Reinforcement Learning

on autonomous humanoid robots

Reinforcement Learning
on autonomous humanoid robots

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 maandag 12 november 2012 om 15:00 uur
door

Erik SCHUITEMA

natuurkundig ingenieur
geboren te Puttershoek

Dit proefschrift is goedgekeurd door de promotoren:

Prof.dr.ir. P.P. Jonker
Prof.dr. R. Babuška

Copromotor: Dr.ir. M. Wisse

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. P.P. Jonker,	Technische Universiteit Delft, promotor
Prof.dr. R. Babuška,	Technische Universiteit Delft, promotor
Dr.ir. M. Wisse,	Technische Universiteit Delft, copromotor
Prof.dr. K.G. Langendoen,	Technische Universiteit Delft
Prof.dr. H. Nijmeijer,	Technische Universiteit Eindhoven
Prof.dr. R.S. Sutton,	University of Alberta, Canada
Dr. K. Tuyls,	Universiteit Maastricht
Prof.dr. F.C.T. van der Helm,	Technische Universiteit Delft (reservelid)

Dit onderzoek is financieel mogelijk gemaakt door Technologiestichting STW, projectnummer DWO7462.

Copyright © 2012, E. Schuitema, Delft.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

ISBN 978-94-6186-075-0

Author email: `erik@essd.nl`

Contents

List of symbols **v**

1 Introduction **1**

1.1 Motivation 1

1.2 RL on real robots 3

1.3 Problem statement 4

1.4 Research goal 5

1.5 Approach 5

1.6 Thesis outline 6

2 Reinforcement Learning for real, autonomous robots **9**

2.1 The Markov Decision Process 9

2.1.1 Goal of the agent 10

2.1.2 Designing an MDP for a robotic task 11

2.1.3 Solving an MDP 12

2.2 Temporal Difference learning 13

2.3 Function approximation 15

2.3.1 Training function approximators 16

2.3.2 Linear function approximation 16

2.3.3 High-dimensional state-action spaces 17

2.3.4 Tile coding 18

2.3.5 Appropriate feature spaces 20

2.3.6 Storing the parameters in memory 21

2.4 Control Delay 22

2.4.1 Introduction 23

2.4.2 Control delay in MDPs 24

2.4.3 TD learning with control delay: dSARSA and dQ 26

2.4.4 Empirical evaluations 29

2.4.5 Conclusions 33
2.5 Conclusions 34

3 Reinforcement Learning on a real bipedal walking robot 35

3.1 Hardware design 35
3.1.1 Introduction 36
3.1.2 Reinforcement Learning requirements 37
3.1.3 System overview 38
3.1.4 Robustness 39
3.1.5 Reliable state information 41
3.1.6 Actuation 43
3.1.7 System invariability 45
3.1.8 Real-time control 46
3.1.9 Conclusions 48
3.2 Software design and simulation 49
3.2.1 Introduction 49
3.2.2 Software requirements 49
3.2.3 Related Work 51
3.2.4 Implementation 52
3.2.5 Results: bipedal walking robot Leo 58
3.2.6 Conclusion 63
3.3 Experimental RL results on Leo 64
3.3.1 The stairs step-up task 64
3.3.2 Hardware changes 68
3.3.3 Learning to walk 70
3.3.4 Pre-programmed limit cycle walking 76
3.3.5 Initializing the learning controller with a pre-programmed solution 79
3.3.6 Conclusions 83
3.4 Conclusions 85

4 The effects of large disturbances on learning to walk 87

4.1 Introduction 87
4.2 Large disturbances during RL 88
4.2.1 Outliers due to unexpected interactions with the environment 89
4.2.2 Outliers due to sensor noise 89
4.2.3 Outliers due to sampling time irregularities 90
4.2.4 Disturbance rejection and detection 91
4.3 Experimental setup 91
4.3.1 The simplest walker 92
4.3.2 Learning to walk 92
4.3.3 Test scenarios 93

4.4 Results and discussion . . . 94
4.4.1 Push . . . 94
4.4.2 Sensor spike noise . . . 95
4.4.3 Sampling time irregularity . . . 96
4.5 Conclusions . . . 99

5 Using Independent Learners for individual actuators 101
5.1 Introduction . . . 101
5.2 Theory . . . 103
5.2.1 Approach . . . 103
5.2.2 MMDP . . . 104
5.2.3 MA-Q(λ) and MA-SARSA(λ) . . . 104
5.2.4 Policy . . . 105
5.2.5 Problems of ILs . . . 106
5.3 Two-link manipulator . . . 109
5.3.1 Learning . . . 109
5.3.2 Results . . . 110
5.4 Meta: a bipedal walking robot . . . 115
5.4.1 Model . . . 115
5.4.2 Learning . . . 115
5.4.3 Results . . . 117
5.5 Leo . . . 121
5.5.1 Learning . . . 122
5.5.2 Results . . . 122
5.6 Discussion . . . 124
5.7 Conclusions . . . 125

6 Reducing system damage using Modular Reinforcement Learning 127
6.1 Introduction . . . 127
6.2 Related work . . . 128
6.3 Approach . . . 130
6.4 Theory . . . 131
6.4.1 MRL-SARSA . . . 132
6.4.2 Related methods . . . 133
6.5 Test setup 1: The simplest walker . . . 135
6.5.1 MRL-SARSA(λ) . . . 135
6.5.2 Related methods . . . 137
6.6 Test setup 2: Leo . . . 140
6.6.1 MRL-SARSA(λ) . . . 140
6.6.2 Alternative methods . . . 140
6.7 Conclusion . . . 144

7 Discussion, conclusions and future directions **147**
7.1 Research goal 147
7.2 Discussion and conclusions 148
7.2.1 RL techniques 148
7.2.2 Hardware and software requirements 148
7.2.3 Identification of practical complications 150
7.2.4 Reducing practical complications 152
7.3 Future directions 154
7.3.1 Autonomously learning from scratch 154
7.3.2 Algorithmic improvements 156

Appendix A Leveling the floor **159**

Appendix B Additional results **163**

Summary **177**

Samenvatting **179**

Dankwoord **183**

Curriculum Vitae **185**

List of symbols

α	learning rate
α_{Cu}	thermal resistance coefficient of copper [K^{-1}]
a	action
$\hat{a}$	effective action
A	action space
β	lenience temperature discounting factor
γ	time discounting factor
c	vertical center of mass offset [mm]
C	Coriolis and centrifugal forces matrix
δ_{TD}	temporal difference error
d	dimension
ϵ	exploration rate
e	eligibility trace
E_{gb}	gearbox efficiency
g	gravitational acceleration
G	gearbox ratio
θ	temperature [K]
$\boldsymbol{\theta}$	feature parameter vector
h	sampling period [s]
I	moment of inertia [kgm^2]
$I_{\tau_{\mathrm{d}}}$	augmented state space
κ	lenience parameter
k	time step (index)
K	number of tilings
K_{r}	resolution scaling factor
K_{τ}	torque constant of a DC motor [NmA^{-1}]
λ	(eligibility) trace discounting factor
l	length [m]
m	mass [kg]

M	mass matrix
N_{Br}	magnetic flux density reduction factor [$\%\mathrm{K}^{-1}$]
π	(control) policy
π^*	optimal policy
P	proportional gain
Q	action-value function
Q^π	action-value function under policy π
Q^*	optimal action-value function
$\hat{Q}$	estimated action-value function
r	reward
r_{d}	edge length of a hypercube for dimension d
R	reward function
R	winding resistance [Ω]
$\mathfrak{R}$	return
σ	slope angle
s	state
$\hat{s}$	predicted state
S	state space
τ	torque [Nm]
τ_{d}	relative control delay
τ_ℓ	lenience temperature
τ_{task}	characteristic time constant of the learning task
τ_{elig}	characteristic time constant of the eligibility of the agent's actions
T	state transition probability density function
T_{d}	absolute control delay [s]
U	motor voltage [V]
V	value function
V^π	value function under policy π
V^*	optimal value function
$\boldsymbol{\phi}$	basis function vector / feature vector
φ	joint angle [rad]
$\dot{\varphi}$, ω	joint velocity [$\mathrm{rad\ s}^{-1}$]
$\ddot{\varphi}$	joint acceleration [$\mathrm{rad\ s}^{-2}$]
Ψ	data set with state-action pairs
w	horizontal center of mass offset [mm]
W	electrical work
x	random variable

Chapter 1

Introduction

1.1 Motivation

Service robots that learn

Service robots have the potential to be of great value in labour intensive environments such as domestic, medical and construction environments. Market analysis (IFR Statistical Department, 2010) has shown a large expected growth in the number of service robots worldwide. By the end of 2009, in total 76,600 professional service robots and 8.7 million personal service robots were sold. For the period 2010-2013, 80,000 new professional service robots are expected to be installed (mainly for defence, milking, cleaning, construction, rescue and security applications, field robots, logistic systems, inspection robots, medical robots and mobile robot platforms for multiple use), while 11.4 million personal service robots (domestic robots for vacuum cleaning, lawn-mowing, window cleaning and other types, and entertainment and leisure robots such as toy robots) are expected to be sold. Medical robots will even become a necessity, since the ageing population in Western countries will require significantly more elderly care in the coming decades than available (WHO, 2007). While the value of factory robots has been long proven, today's commercially available service robots are still a novelty. They mainly consist of vacuum cleaners, lawn mowers and entertainment robots. These machines perform relatively simple and straightforward tasks and are as of yet incapable of versatile manipulation of their environment. A key difficulty is the large diversity in destined environments of service robots. While factory robots work in highly structured, controlled and predictable environments, every household, office or construction site is typically unique. This makes even basic tasks such as locomotion challenging. Because the variety in environments cannot be completely foreseen and tested at the robot's production time, it is hard to manually program robots to perform motor control tasks such as locomotion and object manipulation in a way that is robust against these environmental variations. Furthermore, the aforementioned environments are likely to continuously change by the introduction of novel products and objects that the service robot needs to interact with. This requires service robots to be versatile and able to perform emerging tasks. To let robots function autonomously in such unstructured, highly diverse and renewing environments, having robots *learn* motor control tasks autonomously from interaction with their environment forms an attractive alternative to being manually programmed by experts.

Reinforcement Learning

There are several existing approaches to letting robots learn motor control tasks. In learning from demonstration (LfD), a system learns from solutions demonstrated by an expert (Argall et al., 2009), e.g., by teleoperation or direct manipulation of the system. With imitation learning (IL) (Schaal, IJspeert, and Billard, 2003), a solution is shown on a different physical platform, and the robot needs to translate this solution to its own hardware. LfD and IL are examples of supervised learning; a solution is available and needs to be transferred to the system via learning. An alternative method that does not require a demonstrated solution is learning from experience (LfE), in which the system learns purely from interaction with the environment in a trial-and-error fashion, which is the field of *Reinforcement Learning* (RL) (Sutton and Barto, 1998; Bertsekas, 2007). These three categories of methods – LfD, IL and LfE – can also be combined, e.g., an initial solution from a human demonstrator is learned with imitation learning, after which the solution is further optimized by learning from experience (Schaal et al., 2005). Because an initial solution provided by an expert is seldom immediately satisfactory and sufficiently generic, learning from interaction with the actual environment is a necessary final step.

RL systems have the ability to learn directly from interaction with the environment by receiving feedback on their behavior in the form of *rewards*: good behavior is reinforced by positive rewards and bad behavior is discouraged by negative rewards. A solution, however, is not provided. The rewards indicate *what* is desired, but *not how* to achieve it. Therefore, RL is largely unsupervised. Both model-based and model-free RL techniques exist, where the latter do not require a model of the system and its environment, which can be hard or even impossible to obtain for service robot tasks. These properties – learning from interaction in a largely unsupervised way, without the need for a model – make that RL offers great opportunities for service robots to learn to operate in diverse and changing environments.

Because of its general formulation, RL has been successfully applied to a wide set of problems ranging from games (Tesauro, 1995), economics (Moody and Saffell, 2001) and traffic control (Salkham et al., 2008) to the control of elevators (Crites and Barto, 1996), helicopters (Ng et al., 2004) and soccer robots (Stone and Sutton, 2001; Kohl and Stone, 2004).

1.2 RL on real robots

Applications of RL to robotic motor control tasks in which the data was collected from trials on the real robot (as opposed to simulation) have been demonstrated in a limited number of cases. Kalmár, Szepesvári, and Lörincz (1998) applied RL to let a mobile robot learn to fetch and reposition a ball by learning to invoke pre-programmed, small subtasks. Peters, Vijayakumar, and Schaal (2003) used RL to optimize a parameterized point-to-point movement trajectory on a robot arm with one degree of freedom (DoF) and Peters and Schaal (2006); Peters and Schaal (2008) used RL to learn a 7 DoF humanoid robot arm to hit a baseball by learning the parameters of inherently stable point attractor controllers (called motor primitives) for each DoF. In a similar manner, Kober and Peters (2009) and Nemec, Zorko, and Zlajpah (2010) demonstrated learning the ball-in-a-cup and ball-paddling tasks on a humanoid robot arm. Smart and Kaelbling (2000) used RL to learn a corridor following task in a mobile robot by learning to steer while the forward velocity was controlled to be constant. Sutton et al. (2011) used RL to learn a light-seeking policy and several sensor value maximizing policies on their wheeled prototype Critterbot, which was specifically designed for RL experiments. Ito, Takayama, and Kobayashi (2009) applied RL to learn a light-seeking task on a snake-like robot. Furthermore, RL has been employed for gait synthesis and optimization on bipedal walking robots (Benbrahim, 1996; Salatian, Yi, and Zheng, 1997; Tedrake, Zhang, and Seung, 2004; Ogino et al., 2004; Morimoto et al., 2005; Cherubini et al., 2009) and quadruped robots (Kohl and Stone, 2004; Kamio and Iba, 2005). Morimoto and Doya (2001) applied a hierarchical RL controller for learning a stand up behavior on a three-link robot, where the upper control layer learned to select target angles for the lower layer, which learned to achieve those target angles by selecting motor torques that were added to a standard servo control rule. In the field of robot soccer, Riedmiller et al. (2009) demonstrated a wheeled soccer robot learning a dribble task from scratch.

The number of robots that were capable of performing their learning updates while executing their task, i.e., *in real-time*, is only in the order of 10 worldwide (Benbrahim, 1996; Salatian, Yi, and Zheng, 1997; Kalmár, Szepesvári, and Lörincz, 1998; Morimoto and Doya, 2001; Tedrake, Zhang, and Seung, 2004; Ogino et al., 2004; Kamio and Iba, 2005; Morimoto et al., 2005; Ito, Takayama, and Kobayashi, 2009; Sutton et al., 2011). In the majority of cases, learning was performed at a coarse time scale, for example at every footstep or at every macro action (e.g., 'go forward') taking 1-10s, with the exceptions of Sutton et al. (2011) (their robot learned at 2-10Hz) and Morimoto and Doya (2001) (the lower level of their robot's control hierarchy learned at 100Hz). In most of the aforementioned cases, the learning controller had a pre-programmed structure (such as the motor primitives used by Peters et al.) or was augmented with a pre-programmed controller that aided the solution. Such approaches require prior knowledge of the task. Computing the learning updates was typically done on a computer on the side line instead of on the robot's embedded computer, with the exception of

Tedrake, Zhang, and Seung (2004). Learning on embedded computing hardware, which poses additional technical challenges, is ultimately necessary to ensure the autonomy and mobility of service robots. To keep service robots agile, their mass, volume and power consumption are preferably kept low. This constrains the computational power that they can embed.

1.3 Problem statement

From the literature, we can observe that little is known about applying RL to learning low-level motor control tasks in real-time, on embedded robot hardware. To the best of our knowledge, none of the above robots were capable of doing this. The lack of wide scale application of RL to robots has several known reasons. The most important theoretical difficulty is the inability of current RL algorithms to solve problems with large state-action spaces (the space spanned by all states and all control actions) in reasonable time. In this respect, the state space of as task for a robot with more than 3 degrees of freedom can generally be considered large for RL. Most of the aforementioned successful RL demonstrations were shown on robots with few degrees of freedom, or reformulated the learning problem to one with a state space with low dimensionality. The most important practical difficulty is related to the trial-and-error nature of RL. Performing occasional random actions are essential for RL to improve upon the task solution in a largely unsupervised way. Such explorative actions can lead to control signals and system states that quickly wear down or directly damage the robot. In practice, this limits the time that learning can be performed on a real robot. Generally speaking, it appears that robotic hardware is often simply not suitable for RL. It has led to the common strategy under researchers to limit the solution space in a way that both speeds up learning (there are fewer solutions to try out) and prevents damage to the system, e.g., by only allowing a class of parameterized, usually locally stable control functions of which the parameters are optimized by RL. This was done in nearly all of the aforementioned applications of RL to real robots. Creating such parameterized policy functions, however, requires expert knowledge on the robot and on the task at hand. While knowledge on the robot is available at production time, knowledge on the environment and exemplary task solutions is typically lacking in our envisioned setting of service robots solving a variety of (emerging) tasks in diverse environments.

In summary, the problems in applying RL to real service robots are as follows:

1. There is limited knowledge on applying RL to learning low-level motor control tasks in real-time, on embedded robot hardware.

2. It is problematic to apply RL to robotic tasks due to the large state-action space spanned by their degrees of freedom.

3. Current RL approaches in robotic tasks typically need prior knowledge on the task.

4. Robotic hardware is typically not suited for RL's trial-and-error nature.

1.4 Research goal

The goal of this thesis is to identify and address difficulties in hardware design, software design and RL theory that currently prevent the application of RL to real, autonomous service robots. More specifically, the following research questions are being addressed:

1. What are suitable RL techniques for real-time, autonomous learning of low-level motor control tasks on a real robot without the need for prior knowledge on the task or its environment?
2. What are the hardware and software requirements for a real robot in order to be suitable for these RL techniques?
3. What are the practical complications that arise from applying these RL techniques to a real robot?
4. How can these practical complications be addressed?

1.5 Approach

To find answers to the posed research questions, this thesis starts by selecting RL techniques from the literature that are suitable for our particular purpose, based on their theoretical properties and on existing simulation results. The selected techniques must learn from experience, i.e., from interaction with the real world, require as little prior knowledge on the task or its environment as possible, and be able to run on embedded computing hardware. Preference is given to 'vanilla' techniques, i.e., techniques that exist for quite some time and are therefore relatively well understood. In this way, the work can focus on the particular complications of applying RL to real robots.

The motivating example throughout the thesis is bipedal locomotion, i.e., the task of learning to walk for a bipedal robot. Simulation results have shown that it is possible to learn this task in a matter of hours using vanilla RL techniques without pre-structuring the solution space and without the need for an expert solution (Schuitema et al., 2005). The task is both challenging and interesting to solve with RL for the following reasons. Its dynamics include aspects that are difficult to model or simulate, such as the underactuated, friction dominated degree of freedom between the foot and the ground, and the sequential alternation of the statically stable double-support phase (i.e., when both legs are in contact with the floor) and the statically unstable single-support phase. This makes it difficult to design robust conventional controllers; a practically oriented paradigm such as RL forms an attractive alternative. The task is challenging since the number of degrees of freedom for bipedal walking robots quickly exceeds 5, which results in a very large state space. The inherent instability of many walking robots emphasizes the existence of risky system states such as a fall. The walking task is interesting from an RL perspective, because the use of rewards creates the opportunity to easily have the robot learn to focus on walking speed (by rewarding forward movement and punishing time), minimal energy usage (by penalizing motor work) or a combination (Schuitema et al., 2005).

We believe that RL poses some important requirements on the hardware and software in order to be suitable for RL. However, in the literature, we could find only one robot that was designed specifically for RL (Sutton et al., 2011). Therefore, from the selected RL techniques, hardware and software requirements are derived for a real bipedal walking robot, from which a new prototype is created – bipedal walking robot '*Leo*'. The prototype will serve as a dedicated research platform for RL. Experimentation on this prototype is used to identify the particular complications of applying RL to a real robot. In the remaining work, solutions are proposed to a number of these complications, which are evaluated in simulation. To facilitate the research, a realistic simulation of the prototype is built in which solutions can be tested prior to evaluating them on the prototype.

The work in this thesis is of predominantly practical nature and focuses on the application of the proposed techniques in simulation and on real hardware. There is less focus on deriving new algorithms or providing mathematical proof of the soundness of the proposed techniques.

1.6 Thesis outline

The remainder of this thesis is structured as follows.

Chapter 2 introduces the theoretical preliminaries of RL that are used in the remainder of this thesis. It motivates the choice for Temporal Difference (TD) learning algorithms, linear function approximation and several peripheral techniques, thereby answering research question 1. Furthermore, this chapter discusses the impact of one practical complication – the existence of control delay, i.e., delay between measuring the robot's state and acting upon it – and provides solutions for specific cases.

