de toestand van het systeem, kiest een actie die hoort bij deze toestand, meet de volgende toestand, de bijbehorende beloning en updatet daarmee de waardefuntie (en mogelijk een afzonderlijke strategie). Daardoor is de enige bron van informatie die gebruikt wordt voor het leren bij elk tijdsinterval de laatste overgangssample. Algoritmen welke al deze overgangen afzonderlijk opslaan en hergebruiken om de efficiëntie van het algoritme te vergroten, zoals de "experience replay" algoritmen en Sutton's Dyna algoritme, bestaan al een tijd, maar meer efficiëntie in het leren kan bereikt worden door

- Te zoeken naar een relatie tussen de verzamelde overgangssamples en hiermee, door middel van interpolatie en extrapolatie, het gedrag van het systeem proberen te voorspellen.
- Voorkennis te benutten, bijvoorbeeld over het systeem zelf of over het gewenste gesloten-lus gedrag.
- De beloningsfunctie te gebruiken, die vaak expliciet beschikbaar kan worden gesteld aan de lerende agent.

Dit proefschrift beschrijft en onderzoekt het gebruik van bovenstaande principes, met als doel het verbeteren van de leersnelheid.

Online leren van een model voor RL

Twee nieuwe model lerende acteur-criticus leeralgoritmen worden in dit proefschrift geïntroduceerd: model lerend acteur-criticus (MLAC) en referentiemodel acteur-criticus (RMAC). Beide hebben gemeen dat ze een toestandsmodel van het te regelen systeem leren, welke vervolgens gebruikt wordt voor het doen van één stap voorspellingen over de toestanden waarin het systeem zal overgaan na het toepassen van een bepaalde actie. Indien beschikbaar, kan de functiebenadering van dit procesmodel vooraf getraind worden met voorkennis over het systeem.

Model Lerend Acteur-Criticus

Veel standaard reinforcement learning algoritmen zijn inefficiënt in hun gebruik van gemeten data. Als een overgangssample—bestaande uit vorige toestand, de gekozen actie, de volgende toestand en de beloning—eenmaal

is gebruikt om de acteur en de criticus te updaten, wordt deze informatie weggegooid en niet hergebruikt in toekomstige updates. Om dit probleem te verhelpen zijn verscheidene technieken voorgesteld om gemeten data te onthouden en te hergebruiken, zoals experience replay en geprioriteerd vegen. Een nadeel van deze methodes is dat ze opslag vereisen van alle overgangssamples, wat ze geheugen- en rekenintensief maakt. Dyna architecturen combineren reinforcement learning met het concept van plannen. Door een model van het proces of de omgeving te leren en hiermee ervaringen te genereren waarmee de criticus (en daarmee de acteur) geüpdatet kan worden zijn er vaker updates en dus ontstaat een sneller leergedrag.

In MLAC wordt het geleerde procesmodel niet gebruikt om ervaringen te genereren. In plaats daarvan wordt het model direct gebruikt in de berekening van de strategiegradiënt, met als doel om het leren sneller te laten convergeren zonder het aantal updates van de acteur en/of criticus te verhogen.

De beschikbaarheid van een procesmodel vereenvoudigt de update van de acteur, omdat het mogelijk maakt te voorspellen wat de volgende toestand van het systeem gaat zijn als een bepaalde actie toegepast wordt. De waardefunctie geeft vervolgens informatie over de waarde van deze volgende toestand. Derhalve wordt het berekenen van de optimale ingang voor het systeem gereduceerd tot een optimalisatieprobleem, waarin de doelfunctie de samenstelling van het procesmodel en de waardefunctie is en de beslissingsvariabele de toegepaste ingang of actie.

Referentiemodel Acteur-Criticus

RMAC verschilt van typische acteur-criticus methoden in de zin dat het geen directe afbeelding van toestand naar actie probeert te leren. In plaats van een expliciete acteur/strategie, leert RMAC een referentiemodel wat het gewenste gedrag voor een systeem weerspiegelt, gebaseerd op de waardefunctie. Net als MLAC, leert dit algoritme een procesmodel om één stap voorspellingen te faciliteren. Het verschil met MLAC is dat de expliciete acteur nu vervangen is door een samenstelling van het geleerde referentiemodel met de inverse van het procesmodel om een actie te berekenen.