Chapter 3 derives the hardware and software requirements for the RL techniques proposed in Chapter 2 and presents the hardware and software design of the resulting prototype 'Leo', answering research question 2. Subsequently, the experimental results are presented of applying these techniques in simulation as well as on the real prototype, ultimately demonstrating real-time TD learning on the prototype for two tasks: learning a stairs step-up and learning to walk. From these results, several practical complications are identified in response to research question 3. The main contribution of this chapter is that it demonstrates the successful application of well known RL techniques in solving a non-trivial task with large state-action space on a real robot, in real-time.

Chapter 4 studies the detrimental effects of large and infrequent disturbances on the process of learning to walk, thereby partially addressing research question 4. Simulation results of a simplified model of a walking robot show that large and infrequent deviations of the sampling period or sensor readings have a much smaller effect on the learning process than large and infrequent external disturbances such as a push.

Chapter 5 further addresses research question 4. A learning scheme is proposed in which actuators learn independently but cooperatively to accomplish the global task of learning to walk. This approach makes the proposed RL techniques more scalable in the number of actuators because it reduces memory consumption and reduces the control delay caused by algorithmic computation.

Chapter 6 proposes a method designed to reduce the risk that the robot is exposed to during learning, with the aim to reduce the hardware strain caused by the explorative nature of RL, thereby further answering research question 4. With the proposed method, the robot quickly but coarsely learns to predict the risk of its actions, which enables it to quickly learn how to avoid risky situations such as a fall and learn a solution in a safer way.

Chapter 7 presents conclusions and discussion of the work presented in this thesis and proposes possible future research directions.

Chapter 2

Reinforcement Learning for real, autonomous robots

This chapter introduces the theoretical preliminaries of Reinforcement Learning (RL) that are used in the remainder of this thesis. The focus is on techniques suitable for autonomously learning to perform various tasks on real robots. Some tasks can be performed by learning a single motor control skill, such as locomotion. Other tasks might require learning an ensemble of skills, e.g., a hierarchy, that contains both motor skills and strategy skills. The goal is to use learning techniques that can be applied to learning motor skills as well as strategy skills (although the latter are not demonstrated in this thesis). For a more general and more thorough introduction to RL, see for example (Sutton and Barto, 1998) and (Bertsekas, 2007). Section 2.1 introduces the commonly used framework for RL, the Markov Decision Process. Section 2.2 introduces temporal difference (TD) learning, including its main on-line algorithms. Section 2.3 considers function approximation of continuous high-dimensional state-action spaces, an important prerequisite for learning on robotic systems.

In addition, a contribution on the inclusion of control delay in the MDP framework, and its influence on TD-learning, is discussed in Section 2.4. Control delay is the time delay between measuring the system's state and executing the control action. It is always present in real systems.

2.1 The Markov Decision Process

The common approach in RL is to model the process of learning a task as a Markov Decision Process (MDP) with discrete time steps $k \in \mathbb{N}$ and sampling period h. The dynamic systems that we are interested in – robotic systems – have a continuous state space S and a continuous or discrete action space A. State transitions are considered to be stochastic. The MDP is defined as the 4-tuple

$\langle S, A, T, R\rangle$, where S is a set of states and A is a set of actions. The state transition probability density function $T : S \times A \times S \to [0, \infty)$ defines the probability density over S for the next state $s_{k+1} \in S$ after executing action $a_k \in A$ in state $s_k \in S$. The reward function $R : S \times A \times S \to \mathbb{R}$ is real valued and defines the reward of a state transition as $r_{k+1} = R(s_k, a_k, s_{k+1})$. A control policy (or simply policy) $\pi : S \times A \to [0, \infty)$ defines the action selection probability density for all actions in all states. An MDP has the Markov property, which means that transitions only depend on the current state-action pair and not on past state-action pairs nor on information excluded from s. This implies that s must contain all relevant state information on both the robot and its environment.

Every task – and in case of a modular architecture, every sub-task – has its own MDP definition. For a robotic system in a certain environment, multiple similar MDPs can be defined to learn different tasks. By varying R, it is possible to optimize towards different goals within the same task environment. Changes in the dynamics of the robot, e.g., due to an added load, usually cause changes in T, which results in a new MDP. Furthermore, when different elements of the robot and the environment are relevant for different tasks, e.g., when interaction with different objects takes place, these tasks require different definitions of S. Finally, A can be chosen differently between tasks, e.g., when some actuators do not need to be controlled by the learning agent in order to solve the task.

2.1.1 Goal of the agent

The goal of the learner is to find a control policy that maximizes, from every state $s \in S$, the return, i.e., the long-term sum of discounted rewards $\mathfrak{R}$:

$$\mathfrak{R}_k = \sum_{i=0}^{k_{\text{term}}} \gamma^i r_{k+i+1} \tag{2.1}$$

in which $\gamma \in [0, 1]$ is the (time) discounting factor through which future rewards are weighted equal or less than immediate rewards, and k_{term} is a final time step at which the task terminates. For episodic tasks, i.e., tasks that have a distinct start and end, k_{term} is finite. Examples of episodic robotic tasks are grasping an object or standing up after a fall. For continuing tasks (infinite horizon tasks)[1], $k_{\text{term}} = \infty$. Examples of continuing tasks are state maintaining tasks, such as balancing a bipedal robot, or periodic tasks, such as walking. Note, however, that when failure states are present, continuing tasks can contain episodes as well, especially at the beginning of the learning process. For example, if a robot is learning to walk, a fall on the floor can mark the end of the episode, after which the robot is put into an initial condition. With the definition of an absorbing state – a state that transitions only to itself and that only generates rewards of zero – we can generalize episodic and infinite horizon tasks by defining episodic tasks to end in an absorbing state and always taking $k_{\text{term}} = \infty$ in (2.1).

---

[1] For infinite horizon tasks, γ cannot equal 1; this could lead to infinite returns.

The value function $V^\pi(s)$ gives the expected return of following policy π from state s:

$$V^\pi(s) = \mathrm{E}_\pi\{\mathfrak{R}_k | s_k = s\} = \mathrm{E}_\pi\left\{\sum_{i=0}^{\infty}\gamma^i r_{k+i+1}\middle| s_k = s\right\} \tag{2.2}$$

where $\mathrm{E}_\pi\{\,\cdot\,\}$ denotes the expected value given that the agent follows policy π. The action-value function or Q-function $Q(s,a)$ gives the estimated return of choosing action a in state s and following the control policy afterwards:

$$Q^\pi(s,a) = \mathrm{E}_\pi\{\mathfrak{R}_k | s_k = s, a_k = a\} = \mathrm{E}_\pi\left\{\sum_{i=0}^{\infty}\gamma^i r_{k+i+1}\middle| s_k = s, a_k = a\right\} \tag{2.3}$$

A policy that is better than or equal to all other policies with respect to $\mathfrak{R}$ for all $s \in S$ is an optimal policy, denoted π^*. All optimal policies share the same optimal value function $V^*(s)$ and optimal action-value function $Q^*(s,a)$.

Online RL implies that the system learns from interaction with the real world. The state transition probability function T can be either unknown to the learning agent (model-free RL), learned while learning the task (model-learning RL), or provided a priori (model-based RL). In the remainder of this thesis, we focus on model-free RL.

2.1.2 Designing an MDP for a robotic task

In setting up an MDP to solve a robotic task, design choices have to be made for most of its parameters. While the state transition probability density function T is an intrinsic property of the process, the state space S, the action space A, reward function R, sampling period h and discounting factor γ generally have to be (carefully) chosen manually[2].

State space The state space S is in principle defined by its requirement to comply with the Markov property, although in practice, this requirement is hard or impossible to meet. Often, Markovian state information is not available, e.g., due to a lack of sensors or sensor precision. When the Markov property is (severely) violated, it can be more appropriate to model the process as a Partially Observable MDP or POMDP (see, e.g., (Sondik, 1978)). In addition, the Markov state signal is not unique; an equivalent Markov state signal can easily be obtained, e.g., by an isometric transformation of an existing Markov state vector. This leaves some freedom in defining the state signal.

Action space and sampling period The robot's action space A is often continuous, e.g., it consists of a set of motor voltages. Certain RL algorithms, such as the

---

[2] Ideally, an autonomously learning robot can choose these parameters automatically. However, this is not the current state of the art.

Temporal Difference algorithms discussed in 2.2, are designed for discrete action space. Therefore, from here on we assume A to be a finite set $A = \{a^1, a^2, \ldots, a^n\}$, obtained by discretizing the robot's original, continuous action space. Due to the discrete nature of both the time steps and the actions, the size of the action space $|A|$ and the sampling period h together determine the space of possible task solutions. The number N_a of distinct action sequences $(a_0, \ldots, a_{k_{\text{term}}})$ that can be performed from a starting state s_0 in the characteristic time scale τ_{task} of the task is approximately of the order

$$N_a = \mathcal{O}\left(|A|^{\frac{\tau_{\text{task}}}{h}}\right) \tag{2.4}$$

where each time step, an action is drawn from A. A lower N_a results in a smaller solution space and thus faster learning. Because N_a increases polynomially in $|A|$ but exponentially in $1/h$, it is especially beneficial to increase h. A lower limit of h is related to the actuator dynamics (the minimum time scale at which the actuator can switch between discrete actions) and the sensor resolution (in a very small period of time, sensor values might not change at all). An upper limit of h is related to the natural frequencies of the mechanical system and the desired solution space; h and $|A|$ together determine the maximum obtainable system performance. In addition, a smaller h results in a shorter response time to disturbances (disturbances in the context of RL are discussed in more depth in Chapter 4). To the best of our knowledge, no systematic approach exists to choose $|A|$ and h.

Reward function The reward function R is task dependent. Because the rewards influence the agent's exploratory behavior, its final performance and its learning speed, choosing R is often a process of trial and error. It can even be proven that multiple reward functions share the same optimal policy for a certain task (Ng, Harada, and Russell, 1999). Designing reward structures that lead to faster learning, e.g., by rewarding sub-goals (Mataric, 1994) or by reward shaping (Laud, 2004; Marthi, 2007), is a topic under research.

Discounting factor The discounting factor γ defines the time horizon of the task. Therefore, it can best be chosen to correspond with the characteristic time constant of the task (in seconds), τ_{task}:

$$\gamma = e^{-\frac{h}{\tau_{\text{task}}}} \tag{2.5}$$

2.1.3 Solving an MDP

Several RL methods exist to find a (locally) optimal policy π for an MDP, either directly by performing the optimization in policy space (direct policy search) or indirectly by making use of $V^\pi(s)$ or $Q^\pi(s, a)$. To estimate the (action-)value function, Temporal Difference (TD) learning is a widely used approach. The control policy π can be derived from the value function in different ways, either directly, such as in TD control, or indirectly, such as in actor-critic schemes, in which the policy is stored in a separate memory structure (the actor) that learns from 'critique' of the value function (the critic). To select appropriate RL methods for use in this thesis, we considered the following.

On real robots, most successes (Morimoto and Doya, 2001; Nakanishi et al., 2004; Tedrake, Zhang, and Seung, 2004; Peters, Vijayakumar, and Schaal, 2003) have been accomplished using actor-critic and policy-gradient based algorithms, in which the policy is parameterized and stored separately from the value function. Those algorithms converge to a locally optimal policy (see, e.g., (Bhatnagar et al., 2008; Peters and Schaal, 2008; Bhatnagar et al., 2009)) by performing gradient-ascent on the policy parameter vector. For non-trivial tasks, generally, a reasonable initial solution needs to be available a priori in order to converge to a useful local optimum. An initial solution might not always be available, especially for autonomous robots learning new tasks. Therefore, it is useful to consider algorithms that do not require such an initial solution to perform well, such as TD control algorithms. TD control algorithms are model-free and derive the policy directly from the estimated (action-)value function. They do not require explicit policy parameterization or an initial task solution. In addition, TD control algorithms are relatively well understood and computationally simple. Because of these potential benefits, this thesis focuses on the feasibility and practical implications of using TD control on real robots in real-time. TD learning and TD control are discussed in more detail below. For a recent overview of value function approaches as well as direct policy search approaches in robotics, see (Kober and Peters, 2012).

2.2 Temporal Difference learning

Temporal Difference (TD) learning methods have the goal to estimate $V^\pi(s)$ or $Q^\pi(s,a)$. TD methods estimate the (action-)value function at time step k, $Q_k(s,a)$, by bootstrapping from an initial estimate, using information from single state transitions. Because TD methods learn from single observed state transitions, they do not need a model. They work on-line, for both episodic tasks and infinite horizon tasks. The following recursive reformulation of $Q^\pi(s,a)$ (reformulation of V is analogous) shows the relation between $Q^\pi(s_k,a_k)$ and $Q^\pi(s_{k+1},a_{k+1})$:

$$\begin{aligned}
Q^\pi(s,a) &= \mathrm{E}_\pi\left\{ \sum_{i=0}^{\infty} \gamma^i r_{k+i+1} \middle| s_k = s, a_k = a \right\} \\
&= \mathrm{E}_\pi\left\{ r_{k+1} + \gamma \sum_{i=0}^{\infty} \gamma^i r_{k+i+2} \middle| s_k = s, a_k = a \right\} \\
&= \mathrm{E}_\pi\left\{ r_{k+1} + \gamma Q^\pi(s_{k+1}, a_{k+1}) \middle| s_k = s, a_k = a \right\}
\end{aligned}$$

This formulation can be used to derive the TD error $\delta_{\mathrm{TD},k+1}$ of the transition, which gives the difference between the current estimate $Q^{\pi}_{k}(s_k, a_k)$ and the estimate based on r_{k+1} and $Q^{\pi}_{k}(s_{k+1}, a_{k+1})$:

$$\delta_{\mathrm{TD},k+1} = r_{k+1} + \gamma Q^{\pi}_{k}(s_{k+1}, a_{k+1}) - Q^{\pi}_{k}(s_k, a_k) \tag{2.6}$$

The TD error is used to update the estimate of $Q^{\pi}_{k}(s_k, a_k)$. For discrete state-action spaces (function approximation is explained in Section 2.3), Q can be updated as follows:

$$Q^{\pi}_{k+1}(s_k, a_k) = Q^{\pi}_{k}(s_k, a_k) + \alpha \delta_{\mathrm{TD},k+1} \tag{2.7}$$

in which $\alpha \in (0, 1]$ is the learning rate or step size.

In TD *control*, the policy is directly derived from $Q(s, a)$. An important policy is the greedy policy, which selects $a_{k,\mathrm{greedy}}$, the action with the highest estimated return:

$$a_{k,\mathrm{greedy}} = \arg\max_{a'} Q^{\pi}(s_k, a') \tag{2.8}$$

While greedy actions exploit the knowledge gained and currently stored in $Q(s, a)$, new knowledge can be gained from selecting exploratory, non-greedy actions. A widely used action selection policy that includes exploratory actions is the ϵ-greedy policy $\pi_{\epsilon\text{-greedy}}(s_k, a_k)$, which is defined such that a random action is selected with probability ϵ (uniformly sampled from A) and $a_{k,\mathrm{greedy}}$ otherwise:

$$\pi_{\epsilon\text{-greedy}}(s_k, a_k) = \begin{cases} 1 - \epsilon + \epsilon/n & \text{if } a_k = a_{k,\mathrm{greedy}} \\ \epsilon/n & \text{if } a_k \neq a_{k,\mathrm{greedy}} \end{cases} \tag{2.9}$$

with $\epsilon \in [0, 1]$ the exploration rate and n the number of actions in A. For a good trade-off between exploration and exploitation, the value for ϵ is typically chosen from the range $[0.01, 0.20]$. The softmax action selection policy $\pi_{\mathrm{softmax}}(s_k, a_k)$ is also common in RL and defines a probability distribution over greedy and non-greedy actions that is continuous in $Q(s, a)$:

$$\pi_{\mathrm{softmax}}(s_k, a_k) = \frac{e^{Q(s_k, a_k)/\Theta}}{\sum_{i=1}^{n} e^{Q(s_k, a^n)/\Theta}} \tag{2.10}$$

with $\Theta > 0$ the temperature. A higher temperature leads to more exploration. Choosing an action with the ϵ-greedy policy (2.8-2.9) or the soft-max policy (2.10) becomes a computationally costly operation when the action space is large (e.g. multidimensional), especially when a computationally expensive function approximator is used to represent the Q-function. Therefore, these algorithms are likely to create time delay between observing the state and choosing an action in a real system. The effects of this delay are discussed in Section 2.4.

Popular on-line TD control algorithms are Q-learning and SARSA. SARSA is an *on-policy* algorithm, estimating the value function for the policy being followed.

Q-learning is an *off-policy* algorithm under which $Q(s,a)$ converges to the optimal value function $Q^*(s,a)$ belonging to the optimal policy π^*, independently of the policy actually followed during learning. The TD-errors for these algorithms are computed as follows:

$$\begin{aligned} \delta_{\mathrm{TD}_{\mathrm{SARSA}},k+1} &= r_{k+1} + \gamma Q_k(s_{k+1}, a_{k+1}) - Q(s_k, a_k) \\ \delta_{\mathrm{TD}_{\mathrm{Q}},k+1} &= r_{k+1} + \gamma \max_{a'} Q_k(s_{k+1}, a') - Q(s_k, a_k) \end{aligned} \tag{2.11}$$

To speed up convergence, SARSA and Q-learning can be combined with eligibility traces, see, e.g., (Sutton and Barto, 1998), thereby forming SARSA(λ) and Q(λ), respectively. With eligibility traces, the TD error is not only used to update $Q_k(s,a)$ for $s = s_k, a = a_k$, but also for state-action pairs that were visited earlier in the episode. In this process, more recently visited (s,a)-pairs receive a stronger update than pairs visited longer ago. For discrete state-action spaces (function approximation is explained in Section 2.3), $Q(s,a)$ is updated, $\forall s \in S, \forall a \in A$, as follows:

$$Q^\pi_{k+1}(s,a) = Q^\pi_k(s,a) + \alpha \delta_{\mathrm{TD},k+1} e_{k+1}(s,a) \tag{2.12}$$

with

$$e_{k+1}(s,a) = \begin{cases} \gamma\lambda e_k(s,a) + 1 & \text{if } s = s_k \text{ and } a = a_k \\ \gamma\lambda e_k(s,a) & \text{otherwise} \end{cases} \tag{2.13}$$

where $e_k(s,a)$ contains the eligibility of a state-action pair at time step k, with $e_0(s,a) = 0$, and λ the (eligibility) trace discounting factor. For Q(λ), the eligibility of preceding states is only valid as long as the greedy policy is followed. Thus, for Q(λ), e is also reset after an exploratory action. Choosing a value for λ can be done in the same way as for γ using a characteristic time scale for the eligibility of the agent's actions:

$$\lambda = e^{-\frac{h}{\tau_{\mathrm{elig}}}} \tag{2.14}$$

2.3 Function approximation for RL in high-dimensional state-action spaces

Function approximation (FA) is a necessary component of RL in continuous state-action spaces. While the action-value function $Q(s,a)$ of a discrete state-action space can be exactly represented using a finite amount of memory, this is not possible for a continuous state-action space. Instead, the action-value function is approximated by $\hat{Q}(s,a,\boldsymbol{\theta})$, a function of s, a and a parameter vector $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots, \theta_n]^T$ with a finite number of elements n.

Function approximation results in *generalization*, i.e., the change of an element of $\boldsymbol{\theta}$ results in a change of $\hat{Q}(s,a,\boldsymbol{\theta})$ in a *region* of the state-action space. Generalization can be global or local; the value of a single element of $\boldsymbol{\theta}$ can influence

$\hat{Q}(s, a, \boldsymbol{\theta})$ at every point in state-action space, or only locally in a (small) region. Generalization can lead to faster learning, since a learning update for (s_k, a_k) influences $\hat{Q}(s, a, \boldsymbol{\theta})$ in a region around (s_k, a_k). Many FA techniques have been used in the context of RL, the most common being CMAC (Albus, 1971; Albus, 1981), tile coding (Sutton, 1996; Sutton and Barto, 1998; Stone and Sutton, 2001), support vector machines, fuzzy approximation (Buşoniu et al., 2007), radial basis functions (RBF's) and neural networks. These techniques differ in computational complexity and the smoothness of the approximation (Kretchmar and Anderson, 1997).

Combining TD algorithms such as SARSA and Q-learning with function approximation can introduce convergence problems (Gordon, 1995; Gordon, 2001) such as complete divergence or oscillation around a good solution. Several TD algorithms have been introduced that address these convergence problems (Baird, 1995; Precup, Sutton, and Dasgupta, 2001; Sutton, Szepesvári, and Maei, 2009). In this thesis, however, it was possible to obtain satisfactory convergence behavior with SARSA and Q-learning.

2.3.1 Training function approximators

Training a function approximator is an example of supervised learning; the desired value of $\hat{Q}(s_k, a_k, \boldsymbol{\theta}_{k+1})$ is known – $Q_{k+1}(s_k, a_k)$ – and the elements of $\boldsymbol{\theta}$ should be adjusted towards a more accurate approximation. When $\hat{Q}(s, a, \boldsymbol{\theta})$ is a smooth differentiable function of $\boldsymbol{\theta}$, a common way to train a function approximator is to use a gradient descent update rule of the form

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha(Q_{k+1}(s_k, a_k) - \hat{Q}_k(s_k, a_k, \boldsymbol{\theta}_k))\nabla_{\boldsymbol{\theta}}\hat{Q}_k(s, a, \boldsymbol{\theta})|_{s=s_k, a=a_k, \boldsymbol{\theta}=\boldsymbol{\theta}_k} \quad (2.15)$$

with $Q_{k+1}(s_k, a_k)$ the new estimate calculated by the RL algorithm at time k for $s = s_k$ and $a = a_k$, $\hat{Q}_k(s_k, a_k, \boldsymbol{\theta}_k)$ the current approximated value, α the learning rate or step size and $\nabla_{\boldsymbol{\theta}}\hat{Q}_k(s, a, \boldsymbol{\theta})$ the vector of partial derivatives

$$\nabla_{\boldsymbol{\theta}}\hat{Q}_k(s, a, \boldsymbol{\theta}) = [\frac{\partial \hat{Q}_k(s, a, \boldsymbol{\theta})}{\partial \theta^1}, \frac{\partial \hat{Q}_k(s, a, \boldsymbol{\theta})}{\partial \theta^2}, \ldots, \frac{\partial \hat{Q}_k(s, a, \boldsymbol{\theta})}{\partial \theta^n}]^T \quad (2.16)$$

Typically, $\hat{Q}(s, a, \boldsymbol{\theta})$ cannot exactly represent $Q(s, a)$ due to the limited size of $\boldsymbol{\theta}$. Choosing $\alpha < 1$ results in averaging over multiple updates and reduces fluctuations of the values of $\boldsymbol{\theta}$.

2.3.2 Linear function approximation

In the special case of *linear* function approximation, the approximate value function is linear in its parameters. For each θ^i there is an associated basis function

(BF) $\phi^i(s,a)$, also called feature, which defines the spatial influence of that parameter in the state-action space. $Q(s,a)$ is now approximated as

$$\hat{Q}_k(s,a,\boldsymbol{\theta}_k) = \boldsymbol{\theta}_k^T \boldsymbol{\phi}(s,a) = \sum_{i=1}^{n} \theta_k^i \phi^i(s,a) \tag{2.17}$$

The gradient in (2.15) has the following simple form for linear FA:

$$\nabla_{\boldsymbol{\theta}} \hat{Q}_k(s,a,\boldsymbol{\theta}) = \boldsymbol{\phi}(s,a) \tag{2.18}$$

For TD learning with linear function approximation, every θ^i is updated as follows:

$$\theta_{k+1}^i = \theta_k^i + \alpha \delta_{\mathrm{TD},k+1} \phi^i(s_k,a_k) \tag{2.19}$$

Eligibility traces can be implemented (see, e.g., (Sutton and Barto, 1998)) by storing an eligibility trace per feature instead of per discrete state-action pair in a column vector $\boldsymbol{e} = [e^1, e^2, \ldots, e^n]^T$. Every θ^i can then be updated according to

$$\theta_{k+1}^i = \theta_k^i + \alpha \delta_{\mathrm{TD},k+1} e_k^i \tag{2.20}$$

The elements e^i of the eligibility traces vector can be updated as follows:

$$e_{k+1}^i = \gamma \lambda e_k^i + \frac{\partial \hat{Q}_k(s,a,\boldsymbol{\theta}_k)}{\partial \theta^i} = \gamma \lambda e_k^i + \phi^i(s,a) \tag{2.21}$$

and $\boldsymbol{e}_0 = \mathbf{0}$.

Non-linear function approximation techniques, such as neural-networks trained with backpropagation, have the potential to achieve better approximations with fewer parameters when the basis functions are carefully chosen. However, in general, it is more difficult to choose the parameters of such approximators.

2.3.3 High-dimensional state-action spaces

An important property of robotic systems is the high dimensionality of the state-action space. Robot Leo, a robot of only moderate hardware complexity and number of degrees of freedom (see Chapter 3), already has a state space with dimension 16, while its action space has dimension 7. Although the dimensionality of such a robot can be reduced using virtual constraints (i.e., by using conventional controllers that effectively remove a degree of freedom by keeping joint angles constant, or making them a function of the other joint angles), the state-action space of a robotic task typically has dimension 10 or higher. This poses an important computational constraint on the function approximation technique to be used, as will now be explained in more detail.

The simplest form of (linear) function approximation is the tabular approximation, in which the state-action space is discretized into hypercubes. A hypercube represents a BF that has value 1 inside the hypercube and 0 elsewhere. Each hypercube has an associated parameter θ_i. This method is computationally fast and simple, but since its generalization width equals its resolution, learning is either slow and accurate (with large memory requirements), or fast but inaccurate (with less memory requirements). Usually, in high-dimensional state-action spaces, a satisfactory resolution results in prohibitively high memory requirements due to the *curse of dimensionality*: when every dimension $d = 1, 2, \ldots, D$ is discretized into M bins within its typical range, the number of elements of $\boldsymbol{\theta}$ becomes M^D. The memory requirements for $\boldsymbol{\theta}$ are then exponential in D.

For other FA techniques, the curse may also apply to the computational requirements. In practice, (2.17) is computed by only summing over the BFs that have non-zero value at the specified (s, a). The computational load consists of finding the non-zero BFs, computing their value at the specified (s, a), retrieving the values of their corresponding θ_i from memory and, finally, computing (2.17). Consider again the tabular approximation, but this time the BF's are also non-zero in at least one neighboring hypercube, in which each dimension adds new neighbors. Examples of such approximation schemes using product-space BF's are radial basis functions (RBFs), whose influence is a function of the distance between (s, a) and the location of the BF, and fuzzy approximators (Horiuchi et al., 1996; Jouffe, 1998; Buşoniu et al., 2007). The number of BFs that have non-zero value when computing $\hat{Q}(s, a, \boldsymbol{\theta})$ for a single state-action pair according to (2.17) is at least N^D with $N \geq 2$, a number that increases exponentially in D. This increases the time needed to compute (2.17). A large computational load will cause a time delay on real systems between perceiving state s and taking action a when using, e.g., (2.8) for action selection. In Section 2.4, it is shown that this delay can cause convergence problems.

2.3.4 Tile coding

Tile coding, also known as CMAC (Albus, 1971; Albus, 1981), is a linear approximation scheme. The scheme is based on a (possibly large) set of overlapping multi-dimensional grids of BFs, or tilings (named receptive fields in the CMAC literature) $\mathrm{T}^1, \mathrm{T}^2, \ldots, \mathrm{T}^K$ with K the number of tilings. A single BF is defined for each hypercube in each tiling. This BF has value 1 inside its hypercube and 0 elsewhere. Therefore, each (s, a) has exactly K BFs with non-zero value. Due to their simple shape, finding the BFs with non-zero value for (s, a) is relatively easy. Due to these factors, the computational complexity of tile coding is fairly limited [3]. Each tiling is shifted (displaced) from the origin with a different offset as to create an even distribution of BFs throughout the multi-dimensional state-action space. A good choice of the displacement vector A_k^d for each tiling k in

---

[3] A thorough analysis of the computational complexity of tile coding and other linear function approximation techniques is non-trivial and implementation dependent, and therefore beyond the scope of this chapter.

each dimension d is (Miller et al., 1990):

$$A_k^d = r_d(k-1)(1+2(d-1))/K \tag{2.22}$$

with r_d the edge length of the hypercube for dimension d. An example of tile coding in three dimensions with four tilings, shifted according to (2.22), can be found in Figure 2.1.

The number of hypercubes in each tiling still rises exponentially with D. However, due to the tiling displacement, the input quantization is much finer than the size of the hypercubes. Therefore, good approximation can often be achieved with relatively large hypercubes (compared to, e.g., the tabular approach), resulting in relatively low memory requirements, especially in high-dimensional spaces.

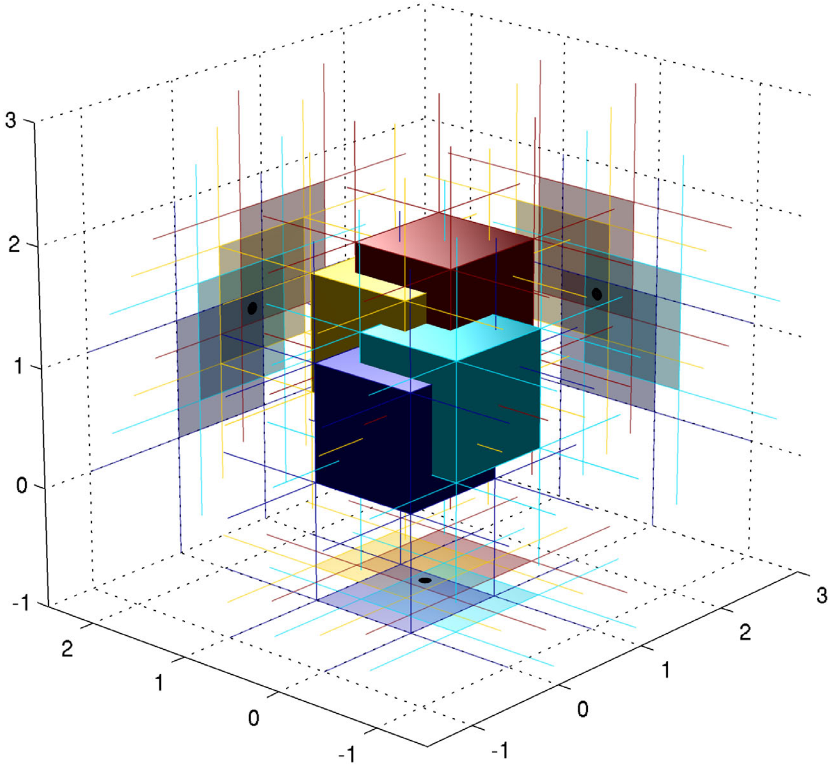


Figure 2.1: *Example of tile coding function approximation in three dimensions with four tilings. The four cubes visualize the tiles activated by a certain state. Their projections are shown on the xy-, xz- and yz-plane. The state is shown as a black dot in the projections. The function value at the dot is calculated according to (2.17) by summing the elements of* $\boldsymbol{\theta}$ *associated with the four activated tiles.*

While in tile coding the BFs are typically binary, i.e., have a uniform value of 1 across their hypercube, this is not necessary (Miller et al., 1990; Lane, Handelman, and Gelfand, 1992; An, 1991). A possible extension – which adds to the computational complexity compared to binary BFs – is to use radial basis functions that have their center in the center c^i of each hypercube:

$$\phi^i = \begin{cases} \mathrm{f}(||(s,a) - c^i||_p) & \text{, if } (s,a) \text{ inside hypercube} \\ 0 & \text{, otherwise} \end{cases} \tag{2.23}$$

with p the norm of the distance function and $\mathrm{f}(\,\cdot\,)$ an arbitrary function of distance. While binary BFs produce a piecewise constant approximation, continuous alternatives of the form (2.23) produce smoother approximations. A downside of such BFs is that for high values of D, $\boldsymbol{\phi}^i$ is small-valued in most of the hypercube's volume. To see this, consider a hypersphere with radius $\frac{r}{2}$ located in the center of a hypercube with edge length r, i.e., a hypersphere touching the hypercube's faces. The volume of this hypersphere scales with $(\frac{r}{2})^D$, while the hypercube's volume equals r^D. Therefore, most of the hypercube's volume is located at a distance $\geq \frac{r}{2}$ from the center, where $\boldsymbol{\phi}^i$ is usually small (also see, e.g., (Dasgupta, 2010)). This effect, however, becomes smaller for larger values of p. In (Miller et al., 1990), satisfactory continuous alternatives for binary BFs were found using the infinity norm distance $||\cdot||_\infty$. Note that when using non-binary BFs, it is advised to normalize (2.17) with the sum of BF values.

In this thesis, all experiments were conducted using tile coding, because it is a linear FA technique with low computational complexity that showed important successes in the past (Sutton, 1996; Stone, Sutton, and Kuhlmann, 2005; Schuitema et al., 2005). We used binary BFs, since we were unable to produce consistently better results using continuous BFs.

2.3.5 Appropriate feature spaces

Despite numerous attempts to automatically set up the feature space (i.e., the number, shape and placement of features) for an MDP (Chow and Tsitsiklis, 1991; Moore and Atkeson, 1995; Munos and Moore, 2002; Whiteson, 2010; Bernstein and Shimkin, 2010), to the best of our knowledge there exists no method that guarantees a certain level of system performance, related to the rewards, for an arbitrary (fully observable) MDP, especially for high-dimensional state(-action) spaces. Therefore, in the remainder of this thesis, feature spaces of MDPs are manually tuned towards satisfactory learning time, final system performance and computational requirements. Usually, fast learning and high final system performance are conflicting goals when setting up the feature space and a preference for either will lead to a different choice of features. The discovery of an algorithm that generates a feature space that is guaranteed to result in a certain level of final system performance would be a major contribution to the field and would greatly increase the applicability of RL in general, but is outside the scope of this thesis.

2.3.6 Storing the parameters in memory

A sufficiently accurate approximation of $Q(s, a)$ can require a considerable number of feature parameters. There are several ways in which the number of feature parameters that need to be stored in memory can be reduced. Below we discuss the two most important methods used in the experiments in this thesis.

Exploiting system symmetry

Many robotic systems and their corresponding tasks contain symmetry at multiple levels. This means that situations that are essentially identical, but mirrored or rotated, offer the opportunity to be learned only once by exploiting this symmetry when estimating and storing the (action-)value function. For example, the situation of a walking soccer robot that is given the task to dribble a ball and kick it into the goal contains at least two levels of symmetry. The robot itself is usually mirror-symmetrical regarding its left and right side, so that when it learned to kick the ball with its left leg, it can automatically kick the ball with its right leg. Furthermore, game situations are mirror-symmetrical with regard to approaching the goal from the left or the right side. Exploiting symmetries can greatly reduce memory requirements and learning time. In practice, when using function approximation, this requires to map symmetrical states to the same features. Sometimes, there are multiple possible implementations. For example, a bipedal robot with symmetrical left and right sides that learns to walk can benefit from this symmetry, but it is not trivial which states should be considered 'mirrored' and which states are 'original'. One solution is to consider all states where the right foot is in front of the left foot as 'mirrored' and map (mirror) them to equivalent states where the left foot is the frontmost foot. Another solution is to define the state space in terms of stance leg and swing leg, where the stance leg is the leg whose foot touches the floor. If both feet touch the floor, the stance leg is the leg whose foot is in front of the other. The latter definition is used in the remainder of this thesis when setting up state spaces for walking robots. This is illustrated in Figure 2.2.

Hashing

For a robotic task, the bounds of the state variables are often unknown a priori and possibly very wide, which makes it difficult to reserve memory storage for the function approximator for the complete state space in advance. More importantly, the robot will most likely not visit all possible states inside the state space region bounded by the bounds of the individual state variables. Therefore, it is beneficial to store an approximation of the (action-)value function only for states that are actually observed. Hashing is a convenient method to map a large state space to a (much) smaller memory space; see, e.g., (Sutton and Barto, 1998). When combining function approximation with hashing, storage for feature parameters is only allocated for features associated with state-action pairs that are actually

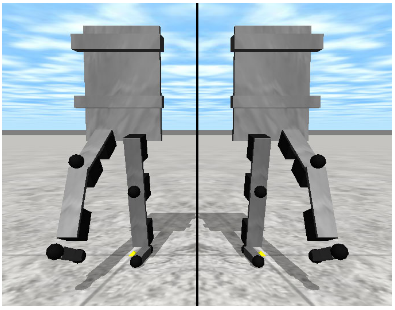


Figure 2.2: *Bipedal walking robot Leo, showing two states that are mirror-symmetrical regarding its left and right side. When estimating $Q(s, a)$ with function approximation, learning time can be decreased by mapping both states to the same features.*

required in computations. After computing the state-action space locations of the features that have non-zero value for a given (s, a), the hashing function[4] maps these feature locations to the memory locations of their parameter values. These memory locations are guaranteed to lie within a predefined, fixed memory range.

In addition, it is not common for the agent to have tried (or eventually try) all available actions $a_k \in A$ for a state s_k that it visits. For the commonly used greedy policy (2.8), however, $Q(s_k, a_k)$ is evaluated for all $a_k \in A$. In this process, many feature values are requested that have never been updated before and will not be updated that time step if their corresponding action is not selected. The parameters of these features contain initialization values. In these cases, it suffices to simply return the initialization value of these features by computation, instead of reserving actual memory for the storage of these feature parameters. This further reduces memory requirements.

2.4 Control Delay in Reinforcement Learning for Real-Time Dynamic Systems: A Memoryless Approach

Control delay – the delay between measuring a system's state and acting upon it – is always present in real systems. Real-time learning control algorithms that require a considerable amount of computation can result in a significant control delay. This section discusses the effects of control delay on real-time RL and proposes possible counter measures. This section is largely based on (Schuitema et al., 2010a).

---

[4] In principle, any hashing function can be used. The software implementations used throughout this thesis made use of the MurmurHash2 hashing function (Appleby, 2008) because of its good hashing properties and small computational footprint.

2.4.1 Introduction

Reinforcement Learning (RL) is a promising approach to adding autonomous learning capabilities to robotic systems. However, examples of real dynamic systems controlled in real-time by RL are still rare; most work on RL is done in simulation. An important difference between such real systems and their simulations is the presence of time delay between observation and control action: *control delay.* Every such real system that runs on-line RL will have a non-zero control delay caused by its sensors, actuators and controller due to computation and communication (also called network delay). The delay is illustrated in Fig. 2.3. In this work, we show that besides negatively influencing the final solution, control delay can be particularly detrimental to the learning process itself, if it remains unaccounted for.

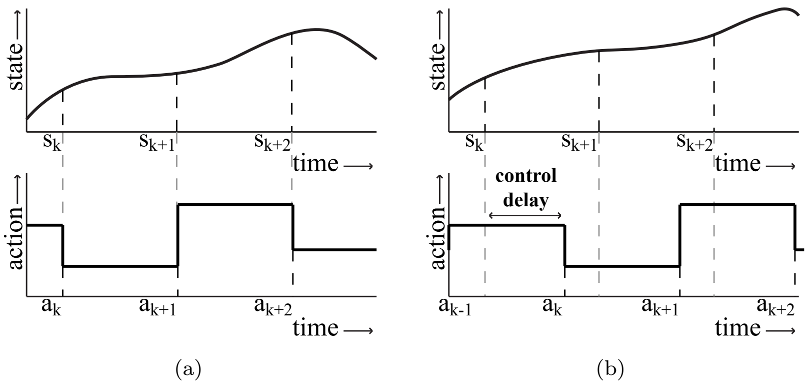


Figure 2.3: *Schematic illustration of control delay between measuring state s_k and acting accordingly with action a_k. (a) No delay. (b) With control delay.*

Although control delay is well studied in the context of conventional (non-learning) control, the influence of control delay on RL has received little attention in the RL literature. Currently, there are two state-of-the-art approaches. In the first approach, the state space of the learning agent is augmented with the actions that influenced the evolution of the system during the delay interval (Katsikopoulos and Engelbrecht, 2003). While this approach works well, the state space increase can cause a large increase in learning time and memory requirements. The second approach uses state prediction. A model of the underlying *undelayed* process is learned, and the control action is chosen for the future state after the delay as predicted by the model (Walsh et al., 2009). This adds the extra burden of acquiring a model of the system, while the added computational complexity may actually increase the delay itself.

As opposed to methods that augment the state space or estimate an explicit model, *memoryless* methods form an alternative approach. Such methods base the next control action only on the most recent observation. A memoryless method that is known empirically to produce good results is SARSA(λ) (Sutton and Barto, 1998; Walsh et al., 2009). The downside of memoryless approaches is that they are likely to perform suboptimally, because they have no means of predicting the state in which the control action will take effect. Furthermore, SARSA(λ) does not take the presence of delay into account in its learning updates. However, memoryless methods do not have the added complexity of learning a model or enlarging the state space, and may perform acceptably, especially when the delay is small.