Een referentiemodel voorziet in de mogelijkheid om, indien beschikbaar, demonstratiedata op te slaan. Enkele leeralgoritmen profiteren ervan als gewenst gedrag of een taak vooraf wordt gedemonstreerd. Dit kan bijvoorbeeld gedaan worden door een mens handmatig een robotarm te laten bewegen

zodat een bepaalde taak wordt uitgevoerd. Het gedemonstreerde traject wordt dan opgeslagen als een rij van opeenvolgende toestanden en het is precies dit type informatie wat kan worden opgeslagen in een referentiemodel.

De toepasbaarheid van de model lerende acteur-criticus algoritmen is geverifieerd door middel van simulaties op een omgekeerd pendulum en een robotarm met twee vrijheidsgraden. Voor het omgekeerde pendulum zijn experimenten uitgevoerd in zowel een oneindige als eindige horizon setting. De resultaten van deze experimenten tonen aan dat het gebruik van lokale lineaire regressie als functiebenadering en/of het toevoegen van het leren van een model de leertijd kan inkorten tot 20 tot 50 procent van de tijd die het kost met een volledig modelvrij acteur-criticus algoritme met een parametrische functiebenadering.

Gebruik maken van de beloningsfunctie

Een alternatief om het leren efficiënter te maken is de lerende agent direct toegang te geven tot de beloningsfunctie. Klassieke reinforcement learning gaat er vanuit dat de beloningsfunctie onderdeel is van de omgeving van de agent en daarom onbekend is. De lerende agent krijgt slechts een keer per tijdsinterval informatie over een beloning. Echter, voor veel problemen (en vooral de problemen in dit proefschrift) wordt de beloningsfunctie ontworpen door een technicus of ingenieur. Derhalve is er een expliciete expressie beschikbaar die de beloningsfunctie representeert en deze kan als zodanig gebruikt worden door de lerende agent.

Het laatste algoritme in dit proefschrift maakt gebruik van deze expliciete kennis van de beloningsfunctie en leert ook online een procesmodel. Dit stelt het algoritme in staat om regelacties te selecteren door de rechterzijde van de Bellman vergelijking te optimaliseren. Snelle convergentie van het leren wordt gedemonstreerd in zowel simulaties als met een echte opstelling. Ook worden resultaten gepresenteerd van een meer complexe taak op een echte opstelling van een robotarm met twee vrijheidsgraden. Het wordt aangetoond dat directe toegang tot de beloningsfunctie de “Value-Gradient Based Policy” (VGBP) in staat stelt sneller te leren dan een acteur-criticus algoritme dat ook een model gebruikt om tot dezelfde waardegradiënt te komen. Desalniettemin hangt de kwaliteit van de verkregen strategie af van de beanderingen van de criticus en het procesmodel en kan nog steeds overtroffen worden door andere algoritmen.

Curriculum Vitae

Ivo Grondman was born on the 6th of May, 1982 in Overdinkel, municipality of Losser, The Netherlands.

After attending secondary school at Stedelijk Lyceum Het Kottenpark in Enschede from 1994 to 2000, he started his studies in Applied Mathematics and Telematics at the University of Twente, Enschede, and obtained Bachelor of Science degrees in both disciplines in early 2007. By late 2008 he obtained his Master of Science degree with Merit in Control Systems from Imperial College London and accepted a position at MathWorks Ltd. in Cambridge, United Kingdom.

In December 2009, he commenced his PhD research on actor-critic reinforcement learning algorithms at the Delft Center for Systems and Control of Delft University of Technology in The Netherlands under the supervision of prof. dr. Robert Babuška and dr. Lucian Buşoniu. In 2011, he received his DISC certificate from the Dutch Institute of Systems and Control. At the start of 2013, he spent three months in the United States as a visiting scholar to conduct part of his research at the Missouri University of Science & Technology in Rolla, Missouri, under the supervision of prof. dr. Sarangapani Jagannathan.

During his research, he supervised several BSc and MSc students, assisted and lectured on various courses in the systems and control area and participated in a number of conferences.

ISBN 978-94-6186-432-1

TU Delft