In this work, we introduce two new memoryless, online algorithms – dSARSA(λ) and dQ(λ). While their complexity remains comparable to that of SARSA(λ) and Q(λ), they exploit the knowledge about the length of the delay to improve their performance. In addition, we present an extension to these algorithms which is, under certain conditions, applicable to systems in which the delay is not an integer multiple of the time step. While this is most likely to be true for real robotic systems, this case has not been considered in previous literature on RL with delay.

2.4.2 Control delay in MDPs

In this section, we define an extension of the MDP definition from Section 2.1 that models control delay. We define control delay as the time delay between the moment of observing a state and the moment when acting upon that state takes effect. Control delay, which we will further refer to simply as delay, can be caused both by delayed observation, e.g., due to transportation of the measured data, and by delayed actuation, e.g., due to lengthy computations. In this work, we only consider constant delays[5]. We define the relative delay τ_d as

$$\tau_\mathrm{d} = \frac{T_\mathrm{d}}{h} \tag{2.24}$$

with T_d the absolute delay and h the sampling period.

In (Katsikopoulos and Engelbrecht, 2003), it is shown that from the point of view of the learning agent, there is no functional difference between observation delay and action delay; both add up to the delay between the moment of measurement and the actual action.

From the TD error definitions for SARSA and Q-learning (2.11), it can be seen that the estimate of the Q-function is adjusted every time step according to a supposedly Markovian (stochastic) state transition based on state s_k and action a_k; the agent learns the effect of executing action a_k. In the delayed case, the action that is executed in s_k is not a_k. If τ_d is an integer, i.e., the delay is an

---

[5] Variable delays in real robotic systems (interesting for future work) can be made (nearly) constant by artificially adding delay to the system's inherent delay until every sample meets the system's worst case delay.

integer multiple of h, the actually executed action is $a_{k-\tau_\mathrm{d}}$. If τ_d is not an integer, *two* actions are (partially) active during the state transition from s_k to s_{k+1}.

The fact that state transitions become dependent on actions selected in the past, which are not part of the input of the learning agent, results in a violation of the Markov property. This relates the problem of delay to the framework of Partially Observable MDPs, or POMDPs.

Existing approaches to MDPs with delay

Delay implies that decisions take effect in future states. When the future state (distribution) can be predicted from the most recent observation and the upcoming actions, e.g. by an explicit state transition model, optimal action selection becomes possible again. From (Katsikopoulos and Engelbrecht, 2003), it is known that when the state space of the MDP is expanded with the actions taken in the past during the length of the delay, forming the augmented state space $I_{\tau_\mathrm{d}} = S \times A^{\tau_\mathrm{d}}$ with τ_d integer-valued, a constant delay MDP can be reduced to the regular MDP $\langle I_{\tau_\mathrm{d}}, A, T, R \rangle$. This formulation makes it possible to use existing RL techniques to solve the delayed case. However, since the state space dimensionality grows with the number of delay steps, learning time and memory requirements will rapidly increase with this approach.

In (Walsh et al., 2009), an approach called Model Based Simulation is presented, in which a state transition model of the underlying *undelayed* MDP is learned by matching actions with the states in which they actually took place. Model-based RL is then used to estimate the optimal (action-)value function. However, such an approach has the additional burden of learning an explicit model of the system.

A *memoryless policy* is a policy that only bases its actions on the current state s, despite the delay. This means that it does not take into account the future state in which the action takes effect. Therefore, it is likely to perform worse than methods that use a model for state prediction. From (Singh, Jaakkola, and Jordan, 1994), it is known that the best memoryless policy of a POMDP can be arbitrarily suboptimal in the worst case. However, this does not mean that a memoryless policy cannot achieve an acceptable level of performance in a given problem. In (Loch and Singh, 1998), it is argued that SARSA(λ) performs very well in finding memoryless policies for POMDPs, compared to more sophisticated and computationally much more expensive methods. In (Walsh et al., 2009), Model Based Simulation is also compared with SARSA(λ), which performs surprisingly well, but not as well as their model-based approach.

In this work, we will use the knowledge on the source of the partial observability - the delay, and its length - to create memoryless algorithms that outperform SARSA(λ), while having similar complexity. They do not enlarge the state space and they are model-free.

In all the aforementioned work, only delays of an integer multiple of the time step were considered, while in real-time dynamic systems, this is usually not the case. Therefore, we also present an extension to our algorithms that make them, under certain conditions, applicable to the case where τ_d can have any non-integer value.

2.4.3 TD learning with control delay: dSARSA and dQ

We now present our contribution: modified versions of SARSA and Q-learning that exploit knowledge about the delay. In these versions, instead of updating $Q(s_k, a_k)$, updates are performed for the *effective* action $\hat{a}_k$ that actually took place in s_k. We will first consider the case where τ_d is an integer, which results in the following effective action

$$\hat{a}_k = a_{k-\tau_\mathrm{d}}. \tag{2.25}$$

The TD errors are computed as follows

$$\begin{aligned} \delta_{\mathrm{TD}_{\mathrm{dSARSA}},k+1} &= r_{k+1} + \gamma Q(s_{k+1}, \hat{a}_{k+1}) - Q(s_k, \hat{a}_k) \\ \delta_{\mathrm{TD}_{\mathrm{dQ}},k+1} &= r_{k+1} + \gamma \max_{a'} Q(s_{k+1}, a') - Q(s_k, \hat{a}_k) \end{aligned} \tag{2.26}$$

We call these variants dSARSA and dQ, where ‘d’ stands for ‘delay’. Both algorithms are memoryless, which means their policy depends only on the current state. Action *execution* is still delayed. They use the knowledge on the length of the delay to improve the learning updates. Eligibility traces can be introduced by modifying (2.13) at the following point: $e_{k+1}(s, a) = \gamma\lambda e_k(s, a) + 1$ if $s = s_k$ and $a = \hat{a}_k$. We will now discuss the most important properties of dQ and dSARSA.

Behavior of dQ-learning

Regular Q-learning is an off-policy algorithm, which means that $Q(s, a)$ is not based on the action selection policy. The only restriction on the action selection policy is that all state-action pairs continue to be updated. In dQ-learning, we restored the temporal match between states and actions. This means that with dQ-learning, the action-value function will converge to the optimal action-value function of the underlying *undelayed* MDP. Action selection itself, however, still suffers from control delay.

When combining dQ-learning with eligibility traces, forming dQ(λ)-learning, convergence is not guaranteed, since eligibility of preceding states is only valid when the greedy policy is being followed. With delayed execution, this is generally not the case. In our empirical evaluations in Section 2.4.4, we will indeed see that the use of eligibility traces can lead to rapid divergence. However, dQ(0) is still an interesting algorithm due to its convergence properties, and we expect it to be an improvement over regular Q-learning.

Behavior of dSARSA

Regular SARSA is an on-policy algorithm. Therefore, the estimated Q-function is based on the policy's action selection probabilities. In the case of dSARSA, execution of actions is still delayed. Therefore, although we restored the temporal match between states and actions in the updates, the actual policy still depends on the history of the agent. The convergence proof of SARSA (Singh et al., 2000) requires the policy to be either greedy in the limit with infinite exploration (GLIE; e.g., ϵ-greedy with decaying ϵ), or restricted rank-based randomized (RRR; e.g., ϵ-greedy with fixed ϵ), both of which dSARSA cannot provide. dSARSA can choose, but not execute, greedy actions with respect to its value function due to the delayed execution of actions. This convergence proof therefore does not hold. However, we expect that dSARSA is still an improvement over SARSA due to the incorporation of knowledge about the delay. When combining dSARSA with eligibility traces, forming dSARSA(λ), the eligibility of preceding states is in this case justified because the action-value function is based on the actually executed policy (the delayed policy). Therefore, dSARSA and dSARSA(λ) are expected to give the same solution.

From the above reasoning and from existing convergence proofs, we can conclude that from the algorithms Q(λ), dQ(λ), SARSA(λ) and dSARSA(λ), only dQ(0) (i.e., without eligibility traces) is guaranteed to converge in the delayed case. More insight into the convergence of dSARSA(λ) and dQ(λ) remains important future work.

The actions that are selected prior to visiting a state s are responsible for the action actually executed in s. While these actions can in principle not be predicted from s, their probability distribution might contain structure when the policy is quasi-stationary and the distribution of initial states is fixed. When the learning rate α is reduced, the policy changes less rapidly, and Q-values represent the average of a larger number of experiences. From this reasoning, we expect the performance of memoryless policies to improve with decreasing α. In Section 2.4.4, we will verify this hypothesis.

Note that although dSARSA and dQ are memoryless and model-free, they can still benefit from a (learned) state transition model by using it to predict the future state after τ_d steps and selecting the action for that predicted state.

Non-integer values of τ_d

We will now consider the case where τ_d is not an integer. From Figure 2.3, it can be seen that the control action is a combination of the two previously selected actions $a_{k-\lceil\tau_\mathrm{d}\rceil}$ and $a_{k-\lceil\tau_\mathrm{d}\rceil+1}$ with $\lceil\tau_\mathrm{d}\rceil$ the smallest integer for which $\tau_\mathrm{d} \leq \lceil\tau_\mathrm{d}\rceil$. Therefore, dSARSA and dQ cannot be directly applied in this case. We will now show that in the special case where the system dynamics can be accurately *linearized* at arbitrary states, over the length of one time step, it is possible to estimate the effective action $\hat{a}_k$ *without* knowledge of the system dynamics. In case $T_\mathrm{d} < h$, [6]the state transition of the real system can be shown to be, in a first approximation, equivalent to the case where the system would have executed the following *virtual* effective action $\hat{a}_k$ during the full time step:

$$\hat{a}_k = a_{k-1}\tau_\mathrm{d} + a_k(1 - \tau_\mathrm{d}) \tag{2.27}$$

Here we assume that the linear combination of two actions results in a new, valid action, which is the case for, e.g., actions of type motor torque or voltage. Switching to the continuous time notation, the locally linearized system at time $t = kh$ around state $s(t)$ has the form

$$\dot{s}(t) = Fs(t) + Ga(t) \tag{2.28}$$

with

$$a(t) = \begin{cases} a_{k-1} & \text{if } kh \leq t < kh + T_\mathrm{d} \\ a_k & \text{if } kh + T_\mathrm{d} \leq t < kh + h \end{cases} \tag{2.29}$$

and with F and G matrices. The exact solution to (2.28) in the delayed case with $T_\mathrm{d} < h$ is

$$\begin{aligned} s(kh+h) &= e^{Fh}s(kh) + \\ &\quad \int\limits_{kh}^{kh+T_\mathrm{d}} e^{F(kh+h-t')}\,dt'\,Ga_{k-1} + \int\limits_{kh+T_\mathrm{d}}^{kh+h} e^{F(kh+h-t')}\,dt'\,Ga_k \\ &= e^{Fh}s(kh) + \\ &\quad e^{F(kh+h)}\left(\int\limits_{kh}^{kh+T_\mathrm{d}} e^{-Ft'}\,dt'\,Ga_{k-1} + \int\limits_{kh+T_\mathrm{d}}^{kh+h} e^{-Ft'}\,dt'\,Ga_k\right) \end{aligned} \tag{2.30}$$

For small time steps, we can approximate the integrals of the form $\int_{t_1}^{t_2} e^{-Ft'}\,dt'$ as

$$\int\limits_{t_1}^{t_2} e^{-Ft'}\,dt' \approx I(t_2 - t_1) - \frac{1}{2}X(t_2^2 - t_1^2) \approx I(t_2 - t_1) \tag{2.31}$$

where the second order terms are neglected. Therefore, for small time steps, the approximate solution to (2.28) in the delayed case becomes

$$s(kh+h) = e^{Fh}s(kh) + e^{F(kh+h)}\left((T_\mathrm{d})\,Ga_{k-1} + (h - T_\mathrm{d})\,Ga_k\right) \tag{2.32}$$

---

[6] To avoid notation clutter, we make the assumption that $T_\mathrm{d} < h$. However, generalizing the results to larger delays is straightforward.

The effect of executing $\hat{a}_k$ from (2.27) can be analysed by considering the exact solution to (2.28) where $a(t) = \hat{a}_k$ for $kh \leq t < hk + h$:

$$\begin{aligned} s(kh+h) &= e^{Fh} s(kh) + \\ &\quad \int_{kh}^{kh+h} e^{F(kh+h-t')}\, dt'\, G a_{k-1} \frac{T_\mathrm{d}}{h} + \\ &\quad \int_{kh}^{kh+h} e^{F(kh+h-t')}\, dt'\, G a_k \frac{h - T_\mathrm{d}}{h} \\ &= e^{Fh} s(kh) + \\ &\quad e^{F(kh+h)} \int_{kh}^{kh+h} e^{-Ft'}\, dt' \left(G a_{k-1} \frac{T_\mathrm{d}}{h} + G a_k \frac{h - T_\mathrm{d}}{h} \right) \end{aligned} \qquad (2.33)$$

When applying the small time step approximation from (2.31) to (2.33), it becomes equal to (2.32), showing that the delayed system behaves approximately as a non-delayed system in which $\hat{a}_k$ was active. In this special case, dSARSA and dQ can again be applied using (2.27).

2.4.4 Empirical evaluations

In this section, we empirically evaluate the performance of dSARSA and dQ on two test problems. We consider integer values of τ_d in a W-shaped maze – a gridworld problem – and non-integer values of τ_d in the simulation of a two-link manipulator – a robotic system.

Integer values of τ_d: the large W-maze

We first consider the large W-maze, see Figure 2.4b, in which τ_d is always an integer. This is a modified version of the W-maze, see Figure 2.4a, that was used in (Walsh et al., 2009) to empirically illustrate several approaches to delayed MDPs. In the original W-maze, a delay of one time step easily results in the agent missing the middle corridor that leads towards the goal. The larger corridors and goal state of the large W-maze make the effect of delayed policy execution less severe.

We evaluated the performance of dQ(0), dQ(λ) and dSARSA(λ) in comparison with Q(λ) and SARSA(λ). While dSARSA(0) is merely a slower version of dSARSA(λ) and of less interest, dQ(0) is the only algorithm with convergence guarantees (see Section 2.4.3) and we therefore include it. We set the delay to 2 time steps. The only reward in the system is -1 for every action, so that the agent learns to reach the goal as fast as possible. We did not use discounting ($\gamma = 1$). We used the ϵ-greedy policy (2.9) with $\epsilon = 0.1$ (fixed).

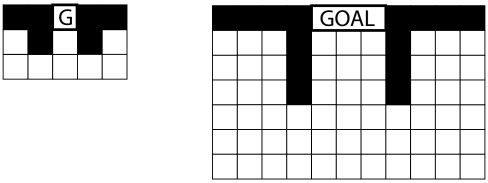


Figure 2.4: *The W-maze (left) and the large W-maze (right), which has larger corridors and a larger goal region.*

First, we compared the policy performance from the solutions found by all methods, as a function of the learning rate α. While keeping α constant, the agent was allowed to learn for $1 \cdot 10^6$ time steps. The policy performance was periodically measured by letting the agent start a trial from each of the 60 possible positions in the maze, execute the greedy policy until the goal was reached (with a maximum of 300 steps), and summing up all trial times (ergo, smaller is better). The average of all performance measurements during the last $5 \cdot 10^5$ steps is plotted against α in Figure 2.5.

The graph shows that for all methods, except dQ(0), the performance is much better for smaller values of α. This confirms the hypothesis that memoryless policies for delayed MDPs can improve their performance by averaging over more samples. This averaging is not needed by dQ(0). Despite its lack of eligibility traces, dQ(0) can still learn fast because it allows for high values of α.

Furthermore, we can observe that at equal values of α, dSARSA(0.8) performed at least twice as well as SARSA(0.8). Instability of dQ(0.8) occurred for larger values of α (Q-values quickly went to infinity).

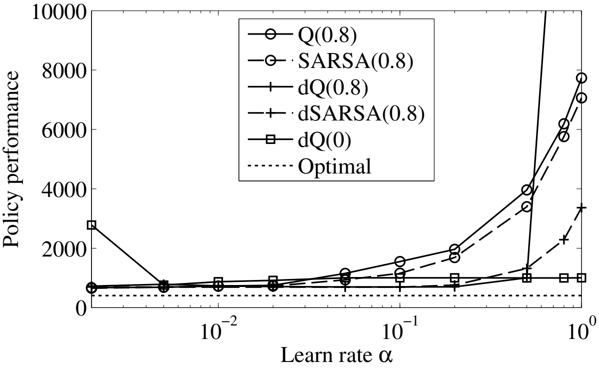


Figure 2.5: *Comparison of several memoryless TD algorithms on the large W-maze problem, showing the final average performance (lower is better) at different values of α, with α on a log scale (average of 20 runs).*

To compare the learning speed of all methods, we again kept ϵ and α constant. However, we chose a different α for each method for the following reason. An increase in α results in faster learning, but also in a decrease in final performance. Therefore, we chose the largest values of α for which each method was able to achieve approximately the same final performance of 800 or better. This resulted in the following values: dQ(0): $\alpha = 1$, dQ(0.8): $\alpha = 0.2$, dSARSA(0.8): $\alpha = 0.02$, Q(0.8): $\alpha = 0.02$, SARSA(0.8): $\alpha = 0.2$. dQ(0) is an exception; for this method, the average performance did not drop below approx. 1000. The result can be found in Figure 2.6. We can observe that in order to achieve the same final performance, dSARSA(0.8) learns much faster than SARSA(0.8) and Q(0.8). Although dQ(0) does not benefit from eligibility traces, in this particular test it can learn as fast as dQ(0.8) and dSARSA(0.8) because it allows for much higher values of α.

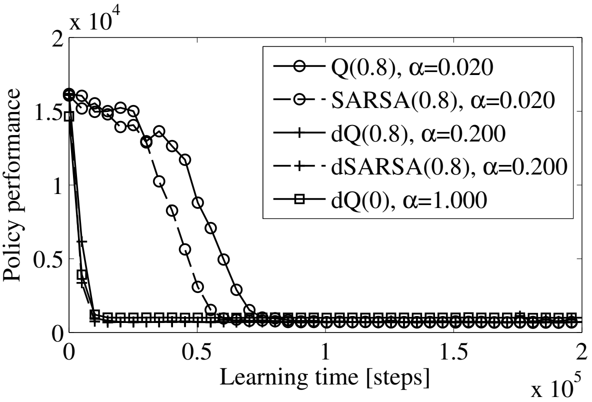


Figure 2.6: *Learning curves of several memoryless TD algorithms on the large W-maze problem. The learn rate α for each method was set to a value that produced a performance measure* < 800 *(average of 20 runs).*

Non-integer values of τ_d: the two-link manipulator

We now consider the simulation of the two link manipulator - a robotic system - depicted in Figure 2.7. This system has been used in previous work (see, for example, (Buşoniu et al., 2007)), has limited complexity and is well reproducible. In this system, the delay can be any non-integer multiple of the time step h, which is generally the case for robotic systems. The system has two rigid links, which are connected by a motorized joint. One end of the system is attached to the world, also with a motorized joint. The system moves in the two dimensional horizontal plane without gravity according to the following fourth-order non-linear dynamics:

$$M(\boldsymbol{\varphi})\ddot{\boldsymbol{\varphi}} + C(\boldsymbol{\varphi}, \dot{\boldsymbol{\varphi}})\dot{\boldsymbol{\varphi}} = \boldsymbol{\tau} \tag{2.34}$$

where $\boldsymbol{\varphi} = [\varphi_1, \varphi_2]$, $\boldsymbol{\tau} = [\tau_1, \tau_2]$, $M(\boldsymbol{\varphi})$ is the mass matrix and $C(\boldsymbol{\varphi}, \dot{\boldsymbol{\varphi}})$ is the Coriolis and centrifugal forces matrix.

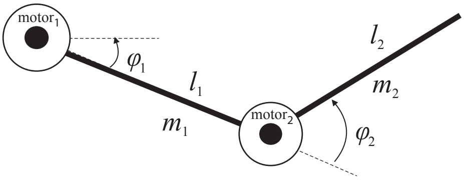


Figure 2.7: *Schematic overview of the two-link manipulator*

We consider delay values of $0 <= T_\mathrm{d} <= h$. The task of the system is to accomplish $\varphi_1 = \varphi_2 = \dot{\varphi}_1 = \dot{\varphi}_2 = 0$ as fast as possible by choosing torque signals for both motors. To this end, a reward is given when the angles and angular velocities are within a small region around 0: $|\varphi| < 0.17$rad and $|\dot{\varphi}| < 0.2$rad/s. Furthermore, a time penalty is given at each time step. The reward function r is

$$r_t = \begin{cases} 100, & \text{if } |\varphi_t| < 0.17 \text{ and } |\dot{\varphi}_t| < 0.2 \\ -1, & \text{all other cases (time penalty)} \end{cases} \tag{2.35}$$

The sampling period $h = 0.05$s, the learning rate $\alpha = 0.4$, the exploration rate $\epsilon = 0.05$ (again using the ϵ-greedy policy (2.9)), the discount factor $\gamma = 0.98$ and the trace decay rate $\lambda = 0.92$. We use tile coding function approximation (Sutton and Barto, 1998) with 16 tilings to approximate the Q-function. The state $s = (\varphi_1, \varphi_2, \dot{\varphi}_1, \dot{\varphi}_2)$. The tile widths for the state space are $\frac{1}{12}$rad in the φ_1 and φ_2 dimensions and $\frac{1}{6}$rad/s in the $\dot{\varphi}_1$ and $\dot{\varphi}_2$ dimensions. The tile widths for the action space are 1Nm in both dimensions. The action space $A = \tau_1 \times \tau_2$ is divided in 5 equidistant discrete steps in both dimensions, allowing for a total of $5 \cdot 5 = 25$ torque combinations. These parameters were selected empirically because they produced stable learning; they were not systematically optimized.

We first tested the effects of a delay of one time step or less using regular SARSA(λ). The results can be found in Figure 2.8. We can observe that for this particular system and for $\alpha = 0.4$, the system without delay smoothly converges. Delay values of $T_\mathrm{d} \geq 0.025$s ($\tau_\mathrm{d} \geq 0.5$), however, slow down the learning process or cause frequent divergence/unlearning. With a delay of a full time step, the system hardly learns for this value of α.

Next, we tested dSARSA(λ), using (2.27) to calculate the effective action. To this end, we set the delay to $T_\mathrm{d} = 0.037$s ($\tau_\mathrm{d} = 0.74$), a value which caused convergence problems for regular SARSA(λ). We compared it against the approach of using augmented state space $I_{\tau_\mathrm{d}} = S \times A^{\tau_\mathrm{d}}$ with $\tau_\mathrm{d} = 1$, so that $s_{aug} = (s, \tau_{1,k-1}, \tau_{2,k-1})$ (tile widths for $\tau_{1,k-1}, \tau_{2,k-1}$ are 1Nm). The results can be found in Figure 2.9. We can observe that for both approaches, the delayed system nicely converges to a solution, unlike with SARSA(λ) (see Figure 2.8). With dSARSA(λ), the system learns faster than with SARSA(λ) with augmented state

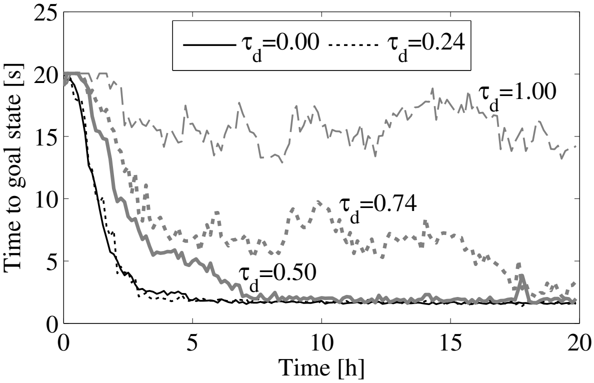


Figure 2.8: *Learning curve of the two-link manipulator running regular SARSA(λ) for* $\tau_\mathrm{d} = 0, 0.24, 0.50, 0.74$ *and* 1.0 *(average of 20 runs).*

space, but slower than the original, non-delayed system with SARSA(λ). Both methods have statistically indistinguishable final system performance. This suggests that the possible performance increase from a model based approach, such as in (Walsh et al., 2009), will only be marginal in this case.

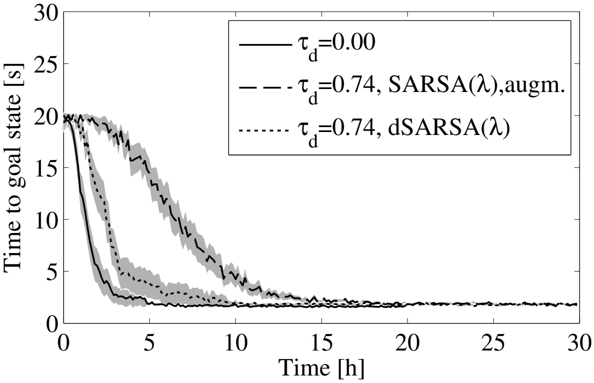


Figure 2.9: *Learning curves of the two-link manipulator with a delay of* $\tau_\mathrm{d} = 0.74$*, comparing dSARSA(λ) and SARSA(λ) with augmented state space, showing the average of 60 runs. The shaded area shows the 95% confidence bounds on the mean.*

2.4.5 Conclusions

In this work, we proposed new memoryless, model-free, online TD learning algorithms for MDPs with delay – dSARSA(λ) being the most important one –

that can perform better than classical TD algorithms by exploiting knowledge about the delay. We showed in a gridworld problem that dSARSA(λ) can significantly outperform Q(λ) and SARSA(λ) in terms of learning speed and final policy performance.

We extended our algorithms to systems in which the delay is not an integer multiple of the time step. We showed in a simulation of a robotic system that a control delay of less than a single time step can already degrade learning performance with classical SARSA(λ), while dSARSA(λ) can successfully learn in that case. dSARSA(λ) learned faster than a memory based approach – SARSA(λ) with augmented state space – while its policy performance did not degrade.

We can conclude that dSARSA(λ) is a better memoryless approach than classical SARSA(λ) for MDPs with delay, and that in some cases, it can compete with memory based approaches.

2.5 Conclusions

In this chapter, the main framework for RL was introduced – the Markov Decision Process – together with the main techniques that will be used throughout this thesis: temporal difference learning and linear function approximation. In addition, techniques have been discussed that reduce the computational requirements of RL in large state-action spaces, being tile coding, exploiting symmetry and using hashing.

Finally, we showed that time delay in the control loop can have a strong negative influence on the convergence of TD learning. We proposed the new memoryless TD algorithm dSARSA(λ) that can perform better than regular SARSA(λ) while maintaining low computational complexity. If a system is accurately linearizable at the time scale of a single state transition, the method can also be applied when the control delay is not an integer multiple of the sampling period.

Chapter 3

Reinforcement Learning on a real bipedal walking robot

Successful demonstrations of RL on real robots are much less common than successful applications in simulation. One possible cause is the lack of suitable robotic hardware specifically designed for RL. This chapter presents the design and experimental results of a bipedal walking robot specifically designed for RL: robot Leo. We chose to develop a walking robot because walking is a complex and challenging task, of which we[1] have extensive knowledge from previous work (Collins et al., 2005; Hobbelen, 2008). In addition, the topology of a bipedal walking robot creates the opportunity to learn other challenging tasks, such as walking stairs and standing up. The hardware design of robot Leo, following from an analysis of the requirements, is presented in Section 3.1. In Section 3.2, the software requirements and design are presented, as well as the simulation environment. In Section 3.3, experimental results in simulation and on the prototype are presented for two tasks; a relatively simple task of stepping up a stairs step with one leg, and the more complex task of learning to walk.

3.1 Hardware design

The specific properties and limitations of real robots have a large impact on their suitability for RL experiments. In this section, we derive the main hardware and software requirements that a RL robot should fulfill, and present our biped robot Leo that was specifically designed to meet these requirements. We verified its aptitude in autonomous walking experiments using a pre-programmed controller. Although there is room for improvement in the design, the robot was able to walk, fall and stand up without human intervention for 8 hours, during which it made over $43,000$ footsteps. This section is based on (Schuitema et al., 2010b).

---

[1] The Delft Biorobotics Laboratory

3.1.1 Introduction

To be able to model a learning task of a real robot as an MDP, the robot and its environment need to meet certain requirements. For example, it should be possible to obtain a state signal that has the Markov property. In addition, the learning process itself – specifically the process of exploration – can be very straining for the hardware. Most robotic hardware lacks the robustness to withstand a large number of learning trials with random exploration. Although the limitations of real hardware are known by experienced researchers, we found no publication that explicitly maps the essential requirements of the RL framework onto hardware and software requirements. This makes it difficult to develop robots suitable for RL. Therefore, this section reports the design of prototype 'Leo', see Figure 3.1, that we explicitly developed for online, autonomous RL research. Leo is a 2D biped (two-legged) robot. We selected the bipedal walking motion as our example task for two reasons. It is a complex, challenging task, and, we have extensive experience with it from previous work (Collins et al., 2005).

In Section 3.1.2, we derive system requirements from the fundamentals of RL. A system overview of Leo is provided in Section 3.1.3, followed by detailed hardware and software requirements and their implementation in Section 3.1.4 to 3.1.8. We present our conclusions in Section 3.1.9.

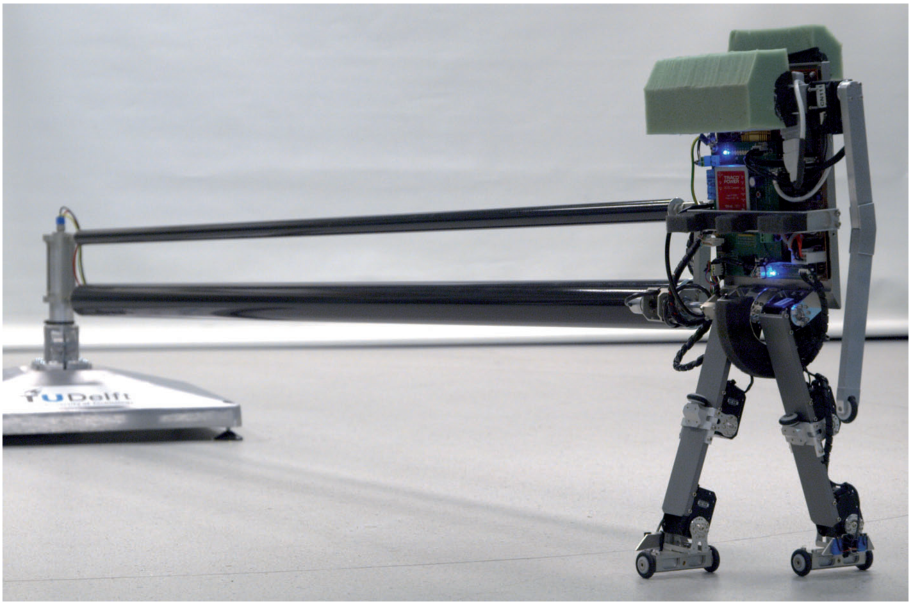


Figure 3.1: *Leo: a 2D walking robot designed for Reinforcement Learning. Leo is equipped with seven motors (hips, knees, ankles and shoulder) and is connected to a boom construction that provides power and lets it walk in circles. It is allowed to fall, after which it stands up autonomously.*

3.1.2 Reinforcement Learning requirements

The concept of RL (see Chapter 2) has important implications for the hardware and software design of a robot that is to be controlled online by RL. Here we summarize the key properties of RL and the robot design requirements that follow from it. The resulting requirements will be numbered for easy reference and will be further explained in the subsequent chapters.

Robustness

A key aspect of on-line RL is that the learning agent *explores*, i.e., a (small) part of its actions consists of suboptimal, often random actions with the goal to explore new terrain and to gain new knowledge. Although exploration can be guided, it can in principle lead to system damage. For a humanoid robot, the cause is usually a fall or a self-collision. From simulation of a biped with a complexity comparable to Leo (Schuitema et al., 2005; Troost, Schuitema, and Jonker, 2008), we know that learning a walking behavior from scratch by controlling two hip motors takes approximately 20 hours or more (without counting the time required to stand up) and hundreds of falls. This leads to the following requirement:

1. The robot can walk over a period of days and is robust against falls and self-collisions.

In Section 3.1.4, we discuss how this requirement has influenced the design of our robot.

State transitions

The common approach in RL is to model the process of learning a task as a Markov Decision Process (MDP) with discrete time steps $k \in \mathbb{N}$ and sampling period h, see Section 2.1. The MDP defined as the 4-tuple $\langle S, A, T, R\rangle$. Here, S is a set of states and A is a set of actions. The state transition probability density function T defines the probability density over S for the next system state $s_{k+1} \in S$, reached after executing action $a_k \in A$ in state $s_k \in S$. The reward function R defines the scalar reward of a state transition as $r_{k+1} = R(s_k, a_k, s_{k+1})$. The goal of the learner is to find a control policy that maps states to actions and that maximizes the expected cumulative sum of rewards from R.

An MDP has the *Markov property*, which means that T and R only depend on the current state-action pair and not on past state-action pairs, nor on information excluded from s. The on-line learning robot will experience a stream of events of the form $s_0, a_0, r_1, s_1, a_1, r_2, s_2, ...$ For the Markov condition to be true, every observation $[s_k, a_k, r_{k+1}, s_{k+1}]$ has to be in accordance with T at any moment during a learning experiment, which might take days. In this period, T must be stationary. This leads to the following robot design requirements:

2. The robot can observe state s, which holds all information relevant to the learning problem.

3. The effect of action a in every state s is predictable.

4. The sampling time is constant.

5. T must be stationary within a time frame of tens of hours.

In addition, within the MDP framework, a control action a_k that is based on state observation s_k is assumed to take place immediately after the observation itself, which poses an additional requirement:

6. The time between measurement s_k and action a_k is zero.

In a real robot, these conditions can only be met approximately. The system will suffer from sensor and actuator noise, finite accuracy in the periodic timing, and computational delays due to, e.g., running the learning algorithm. However, we made specific design decisions that should keep the violation of these requirements to a minimum. These are discussed in Section 3.1.5, 3.1.6, 3.1.7 and 3.1.8.

System complexity

For the learning system to discover the long term value of all actions $a \in A$ in all states $s \in S$, in principle, it should experience all actions in all states at least a number of times. This search space can be limited in several ways by changing the learning algorithm. Examples are restricting the allowed policies by parameterizing them (policy search and policy iteration methods), extracting recurring tasks by using hierarchies of learning tasks (Hierarchical Reinforcement Learning), and assuming that neighboring states and actions have related values (function approximation and generalization). However, all current RL learning algorithms have in common that they become intractable for very large state-action spaces. For robots, the state-action space grows exponentially in the number of degrees of freedom. Therefore, our robot design should meet the following additional requirement:

7. The robot's number of degrees of freedom, and thereby its state-action space, is limited such that learning succeeds within a reasonable time frame.

This requirement is further discussed in Section 3.1.3.

3.1.3 System overview

The robot was designed by checking all 7 requirements, which are explained in more detailed in Section 3.1.4 to 3.1.8. Although the complete system design is the final result (i.e., the conclusion) of Section 3.1, we provide the overview here at the start for ease of understanding.

Leo (see Figure 3.1) is small, approx. 50 cm in height, and light, approx. 1.7 kg. It has foam bumpers on both sides of the top of the torso and between the hip motors, thereby capable of taking numerous falls in a wide range of configurations. From simulation results on bipedal walking robot ‘META’ (Schuitema et al., 2005; Troost, Schuitema, and Jonker, 2008), we know that learning to walk can take place in an acceptable time frame of days for a robot with 7 degrees of freedom. Therefore, to comply with requirement 7, we designed our robot to have a number of degrees of freedom comparable to robot META. Leo has 7 servo motors (Dynamixel RX-28; max. torque 3 Nm): two in the ankles, knees and hips and one in its shoulder. The servo motors communicate with an embedded computer (VIA Eden 1.2GHz CPU and 1GB RAM) over RS-485 serial ports. They are capable of position control and voltage control, and can communicate their current position and temperature. While previous research has shown the added advantage of accurate torque control (Hobbelen, De Boer, and Wisse, 2008), which these servo motors cannot accomplish, they are commercially available, all-in-one, easily replaceable packages. Leo’s feet have pressure sensors that measure foot contact. Furthermore, Leo has an arm that enables it to stand up after a fall.

Leo is connected to a boom (length 1.72m) with parallelogram construction. This keeps the hip axis always horizontal, which makes it effectively a 2D robot and thus eliminates the sideways stability problem. The boom also supplies power to the robot and makes it walk in circles, which together guarantee long-term continuous operation. An encoder in the hip joint measures the absolute angle between the torso and the boom. The foot contact points can roll sideways, see Figure 3.1, which is needed to counter the effects of running in circles.

A wide variety of learning tasks can be conducted, ranging from learning a walking motion by actuating the two hip motors (keeping the ankles stiff; virtual constraints on the knees), to learning optimal ankle push-off, to learning a complete stand up behavior using all 7 motors.

To test the hardware, a controller was programmed (i.e., not learned) according to the limit cycle walking paradigm (Hobbelen, 2008). Typical walking strides of Leo controlled by this controller are shown in Figure 3.2 by means of the evolution of the angles of both hip motors and the torso.

3.1.4 Robustness

In order to comply with requirement 1, the robot should be able to walk over a period of days and be able to withstand falls and self-collisions. This has led to the following design choices.

By keeping the robot small, approximately 50 cm in height, the impacts during a fall are kept small as they scale more than cubically with the robot’s height. By choosing ’smart’ actuators with internal protection against overloading and overheating, the actuators are less likely to fail. In case they do fail, they are easy to replace. To reduce the impact on the torso during a fall, foam was placed on both sides of the top of the torso. A strong leaf spring with added foam between the two hip motors protects them at all possible hip joint angles. In the construction of previous robots, we used micro switches in the feet to detect foot

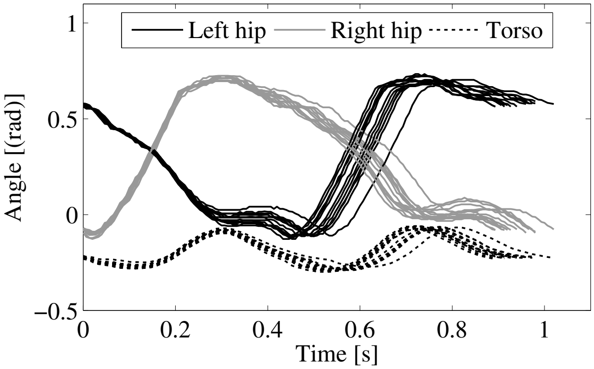


Figure 3.2: *The hip angles and torso angle for typical strides of Leo when using a limit cycle walking controller. The strides are aligned at left heel strike.*

contact, which were quite sensitive to failure. To increase the robustness of this design, we used pressure sensors in the feet, which reduced the number of moving parts.

Empirical verification and design improvements

The first tests with our robot exposed a number of weak mechanical links. The weakest links were the aluminium brackets that were bought with the servo motors, which broke several times. In the hips, we replaced these with custom designed brackets that had increased flange thickness (2.5mm) and were made of higher quality aluminium; in the knees, we improved the mounting of the bracket to the lower leg to decrease the chance of fatigue.

The single board computer formed a weak link as well. At each impact with the floor, the inertia of the heavy CPU cooler caused significant bending of the motherboard, which ultimately led to loose contacts in the motherboard area around the CPU. To solve this, we improved the mounting of the CPU cooler and the mounting of the motherboard to the robot.

After implementing the above mentioned changes, we tested the robustness of the robot by letting it walk using a pre-programmed (i.e., non-learning) limit cycle walking controller until it failed. The hip angle was controlled to a fixed reference angle using voltage control until the detection of the next heel strike. Upon foot contact, the back leg performed knee flexing while swinging forward. The torso was controlled to a reference angle as well. The robot made over $43,000$ footsteps and walked more than $6,000$ meters in a period of about 8 hours, of which 2.3 hours were spent to periodically let the motors cool down by resting on the floor. The robot fell 30 times (mostly automatically, to cool down) and stood up by itself afterward. The experiment stopped due to a rare software bug (that has now been fixed). Inspection of the machine revealed the failure of the potentiometer of an ankle motor. One foot sensor also stopped working. In addition, the gearboxes of the hip motors showed severe damage.

Although the operational lifetime of the robot is approaching the same order of magnitude as our requirement – days of autonomous operation – this goal is not met yet. To further increase the lifetime of the hardware, the potentiometers inside the servo motors (also see Section 3.1.5) were replaced by contactless magnetic encoders. To increase the lifetime of the gearboxes in the hip joints, torsionally flexible coupling elements were placed between the outgoing motor axle and the bracket to dampen high torques on the gearbox (see Section 3.3.2 for a more detailed explanation). These originate from the motor itself when large voltage differences are suddenly applied (a learning controller may do this regularly as exploratory actions) and from impact with the floor during walking and falling.

3.1.5 Reliable state information

To produce reliable and reproducible state information (requirement 2), the robot should be able to accurately measure its complete state and that of the environment. This section describes the robot's sensors.

Sensors

The Dynamixel RX-28 servo motors use a potentiometer to record the angular position. According to the specifications, positions can be measured in the range $[0, 300]^\circ$ with 10-bits accuracy (i.e., 1024 discrete position values can be observed), or 0.3°. To verify this, we compared the position readout from 30 servo motors with the readout from a 13-bit absolute magnetic encoder by means of a custom made calibration device. This revealed a large spread in linearity and range of the position information. Local deviations of up to 4° and linear deviations resulting in at most 10° have been observed. A few typical error plots are shown in Figure 3.3. In addition, at approximately 16% of the 1024 possible position values, the sensor reading would only change after an angular deviation of 0.6° instead of 0.3°, resulting in an effective accuracy of 9-bits at those positions.

While any (stationary) non-linearities in the position readout will not matter for the learning process - the learning agent simply maps states to actions and does not have a notion of absolute angles - it does pose a problem when a servo motor needs to be replaced. In that case, the learning agent would have to adapt to the new mapping from true angles to measured angles, which is typical for each motor. We addressed this by creating a lookup table for each servo motor, which maps the measured angles to the calibrated reference angles.

The servo motors also provide a velocity signal. However, we found the update rate of that signal to be 7.5Hz, which is too low for our purposes. Therefore,

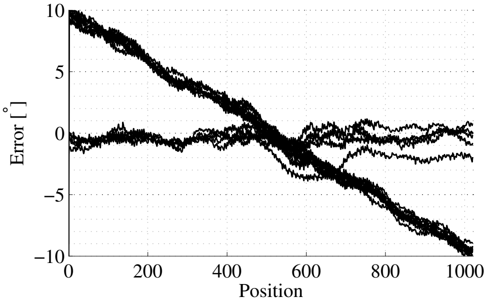


Figure 3.3: *Typical angle measurement errors from Dynamixel RX-28 servo motors from calibration with a 13-bit absolute magnetic encoder. While some motors merely showed local calibration errors, others showed an additional, systematic error that grows linearly with the position.*

we derive the velocity signal ourselves by differentiating the position signal and filtering the result with a third order Butterworth filter.

A 13-bit absolute magnetic encoder in the hip joint measures the angle between the torso and the parallelogram construction of the boom, which serves as our vertical reference.

The robot lacks a sensor that would measure its elevation. Its state is only fully determined from the other sensors if it is in contact with the floor with at least one point. Therefore, we have to verify during learning experiments that the robot touches the floor at all times.

The foot impact with the floor creates vibrations throughout the combined construction of robot and boom. Such vibrations are visible in the form of large state transitions, especially in the velocity state variables. The vibrations form a disturbance on top of the average dynamics of the robot and can make the learning problem harder. The initial boom construction with two $\varnothing 25 \times \varnothing 23.5$ mm carbon composite tubes was not stiff enough and caused large vibrations, which especially disturbed the torso angle measurements. The lower tube was replaced with a $\varnothing 55 \times \varnothing 53$ mm tube, which increased the stiffness of the construction. This reduced the vibrations and increased their frequency. When calculating joint velocities, we filter the differentiated position signal with a third order Butterworth filter with a low cutoff frequency (10Hz). This further reduces the effect of the vibrations on the velocity signals.

Environment

When the environment can be considered stationary, its properties are not part of the state signal. This requires the floor to be flat and level over the whole walking circle. We encountered problems with height irregularities in the floor; although they are small - several millimeters - they are significant in relation to the leg length of the robot. This is a disadvantage of our small robot design. Our tests have shown that the floor height variations have a significant negative impact on the walking behavior. Since the position of the robot within the circle is not measured (we do not want the robot to 'memorize' this particular floor), the floor height irregularities have to be treated as disturbances.

3.1.6 Actuation

The RX-28 servo motors provide position control, velocity control and an open loop actuation mode. Although the manufacturer describes the latter as torque control (literally denoted as “endless turn mode”), this mode is actually voltage control. Below we discuss the available choices for the primitive action – the control action of the RL agent.

Primitive actions

In the MDP framework, the learning agent chooses a primitive action from the finite set A. We chose the primitive action $\boldsymbol{a} = (U_1, U_2, \ldots, U_M)^T$ to be a vector of desired voltages U_i for each motor (i.e., with the motors operating in voltage control mode), with M the number of motors controlled by the agent. The voltages are kept constant over the next sampling period h. Each voltage U_i is chosen from a finite set of values, which is problem dependent. While Leo's actuators are servo motors, we chose not to use their internal position controllers, but to directly set desired voltages, as position control of the joints could lead to undesired behavior during walking. When the robot transitions from a single stance state to a double stance state, e.g., when it makes a footstep or when its foot accidentally hits the floor while swinging it forward, the extra foot contact results in additional constraints. When attempting to control the joints towards positions that are not in accordance with these constraints, the corresponding large forces could lead to abrupt and undesired movements (e.g., a foot could suddenly lose contact with the floor).

Especially in the double stance phase of the robot, when the system is over-actuated, position control of the joints could lead to undesirable behavior during walking.

For $\boldsymbol{a}$ to have a reproducible effect on the system in every state, some hardware properties need to be considered. Because of the backlash in each gearbox, the previous action determines whether the motor will directly exert force on the joint, or that the gears have to first bridge the backlash. Depending on the sampling period h and the new action, the time needed to bridge the backlash is variable and may be relatively large or small with respect to h. Although this topic deserves further attention to see how backlash exactly influences state transition probabilities, we did not further investigate the matter. The backlash we observed in Leo's motors was approximately 0.5°. The second hardware property is the temperature dependence of the DC motor, which we discuss in more detail below.

Temperature compensation

Initial experiments with the prototype using the pre-programmed walking controller showed that the robot's walking gait – initially stable enough for the robot to make 50 or more footsteps – would change significantly over a time span of 30 minutes, up to a point where the robot would typically fall after 5 footsteps. During this time span, the motors would reach a temperature of approximately 75°C. Allowing the motors to cool down to room temperature would restore the robot's initial stable walking behavior. This revealed the need for a better understanding of the effects of the temperature of the DC motors on the robot's actuation.

The motor torque at a given voltage U depends on the temperature of the motor, which is measured inside the RX-28. While temperature effects are probably not noticeable when the motor uses its internal control loop for position or velocity control, this effect is important in voltage control mode. In our robustness test, the pre-programmed controller was based on voltage control.

To comply with requirement 3, we compensated for the temperature dependency. We used the following model (omitting gear box friction) for the output torque τ as a function of the voltage U:

$$\tau(U) = K_\tau G \frac{U - K_\tau G \omega}{R} \tag{3.1}$$

with K_τ the motor's torque constant, R the winding resistance, G the gearbox ratio and ω the joint angle velocity. However, R and K_τ are temperature dependent. The winding resistance R will change with temperature θ according to

$$R(\theta) = R_{\text{ref}}(1 + (\theta - \theta_{\text{ref}})\alpha_{\text{Cu}}) \tag{3.2}$$

with α_{Cu} the thermal resistance coefficient of copper. The torque constant K_τ will change according to

$$K_\tau(\theta) = K_{\text{T,ref}}(1 + (\theta - \theta_{\text{ref}})N_{\text{Br}}) \tag{3.3}$$

with N_{Br} the temperature dependency of the magnetic flux density of the permanent magnets. The output torque τ can now be written as a function of U and θ:

$$\tau(U, \theta) = K_\tau(\theta) G \frac{U - K_\tau(\theta) G \omega}{R(\theta)} \tag{3.4}$$

For voltage control, we can keep the generated torque at a given voltage U_{ref} constant by supplying a corrected, temperature dependent voltage $U_{\text{corr}}(\theta)$. Its formula can be obtained by solving the following equation for U_{corr} using (3.2), (3.3) and (3.4):

$$\tau(U_{\text{corr}}, \theta) = \tau(U_{\text{ref}}, \theta_{\text{ref}}) \tag{3.5}$$

This results in:

$$U_{\text{corr}}(\theta) = U_{\text{ref}} \frac{K_{\text{T,ref}}}{K_\tau(\theta)} \frac{R(\theta)}{R_{\text{ref}}} + \omega K_{\text{T,ref}} G \left(\frac{K_\tau(\theta)}{K_{\text{T,ref}}} - \frac{K_{\text{T,ref}}}{K_\tau(\theta)} \frac{R(\theta)}{R_{\text{ref}}} \right) \tag{3.6}$$

The DC motor inside the RX-28 was identified as the Maxon RE-max 17, type 214897. We used catalog values for the motor parameters: $K_\tau = 9.92 \cdot 10^{-3}\text{Nm/A}$ and $G = 193$. We used $\alpha_{\text{Cu}} = 3.93 \cdot 10^{-3}\text{K}^{-1}$. We chose $N_{\text{Br}} = 0$, because its true value is unknown and relatively small compared to α_{Cu}. Detailed experiments on individual motors can further improve the model and its parameters.

We tested the temperature compensation by letting Leo walk using the preprogrammed controller. With compensation according to (3.6), we observed qualitatively similar walking behavior in the temperature range of 45°C to 70°C. Above 70°C, the robot would start to fall more frequently. Without compensation, the robot noticeably fell more frequently at much lower temperatures. We can conclude that temperature compensation is important when using voltage control, and that there is room for improvement in our model (parameters).

3.1.7 System invariability

The MDP framework is defined for a stationary state transition function T. For this to be true for a robotic learning task, the robot and its environment should be invariable during the length of a learning experiment (requirement 5), or at least change slowly enough so that the learning system is able to adapt in time to the changing situation.

To verify the robot's invariability over time, we performed the following test during the robustness experiment described in Section 3.1.4. After every 20 minutes of walking, we recorded a two minute data set Ψ_t of state measurements and actuation patterns: $\Psi_1, \Psi_2, .., \Psi_{16}$. We used half of each data set, $\Psi_{t,m}$, to build a state transition model using Local Linear Regression (LLR) (see, e.g., (Rencher and Schaalje, 2000)), and the other half, $\Psi_{t,v}$, for validation. If the system behavior does not vary over time, or changes slowly enough, the model built from any $\Psi_{t_1,m}$ should be able to predict state transitions from any other data set $\Psi_{t_2,v}$.

To calculate a measure for the predictive power of data set $\Psi_{t_1,m}$ with relation to data set $\Psi_{t_2,v}$, we did the following. For every state transition $(s_{k+1}|s_k, a_k)$ from $\Psi_{t_2,v}$, the 40 nearest neighbors around (s_k, a_k) in $\Psi_{t_1,m}$ were used to build a local linear model. This model was then used to produce prediction $(\hat{s}_{k+1}|s_k, a_k)$. The root mean squared prediction error $\text{RMSE}(\Psi_{t_2,v}|\Psi_{t_1,m})$ over the total validation set (of N samples) then served as our measure:

$$\text{RMSE}(\Psi_{t_2,v}|\Psi_{t_1,m}) = \sqrt{\frac{\sum\limits_{k=0}^{N-1} ||\hat{s}_{k+1} - s_{k+1}||^2}{N}} \tag{3.7}$$

This error can be compared with the cumulative prediction error from simultaneously recorded data by calculating $\text{RMSE}(\Psi_{t_k,v}|\Psi_{t_k,m})$. In Figure 3.4, $\text{RMSE}(\Psi_{t_2,v}|\Psi_{t_1,m})$ is plotted for all combinations of the 16 data sets that were recorded during the 8 hour test described in Section 3.1.4.

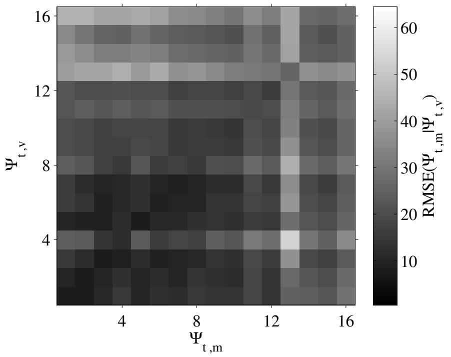


Figure 3.4: *Root mean squared error comparison between data sets recorded every* 20 *minutes during the robustness test. The RMSE generally grows when the time between the model data set and validation data set becomes larger, which indicates that the system is not time invariant. Data sets recorded at a later time generally produce larger RMSE values.*

From the results, two things become apparent. We can see that in general, RMSE($\Psi_{t_2,v}|\Psi_{t_1,m}$) increases when t_1 and t_2 are further apart. We can also see that data sets recorded near the end of the testing period generally produce larger cumulative error sums than data sets from the beginning of the experiment. In both cases, the difference is approximately a factor three. We must conclude that the system is not time invariant. Detailed inspection of the recorded data revealed that the position signal from the right ankle's servo motor was heavily deteriorating during the experiment (the signal showed more and more random spikes). This is the most likely cause of the observed time variance of the system. To avoid this type of problem, the potentiometers inside the servo motors were replaced by contactless magnetic encoders.

The noise in the graph can be (partially) explained by the difference in motor temperatures (the robot cooled down after every 30 minutes of walking), floor height variations, and by the fact that the robot would sometimes switch to a slower walking gait during a significant part of the data recording time. These sources of time variability can be decreased by improving the temperature compensation model and by decreasing the floor height variations. For the latter, we changed the robot's walking surface as described in Appendix A.

3.1.8 Real-time control

In the MDP framework, it is assumed that state transitions occur periodically, i.e., that the sampling time is constant (requirement 4). Furthermore, there should be no *control delay*, by which we mean the delay between the moment of measuring the state and the moment of acting upon that state (requirement 6). In reality, these requirements can only be approximately satisfied.

Periodic sampling

When the length of a sampling period is not constant, this means that control actions have varying length and therefore varying effect. This makes the learning problem harder, because the more variation in sampling period, the more learning experience is needed to learn the average effect of taking a control action in a certain state. Good periodic behavior requires hard real-time behavior. For Leo, we used Linux with the Xenomai real-time extension. On a sampling period of 6666.6μs, we measured an average standard deviation of 230μs (3.4%). Although this is generally a good result, the magnitude and frequency of allowable sampling time irregularities on RL are as of yet unknown.

Control delay

Control delay reduces the stability of a system and the possibility to accurately control it. Control delay is also not modeled in the MDP framework. The consequence of control delay for a RL system is that control actions take effect in future states, instead of immediately. However, the learning update is by default performed as if the control action was performed in the measured state. This can cause convergence problems (see Section 2.4 and (Schuitema et al., 2010a)).

To minimize the control delay in our robot, all processing is done on-board, which avoids transporting state and actuation signals over a network (transporting all sensor and actuator lines individually over the central boom joint is impractical). The RX-28 motors are controlled by a serial link, through which they are addressed one after another. Although this causes extra delay compared to a parallel connection, the serial bus needs only two data cables to address all motors. Despite the relatively high bus speed of 0.5 Mbaud, the process of requesting position data and sending actuation commands still takes several milliseconds. By operating two RS-485 serial ports in parallel, each port controlling the motors of a single leg, this time was roughly cut in half while avoiding extra cabling. The foot sensors and hip sensor are read out over a much faster Serial Peripheral Interface (SPI) bus, causing a small delay in the order of microseconds. In total, measuring the complete state of the system takes on average 1850μs with a maximum of 2350μs.

Delay in the context of RL has been studied, see e.g. (Walsh et al., 2009), but not specifically in the context of robotics. In Section 2.4, it is shown that a delay of less than a time step can already have a negative influence on the learning performance of a robotic system. Both studies show the benefit of extending the MDP framework to incorporate the delay. In Section 2.4, on-line temporal difference learning algorithms are proposed that improve the learning performance of a robotic system suffering from delay, while maintaining low computational complexity.

Software

Leo's software architecture is event based. The controller software is separated from the rest of the software. The robot produces state information at regular intervals, to which a controller (and also a logger, visualization program, etc.) can subscribe. The state information producing loop is the only periodic loop in the system; controller calculations, actuation and logging are purely reactive because the system is event based. The periodic state measurement loop does not wait for actuation signals, which guarantees real-time periodic state information for all its subscribers. The robot provides an actuation interface to the controller, containing functions that can change the actuation signals and return key properties of the actuators.

Because of the abstraction of the state information interface and actuation interface, these interfaces can be replaced by simulated versions. As a result, the controller code can be used both on the robot and in simulation *without modification*, which facilitates development. In addition, any controller, whether pre-programmed or learning, can be connected to the state information and actuation interfaces and can be replaced on-line. For example, during the process of learning to walk, a pre-programmed stand up behavior can seamlessly take over control after a fall. More detailed information about the software architecture can be found in Section 3.2.

3.1.9 Conclusions

With Leo, we designed and built a bipedal walking robot specifically for online, autonomous Reinforcement Learning experiments. Due to its boom construction which makes it run in circles and supplies power, its fall protection and its ability to stand up by itself, LEO can perform RL experiments without human assistance.

We checked the robustness of the system by letting it walk for 8 hours using a pre-programmed controller, during which it made $43,000$ footsteps and fell 30 times before failing. Although this is an amount of effort comparable to that needed for a learning experiment, it was desirable to further improve the robustness. To this end, we replaced the position recording potentiometers in the actuators with contactless magnetic encoders and added torsionally flexible coupling elements to the joints to reduce gearbox damage.

We made the actuation more predictable by compensating for the temperature of each motor. We verified the invariability of the system over a period of 8 hours by periodically building a model from measured state transitions, and validating this model with measurements recorded later. This showed that the system is not completely time invariant, which was mostly caused by a deteriorating position recording potentiometer in one of the actuators.

We discussed the timing characteristics of our real-time control loop. The control delay between measuring the state and actuating the motors is significant and negatively influences learning performance in simulation. Therefore, the framework of constant delay MDPs is more adequate for online RL experiments on this robot, which requires different learning algorithms.

3.2 Software design and simulation

In this section, we present a motion control framework that is designed to facilitate the research on Reinforcement Learning (RL) for real humanoid robots. By means of abstract interfaces for state observation and actuation, it facilitates writing C++ controller code that can be tested in simulation first and then run on the real robot without modifying the code. It provides real-time compatible implementations of common temporal difference RL algorithms. We present results on the application of this framework to our prototype Leo.

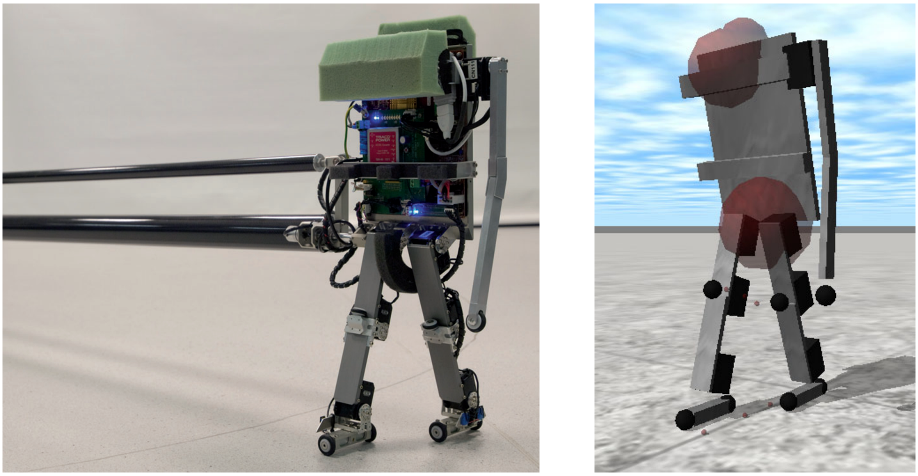
3.2.1 Introduction

The programming of humanoid robots is a challenging task. While the complexity of the hardware of humanoid robots continues to grow, writing behavioral software for them also becomes increasingly complex. Reinforcement Learning is one of the promising paradigms to reduce this programming effort. Unfortunately, the number of successful real world applications of Reinforcement Learning on robots lags far behind theoretical and simulation results. In this work, we collect the software requirements for doing efficient research on Reinforcement Learning techniques for real robots. We deduce these requirements from the RL theory as well as from our own experience with initial implementations on a real robot. While most required parts of the software are publicly available as separate modules, to our best knowledge no current project integrates all of them. In our implementation, we reused software libraries whenever possible for tasks such as physics simulation and XML interfacing, and complemented it with new code such as the real-time implementation of RL algorithms.

We tested our framework on prototype Leo, see Figure 3.5. At its core is a single board computer with a 1.2GHz VIA Eden CPU and 1GB of RAM, running Linux with the real-time Xenomai extension (Gerum, 2004). All sensors and actuators are interfaced through this single board computer. We present Leo's dynamic system model and compare the real robot with its simulation, as well as the timing characteristics of the robot.

3.2.2 Software requirements

Reinforcement Learning often requires many trials, in which the robot will inevitably make mistakes, which can damage the hardware. Learning in a realistic simulation can speed up the research by saving hardware maintenance time and



(a) Robot Leo (b) Simulation of Leo

Figure 3.5: *Leo: a 2D walking robot suitable for on-line RL research.*

by running several tests in parallel, often faster than real-time[2]. Because creating learning controllers can result in complex code, efficient testing of such code requires 1-on-1 controller code sharing between the real robot and its simulation. This leads to the following requirements (which are essential, though not unique for RL robots):

1. A realistic simulation of the robot and its environment can easily be created and modified.

2. Controller code works in simulation as well as on the real robot without additional modifications.

When the robot's dynamics change due to wear and tear, e.g., frictions change, parts slightly bend or actuator behavior changes over time, the learning problem might change significantly or even become impossible. Continuing the learning problem can overwrite and invalidate previous learning data and even damage the machine. As it is not always possible (or desirable) for a human to continuously monitor this process, the machine should be able to detect anomalies and call for maintenance, which leads to the following requirement:

3. The software architecture facilitates the incorporation of a self-diagnostics module that monitors the system dynamics.

---

[2] Here, real-time means that time in the simulation progresses at an equal pace with the wall-clock time, i.e., the real world time experienced by humans.

The common framework for RL is that of the Markov Decision Process (MDP) (see Section 2.1). The learning system is modeled as an MDP with discrete time steps labeled $k = 0, 1, 2, \ldots$. At every time step, the system transitions from state s_k to s_{k+1} by taking control action a_k. For an MDP to be constant (i.e., time invariant), every observation s_{k+1} should only depend on the previous observation s_k and action a_k, while a_k should take place simultaneously with s_k. This means that the sampling time is constant (shorter or lengthier control actions will result in different s_{k+1}). Interruptions of the control loop are especially detrimental in dynamically unstable systems, which may not be able to compensate with subsequent actions. There also should be no delay between a_k and s_k, even though a_k is the result of calculations based on s_k. In practice, on a real system these assumptions are only met approximately. From Section 2.4, we know that the larger the delay between s_k and a_k, the more difficult the learning process becomes. However, knowledge about the length of the delay can improve the learning performance again. This leads to the following software requirements:

4. The sampling period is constant, ergo, the system is real-time.
5. The delay between measurement s_k and control action a_k is minimal.
6. The delay between measurement s_k and control action a_k is measurable.

Each learning problem requires different state information (i.e., a different set of sensor inputs), different learning parameters, rewards and a different training environment. It should be easy to modify and test such settings, which leads to the following requirement:

7. The robot's RL problem can be easily defined and modified without recompilation.

Note that in principle, communication middleware is not required for a RL motion control system. However, when the gathering of sensor data for state s_k requires addressing multiple sources over various communication protocols, or when it is desirable to remotely monitor the state of the robot, such middleware can provide a solution.

3.2.3 Related Work

There is an abundance of software frameworks and toolboxes for robot control. Some only target the communication infrastructure (two prominent examples of the real-time and non-real-time variety are OROCOS RTT (Bruyninckx, Soetens, and Koninckx, 2003) and ROS (Quigley et al., 2009), respectively), providing a well-defined way for the various components in a robot system to communicate with each other. Others, such as Player (Collett, MacDonald, and Gerkey, 2005), are mainly concerned with sensor and actuator abstraction, allowing controllers to become platform-agnostic. These two are often combined, like in YARP (Metta, Fitzpatrick, and Natale, 2006). The platform independence can then be further exploited to allow a transparent interface to a simulator such as Stage (Vaughan, 2000), Gazebo (Koenig and Howard, 2004) for Player or Webots (Michel, 2004) for URBI. Finally, there exist projects integrating these systems with additional libraries for signal processing, visualization or user interfacing (RoboFrame (Petters, Thomas, and Von Stryk, 2007)) and higher-level scripting (OpenRAVE (Diankov and Kuffner, 2008), URBI (Baillie, 2004)).

Learning systems are much scarcer. While it is easy to obtain implementations of specific algorithms (FANN (Nissen, 2003)) or algorithmic classes (Reinforcement Learning Toolbox (Neumann, 2005)), they lack integration with (generalized) simulators or robot interfaces, reducing their portability. PyBrain (Schaul et al., 2010) is an exception, using the XML-based XODE (Denniss, 2004) language to interface with the ODE (Smith, 2011) dynamics engine. However, since PyBrain is written in Python, it cannot be used for on-line learning on a dynamically unstable system such as a two-legged robot, as the real-time requirements for motion control are broken by the garbage collector.

We can observe that the available software is mainly concerned with control at a higher level than basic motion. This make sense, because there lies the largest integration effort for most projects. However, for our particular case of Reinforcement Learning for motion control of unstable systems, which requires both hard real-time behavior (requirement 4) as well as transparent portability between simulation and the real system (requirement 2), we needed to come up with our own solution. It is not intended to replace the higher-level frameworks or provide the breadth of lower-level algorithm libraries, but rather to integrate with them and perform the specific task of on-line learning of motion control.

3.2.4 Implementation

Here we discuss the C++ implementation of our motion control framework and its integration with existing software libraries.

Motion control architecture

The architecture of the motion control framework has three main classes; the `State Transition Generator` (`STG`) that produces state information, the `Actuation Service` that provides an interface to the actuators, and the `Policy Player` that selects and executes a control policy. The `STG` and the `Actuation Service` are abstract interfaces that form the abstraction layer between control software and the robot – whether real or simulated. A UML diagram of the software architecture can be found in Figure 3.6.

The `STG` is responsible for producing periodic state information, either by polling hardware sensors or by computing the next state in simulation. By means of a publish/subscribe architecture (also found in, e.g., (Neumann, 2005)), state information events are sent to a set of subscribers (derived from the `STG`

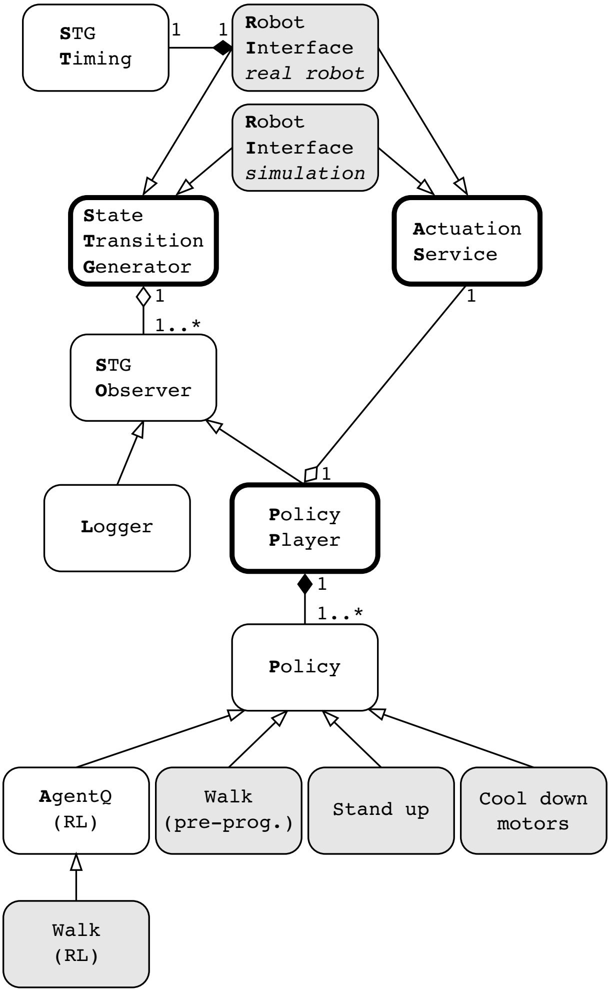


Figure 3.6: *Software architecture of the motion control framework. The* `State Transition Generator` *sends periodic state information events to a collection of* `STG Observers`*, such as the* `Policy Player` *and a* `Logger`*. The* `Actuation Service` *provides an interface to the robot's actuators. The* `STG` *and* `Actuation Service` *together form a* `Robot Interface`*, which can be implemented for a simulation and for a real robot. Its timing is monitored by the* `STG Timing` *class. The* `Policy Player` *selects a* `Policy` *from a collection and executes it by passing the state information from the* `STG` *and by exposing the* `Actuation Service`*. An arrow points to a base class (inheritance), a black diamond points to a container class (composition) and a hollow diamond points to a container/'user' class (aggregation). White blocks represent generic classes, grey blocks represent robot specific implementations.*

`Subscriber` class), the most important one being the `Policy Player`. Other possible subscribers are logging and visualization clients. On a real robot, state events are generated on a real-time periodic basis. In simulation, they are generated directly after computing the next control action. State information events are transported using POSIX queues, because these are available on a number of platforms, including Linux with Xenomai extension.

The `Actuation Service` is responsible for exposing a generic interface to the robot's actuators, which are either simulated or real hardware actuators. Through the service, the actuator signals can be changed at any time (i.e., instantly, on demand), as opposed to the often used construction of actuating the system at a fixed time, such as the start or end of the sampling period. Actuation on demand minimizes the delay between observing the state and acting upon it and thereby fulfills requirement 5. On the other hand, actuation does not need to occur after every state information event, but can be done, e.g., every second or third sample. In this way, a controller can collect state information more frequently than it needs to actuate the system, which can be useful for filtering and event detection. It also facilitates switching between controllers that need to actuate at different frequencies while letting the `STG` run at a constant frequency.

The `Policy Player` is responsible for controlling the robot. It contains a number of policies, as well as decision logic to switch between them. Based on the robot state and the applicability of the policies (not all policies can be executed in every state, e.g., a walking policy is not available when the robot fell down), it decides which policy should be in control at any given moment. That policy then receives the current robot state and is given access to the `Actuation Service` to control the actuators. Policies can be pre-programmed or derived from one of our provided TD learning policies. By adding several controllers to the `Policy Player` and programming the switching logic, the robot can switch at run-time between, e.g., a RL controller that learns to walk and a pre-programmed controller that lets the robot stand up. The policy player can monitor the health of the system and, if necessary, switch to a fail-safe controller in order to meet requirement 3 (facilitation of self-diagnostics). Because the state information events and the actuation interface do not contain properties specific to hardware or simulation, control policies can be run on both the robot and in simulation, which fulfills requirement 2.

When an `STG` and an `Actuation Service` are implemented in a single interface, we speak of a `Robot Interface`. On a real robot, the most important timing characteristics of this interface, such as the mean and standard deviation of the sampling period, actuation period and actuation delay, can be automatically measured by means of a generic `STGTiming` class, which fulfills requirement 6.

All base classes of the motion control framework can be compiled for Windows, Linux and Linux with Xenomai extension (the latter being important for requirement 4 of supporting real-time systems), 32-bit and 64-bit variants.

Reinforcement Learning

Reinforcement Learning policies can be created by deriving from the generic `AgentQ` class, see Figure 3.6, which is a real-time implementation of temporal difference learning[3] where real-time means that the policy's execution time is bounded and not interrupted by, e.g., system calls.

The `AgentQ` class supports common TD learning variants such as Q-Learning, SARSA, R-Learning (Sutton and Barto, 1998), dQ and dSARSA (see Section 2.4.3). Additionally, Hierarchical Reinforcement Learning is supported in the form of MAXQ-Q (Dietterich, 2000). To approximate the Q-function, we provide a real-time implementation of tile coding (see Section 2.3.4), for which we optimized execution time and memory access latency in order to reduce its computational footprint on the robot's embedded computer. An `AgentQ` instance can be configured at run-time to set the desired algorithm, learning rate, discounting factor, exploration rate and eligibility trace discounting factor. For tile coding, the maximum memory size and number of tilings can be specified, as well as the memory initialization parameters. Classes inheriting from `AgentQ` can make additional elements of the learning task configurable, such as the height of the implemented rewards, the size of the tiles in each dimension and so on. If the `STG` supports sensor and actuator lookup by name (currently only supported in simulation), it is also possible to specify the sensors and actuators that form the state-action space of the learning algorithm. Configuration is done by means of an XML configuration file, which is discussed in more detail below.

Configuration In order to meet requirement 1 and 7, all configurable aspects of the system, such as the Robot Interface, the Policy Player and its policies, are stored in an XML file that is read and processed in the initialization phase. The configuration data is made accessible by the `Configuration` class, see Figure 3.7. It supplies `ConfigSection` interfaces to data collections, which in their turn supply `ConfigProperty` interfaces to individual data values, such as integers, booleans, floating point values and strings, as well as `ConfigSection` interfaces to nested data collections. Data sections and properties are identified by a unique text string. The `Configuration` class is a generic interface, for which we implemented a realization for XML files using the TinyXML library. In this construction, it is easy to support multiple file formats while exposing a single, stable configuration interface to the framework by means of the `ConfigSection` and `ConfigProperty` classes. The XML configuration format is illustrated in Listing 3.1. A special `include` tag allows sub-hierarchies to be read from ancillary files that contain shared configuration data, for example a robot configuration that is used in different learning environments. In a separate `constants` section, numerical constants can be defined for use in mathematical expressions later in the

---

[3] We plan to support more RL algorithms using external libraries, but these are usually not written with real-time processing in mind.

document. We employ the MUPARSER (Berg, 2010) library to evaluate mathematical expression in the configuration. Defining constants and using mathematical expressions greatly facilitates the parameterized definition of robots, environments and learning settings.

Listing 3.1: *Example XML code used to configure the simulation environment and the controller parameters. The code is pre-processed to substitute `<include>` tags and to resolve mathematical constants and expressions.*

```
<constants>
  <euler_e>2.718281828459</euler_e>
  <ctrlFreq>30.0</ctrlFreq>
  <tauTask>4.0</tauTask>
   ...
</constants>
<openDynamicsEngine>
  <samplingperiod>1.0/ctrlFreq</samplingperiod>
  <include>
    <filename>robotconfiguration.xml</filename>
    <path>leo/</path>
  </include>
   ...
</openDynamicsEngine>
<policy>
  <algorithm>sarsa</algorithm>
  <memorySize>1024*1024*8</memorySize>
  <numTilings>16</numTilings>
  <alpha>0.20</alpha>
  <epsilon>0.05</epsilon>
  <gamma>euler_e^(-1.0/(ctrlFreq*tauTask))</gamma>
   ...
</policy>
```

Logging The logging of messages to both screen and disk is essential in controller development for robots. To guarantee 1-on-1 controller code sharing between the robot and its simulation, an abstract logging base class was created that uses standard output streams on non real-time operating systems while using the RTDK real-time printing library on Xenomai.

Simulation

The heart of our simulation environment is formed by the OPEN DYNAMICS ENGINE (ODE; (Smith, 2011)), a fast and stable rigid body dynamics simulator.

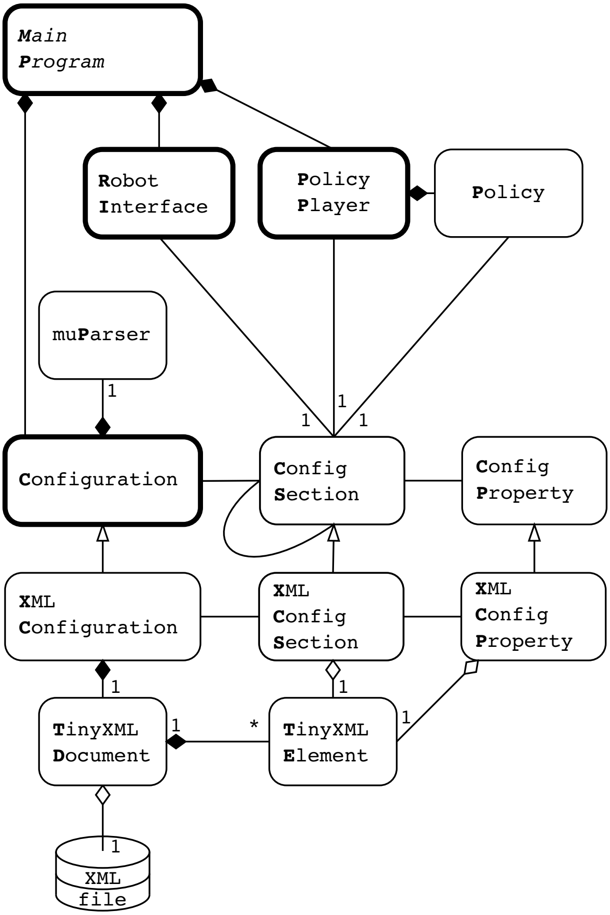


Figure 3.7: *Software architecture of the configuration framework. The* `Configuration` *class provides an interface to a single configuration file by providing* `ConfigSection` *interfaces to data collections, which in their turn provide* `ConfigProperty` *interfaces to individual data as well as* `ConfigSection` *interfaces to nested data collections. The* `Robot Interface`*, the* `Policy Player` *and the policies are configured by passing a* `ConfigSection` *reference. We implemented an XML realization of the* `Configuration`*,* `ConfigSection` *and* `ConfigProperty` *classes through the* TinyXML *library. An arrow points to a base class (inheritance), a black diamond points to a container class (composition) and a hollow diamond points to a container/'user' class (aggregation). Line segments indicate association, such as object creation or reference passing.*

We implemented the `STG` and `Actuation Service` interfaces on top of ODE to integrate it in our motion control framework. Furthermore, we added XML configurability through the `ConfigSection` interface for the robot's physical configuration, its initial condition and other simulation parameters in order to fulfill requirement 1. We use an XML interface similar to existing ones (Koenig and Howard, 2004; Denniss, 2004).

Basic 3D visualization of the simulation is automatically available by means of a generic user-interface widget based on the DRAWSTUFF library (part of ODE). The visual representation of simulated objects is configurable in XML.

ODE servo motor model Servo motors are widely used in humanoid robots; our prototype Leo is actuated by servo motors as well. Therefore, we added a generic servo model to the simulation framework. We use the following simplified model for a DC motor with gearbox to calculate the joint torque τ resulting from the motor voltage U:

$$\tau(U) = K_\tau G \frac{U - K_\tau G \omega}{R} \tag{3.8}$$

with K_τ the motor's torque constant, R the winding resistance, G the gearbox ratio and ω the joint's angular velocity. The voltage on the DC motor is clipped to the region $[-U_{\max}, U_{\max}]$ with $U_{\max}$ the maximum allowed voltage.

Any torque on the joint accelerates its connected body parts, as well as the rotor of the DC motor. Although the DC motor's rotor inertia I_r is usually small, the gear box causes its effect to be equivalent to a differential inertia $I_{\mathrm{r,eqv}} = G^2 I_\mathrm{r}$, i.e., an inertia that is only affected by joint torques, not by torques on the attached body parts. The effects of I_r are modeled by adding a meta-object to the simulation in the form of a 'virtual disc', which has inertia, but no mass or geometry, and only one degree of freedom: its angular position. Its inertia is $I_{\mathrm{r,eqv}}$, angular position x_{vd} and angular velocity $\dot{x}_{\mathrm{vd}}$. A PD controller calculates the torque τ_{vd} necessary to adjust x_{vd} and $\dot{x}_{\mathrm{vd}}$ towards the joint's position x_{joint} and angular velocity ω:

$$\tau_{\mathrm{vd}} = K_{\mathrm{P,vd}}(x_{\mathrm{joint}} - x_{\mathrm{vd}}) + K_{\mathrm{D,vd}}(\dot{x}_{\mathrm{joint}} - \omega) \tag{3.9}$$

with adjustable parameters $K_{\mathrm{P,vd}}$ and $K_{\mathrm{D,vd}}$. The torque τ_{vd} is applied to the virtual disc, while $-\tau_{\mathrm{vd}}$ is applied to the joint (action equals reaction). The total torque applied to the joint as a function of U then becomes

$$\tau(U) = E_{\mathrm{gb}}[\tau(U) - \tau_{\mathrm{vd}}] - \tau_\mathrm{f} \tag{3.10}$$

in which E_{gb} is the limited gear box efficiency and τ_f is a friction term.

3.2.5 Results: bipedal walking robot Leo

In this section, we present the application of the proposed framework to our prototype, robot Leo. We discuss a comparison between the behavior of the real robot and its simulation, as well as the timing characteristics of the real robot.

Creating Leo's model in Open Dynamics Engine

The design of Leo (see Section 3.1 for an in-depth description of the hardware) was done in the CAD program Solidworks, which allowed the straightforward extraction of the most important mechanical properties of the machine's parts, such as mass, inertia and dimensions. A schematic drawing with model parameter definitions is shown in Figure 3.8; model parameter values are given in Table 3.1; locations of joints with respect to the body parts are reported in Table 3.2. Contacts in ODE are modeled as a stiff spring-damper combination between two intersecting (colliding) objects. We calibrated the simulation of foot contact with the floor – important for realistic simulation – by first setting the stiffness parameter such that the robot in rest would sink into the floor by a realistic amount. Next, we tuned the damping parameter such that the behavior of foot contact was qualitatively the same in the real robot and in simulation during a walking motion.

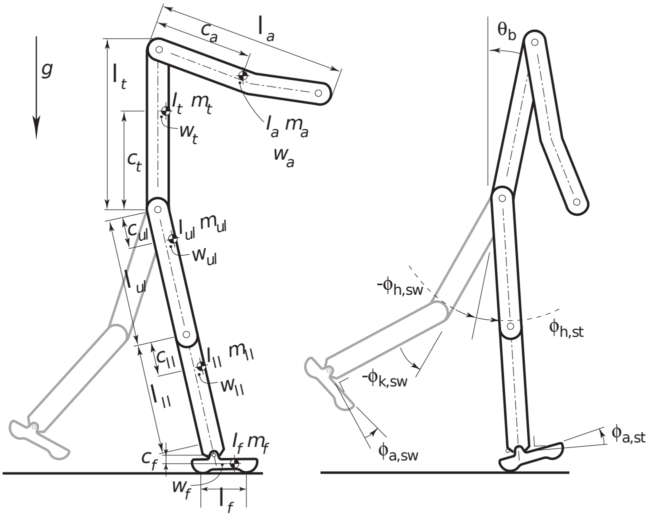


Figure 3.8: *Schematic overview of the two-dimensional 8-link simulation model of Leo. The left model shows the parameter definitions; parameter values are given in Table 3.1. The right figure shows the degrees of freedom of the model. The angle* φ_{torso} *is measured with respect to gravity; the other angles are relative joint angles.*

All the joints in Leo are formed by Dynamixel RX-28 servo motors (see also (Lima et al., 2009) on the modeling of the AX-12 servo motor). Besides the servo's internal position control loop, it allows an open loop "endless turn" mode, which is documented as torque control, but which is in practice (very close to) voltage control. We use this voltage control mode for both our pre-programmed walking controller as well as our learning controllers. We modeled the servo by using (3.10) with catalog values for the motor parameters (its DC motor was identified

Table 3.1: *Parameter values of the simulation model of Leo; the mass m, moment of inertia I, length l, vertical center of mass offset c and horizontal center of mass offset w of all body parts.*

	torso	torso + $m_{\mathrm{b,virt}}$	upper leg	lower leg	foot	arm	boom
m[kg]	0.913	1.231	0.180	0.127	0.073	0.095	0.860
I[gm^2]	4.68	8.71	0.273	0.153	0.0488	0.873	319
l[mm]	212	212	116	105	81	-300	1700
c[mm]	131	97	63	61	31	148	0
w[mm]	-1	0	3	8	0	14	835

Table 3.2: *Locations of the joints of Leo, expressed as horizontal offsets from the centers of the respective body parts.*

	horizontal offset	vertical offset
torso-shoulder[mm]	-5	–
torso-hip[mm]	3	–
lo.leg-knee[mm]	-4	–
foot-ankle[mm]	-9	–
floor-ankle[mm]	–	49

as the Maxon 214897), which are defined at room temperature. Because the servos typically operate at temperatures in the range $40 - 75°$C, the model quickly loses its validity and temperature compensation on the real machine is desired; see Section 3.1.6 for a derivation. To guarantee the same maximum torque at all typical temperatures, the maximum voltage available to any controller resulted in $U_{\mathrm{max}} = 0.76 * U_{\mathrm{supply}}$ with U_{supply} the power supply voltage. The friction parameters were tuned to qualitatively match the simulation with the real robot.

The boom construction, illustrated in Figure 3.9, is connected to the robot at the location of the hip axis and always keeps the hip axis horizontal, thereby effectively making Leo a 2D robot. Because its significant mass, it has additional effects on the robot that cannot be neglected in our simulation. By analyzing the dynamics of the boom, it can be shown that the boom's influence can be modeled by adding a virtual point mass to the torso. To model the effects of gravity, an additional external force needs to be modeled as well. The analysis is as follows.

When a force F is exerted by the robot onto the end of the boom, at distance l_{b} from the central pivot point and perpendicular to the boom, a torque τ is generated:

$$\tau = F l_{\mathrm{b}} \tag{3.11}$$

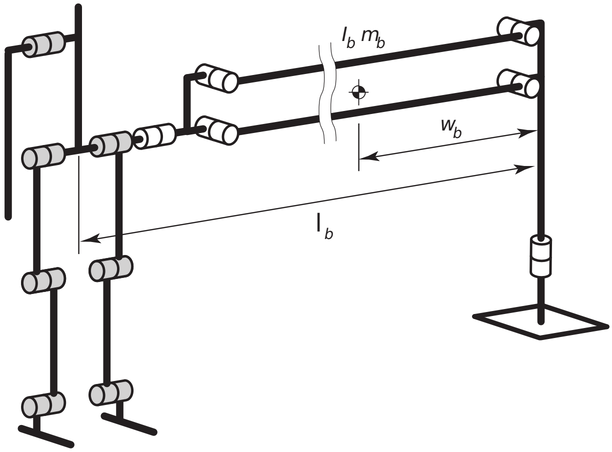


Figure 3.9: *Schematic overview of Leo and its boom construction, showing all degrees of freedom and the model parameter definitions of the boom.*

that causes an angular acceleration $\ddot{\theta}$ of the boom:

$$\tau = \ddot{\theta} I_{\mathrm{b}} = \ddot{\theta} \left(w_{\mathrm{b}}^2 m_{\mathrm{b}} + I_{\mathrm{b,w}} \right) \tag{3.12}$$

where the boom's mass m_{b} is centered at distance w_{b} from the central pivot point and $I_{\mathrm{b,w}}$ is the boom's inertia around the axis through its center of mass and perpendicular to the boom. The angular acceleration $\ddot{\theta}$ causes a linear acceleration $a = \ddot{\theta} l_{\mathrm{b}}$ of the robot. Combining (3.11) and (3.12) and using a gives

$$F l_b = \left(w_{\mathrm{b}}^2 m_{\mathrm{b}} + I_{\mathrm{b,w}} \right) \frac{a}{l_{\mathrm{b}}} \tag{3.13}$$

from which we can derive ($F = ma$) that the boom adds a virtual mass $m_{\mathrm{b,virt}}$ to the torso:

$$m_{\mathrm{b,virt}} = \frac{\left(w_{\mathrm{b}}^2 m_{\mathrm{b}} + I_{\mathrm{b,w}} \right)}{l_b^2} \tag{3.14}$$

Here we neglect the fact that vertical movement of the hip slightly changes the distance l_{b}. Because the boom construction is attached to the robot at the hip axis and does not rotate when the torso does, the rotational inertia of $m_{\mathrm{b,virt}}$ in the robot plane is zero and $m_{\mathrm{b,virt}}$ can therefore be modeled as a point mass added to the torso.

When considering gravity, the boom causes a force F_{g} at the hip location:

$$F_{\mathrm{g}} = m_{\mathrm{b}} \frac{w_{\mathrm{b}}}{l_{\mathrm{b}}} g \tag{3.15}$$

with g the gravitational acceleration. The modeled point mass $m_{\mathrm{b,virt}}$, however, will undergo a different gravitational force. The difference is compensated by modeling an external force F_{ext} on the torso (at hip location), which is always present:

$$F_{\text{ext}} = \left(m_{\text{b,virt}} - m_{\text{b}} \frac{w_{\text{b}}}{l_{\text{b}}} \right) g \tag{3.16}$$

Comparison between the robot and its simulation

To verify that Leo is indeed capable of walking, we implemented a pre-programmed (i.e., non-learning) limit cycle walking controller (Hobbelen, 2008) in our framework and tested it on the real robot and on its simulation. The controller code was exactly the same for robot and simulation. In Figure 3.10, we compare the evolution of both hip joint angles and the torso angle by plotting the mean and standard deviation of 15 typical subsequent footsteps of the robot, synchronized at left heel strike.

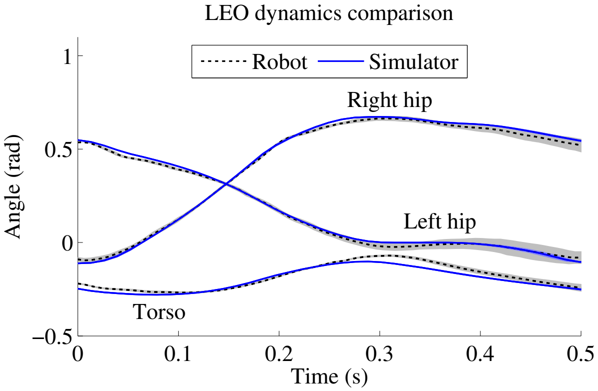


Figure 3.10: *Comparison between robot Leo and its simulation using the same pre-programmed limit cycle walking controller. Both hip angles and the torso angle are plotted against time, showing the mean and standard deviation of 15 footsteps, synchronized at heel strike.*

One can observe that the walking behaviors on the robot and its simulation are qualitatively similar in terms of the evolution of hip and torso angles. However, significant differences remain. Further calibration of the model parameters can improve the model, which is in our case only necessary if future (learning) controllers show qualitatively different behavior between robot and simulation.

Timing characteristics

In Section 3.2.2, we derived several timing requirements for on-line RL. We evaluated the timing characteristics on Leo in two situations: running a pre-programmed limit cycle walking controller and running a RL controller with the SARSA(λ) algorithm (see Section 3.3 for a thorough description of the controllers). The results are presented in Table 3.3 and show the timing statistics for the sampling period h, the actuation period h_{a} and the actuation delay T_{d}, calculated over a period of 10 seconds of typical operation. Because actuation can be performed on demand and is not compulsory every time step, h_{a} may differ from h, such as for the RL controller. All timing was measured using the generic `STGTiming` class. We can observe that the timing is on average accurately periodic, however, it contains significant but bounded jitter (minimum and maximum values were calculated over a period of several minutes). Further investigation pointed out that the serial communication link with the servo motors was the most important cause of the timing jitter. The actuation delay is variable, but measurable, and depends on the control algorithm.

Table 3.3: *Timing statistics of the sampling period h, the actuation period h_{a} and the actuation delay T_{d} for the pre-programmed limit cycle walking controller and the (preliminary) RL controller using SARSA(λ). The statistics were calculated over a period of* 10 *seconds of typical operation. State information was produced at 150Hz. The pre-programmed controller was actuated at 150Hz, the RL controller at 30Hz.*

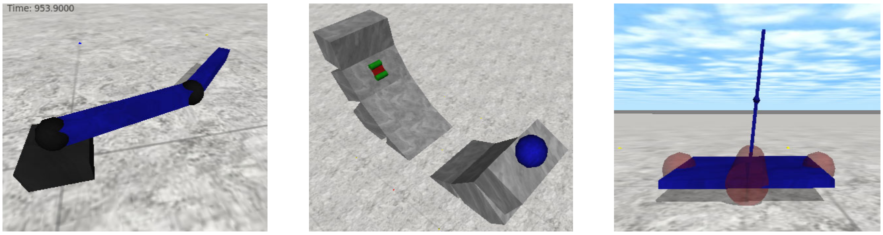
	Pre-programmed controller				RL controller			
	Avg	Stdev	Min	Max	Avg	Stdev	Min	Max
h [μs]	6667	229	6217	7116	6667	273	6213	7111
h_{a} [μs]	6667	239	6165	7165	33348	449	32825	42362
T_{d} [μs]	581	14	569	658	9538	373	584	9759

Other applications

The framework presented here has proved useful in the development of several toy problems that aid the research on Reinforcement Learning algorithms. An impression of the 'Two-link manipulator', the 'Mountain car', and the 'Pole balancing cart' projects can be found in Figure 3.11. Furthermore, the framework has been used for the software implementation of a new robot prototype (Karssen and Wisse, 2012).

3.2.6 Conclusion

We have derived the requirements for on-line Reinforcement Learning of motion control for humanoid robots, and created a framework that satisfies them. The provided simulation environment allows safe and realistic evaluation of controllers and is highly configurable via XML in terms of the robot's dynamics model, its environment and the learning controller. Controller code – newly created or based on the provided real-time implementations of temporal difference learning algorithms – can be shared 1-on-1 between the simulation and the real robot. Using a publish/subscribe architecture, state information is distributed to the controller



(a) Two-link manipulator (b) Mountain car (c) Pole balancing cart

Figure 3.11: *An impression of several Reinforcement Learning toy problems created with the presented framework.*

and additional modules such as logging and visualization services. The system is real-time periodic when run under Linux with the Xenomai extension, with both minimal and measurable control delay. We successfully applied our framework to the modeling and control of a bipedal walking robot, thereby verifying its characteristics.

3.3 Experimental RL results on Leo

The prototype 'Leo' that was described in Section 3.1 and 3.2 has served as a platform for a number of experiments, which are described in this section. First, learning from scratch on the real robot is demonstrated by learning a stairs step-up task in a simplified hardware setup, in which the torso was mounted onto a stand and only one leg was used. After this experiment, we implemented hardware changes to better protect the prototype against hardware damage. Next, we describe the MDP design for the task of learning to walk and present simulation results. We also verified that the prototype is capable of walking by using a preprogrammed controller. While the simulation results indicate that learning to walk from scratch is possible in a limited time span of several hours, the number of falls occurring in that process is considered to be prohibitively large for our prototype. Therefore, we explore a method in which the prototype starts the learning process with a suboptimal but functional policy, which it continues to improve using the same RL methods that were used to learn to walk from scratch in simulation. This avoids the largely explorative and harmful initial period of the learning process, but still allows us to study RL on the prototype.

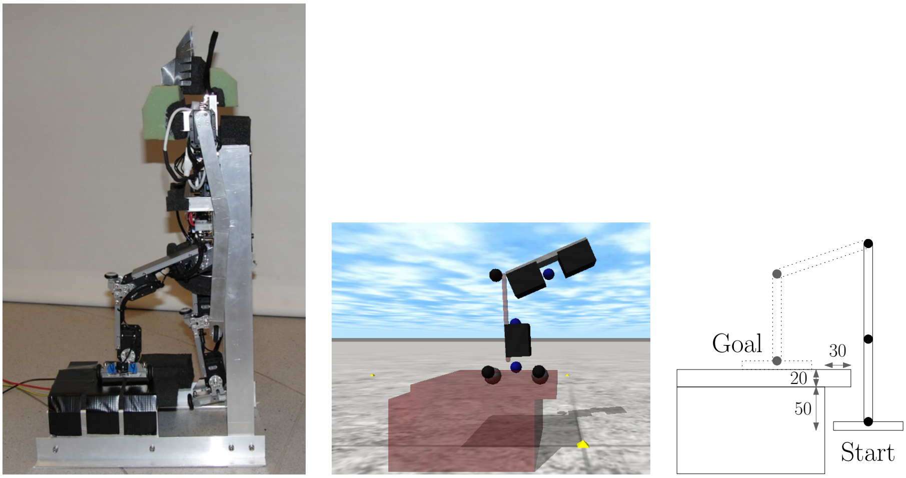
3.3.1 The stairs step-up task

Learning a task from scratch on Leo was first performed using a simplified hardware setup and a relatively simple task: to step up a stairs step with one leg. While our ultimate goal is to let Leo learn more complex tasks, such as walking and standing up, the stairs step-up task has a much smaller state-action space and is therefore more appropriate to test RL on the real robot. For the stairs step-up task, we mounted the robot's torso onto a stand with its feet hanging above the floor at all times, see Figure 3.12. The task was performed using one leg only. The hip and knee motor were controlled by the learning agent; the ankle motor was controlled by a pre-programmed controller to keep the foot perpendicular to the lower leg:

$$U_{\text{ankle}} = P_{\text{ankle}}(\varphi_{\text{ankle,ref}} - \varphi_{\text{ankle}}) \tag{3.17}$$

with $\varphi_{\text{ankle,ref}} = 0$ and $P_{\text{ankle}} = 10$. In this way, the controller acts as a virtual constraint.

The goal of the task is to place the foot onto the step as fast as possible. The task is episodic and ends when both the toe and the heel make contact with the step, which is detected by two separate contact sensors. To make the task non-trivial, the step has a protrusion that requires the robot to first move its foot backward before it can be lifted and placed onto the step. We used Q(λ)-learning with the ϵ-greedy policy (2.9), combined with tile coding function approximation. The MDP design, learning parameters and other details are discussed below.



(a) Robot Leo

(b) Simulation of Leo's leg

(c) Schematic overview (sizes in mm)

Figure 3.12: *Robot Leo, attached to a stand, which fixates its torso and keeps its legs above the floor at all times. Only the left leg is used to learn the stairs step-up task of placing the foot onto a stairs step as fast as possible, starting with a stretched vertical leg.*

State-action space

A Markovian state vector $\boldsymbol{s}$ was composed of the angle and angular rate of the hip, knee and ankle joint – a total of 6 dimensions:

$$\boldsymbol{s} = \begin{pmatrix} \varphi_{\text{hip}} \\ \dot{\varphi}_{\text{hip}} \\ \varphi_{\text{knee}} \\ \dot{\varphi}_{\text{knee}} \\ \varphi_{\text{ankle}} \\ \dot{\varphi}_{\text{ankle}} \end{pmatrix} \tag{3.18}$$

Although the ankle of the leg was controlled towards a fixed position using (3.17), because of the relatively low gain, significant deviations from $\varphi_{\text{ankle,ref}}$ were possible. In addition, the ankle angle has a large influence on whether the foot makes full contact with the step. Therefore, the state information on the ankle joint was included in the state.

We chose the primitive action $\boldsymbol{a}$ to be a vector of desired constant voltages for the hip and knee motor for the next sampling period h:

$$\boldsymbol{a} = \begin{pmatrix} U_{\text{hip}} \\ U_{\text{knee}} \end{pmatrix} \tag{3.19}$$

While Leo's actuators are servo motors, we chose not to use their internal position controllers, but to directly set the desired voltages. While position control could have been used for this task, it is less appropriate for the task of learning to walk, see Section 3.3.3. Since this experiment serves as a 'primer' for the task of learning to walk, we chose to use the same type of primitive actions. To summarize, the state-action space consists of 6 state dimensions and 2 action dimensions.

Function approximation

A tile coding function approximator was used to approximate the 8-dimensional Q-function. It contained 16 tilings, displaced according to (2.22). It generalizes only between states; Q-values for different actions are estimated independently. The tile widths used for all state variables, i.e., the generalization widths, can be found in Table 3.4. The feature parameters are trained in a gradient descent fashion according to (2.15) and (2.18). The memory required for storing the feature parameters was reduced through hashing, as described in Section 2.3.6. To stimulate exploration and to break ties at the beginning, we initialized $\hat{Q}(s,a)$ with random values between -1 and 1. Each time the experiment was repeated, the randomization was different.

Learning parameters

We discretized the voltage range $[-10.7, 10.7]$ of each actuator controlled by the agent into 5 discrete voltages, giving a total of $5^2 = 25$ elements in A. We chose

Table 3.4: *Tile widths (generalization widths) used in the tile coding function approximation of $Q(s,a)$ for the stairs step-up task of robot Leo. Q-values for different actions are estimated independently.*

	φ	$\dot{\varphi}$	U
Hip	$1.8 \cdot 10^{-1}$ rad	25 rad/s	-
Knee	$2.9 \cdot 10^{-1}$ rad	34 rad/s	-
Ankle	$3.8 \cdot 10^{-1}$ rad	42 rad/s	-

a sampling period of $h = \frac{1}{20}$s. We estimated the characteristic time constant of this task to be $\tau_{\text{task}} = 1$s and chose h such that the number of time steps in a typical solution was small, but still large enough for the agent to receive feedback on a 5% faster solution – a solution that is one sampling period faster on the characteristic time scale of the task, and thus saves one time penalty. Because the task is episodic, we chose $\gamma = 1$. The measured control delay of the RL agent on the robot was $T_{\text{d}} = 1.4$ms – 3% of the sampling period. In Section 2.4, it is shown that when the control delay becomes too large, it can significantly violate the Markov property and therefore cause convergence problems. However, for this task, the control delay is relatively small and therefore neglected. The learning rate was $\alpha = 0.25/16$ (where 16 is the number of tilings and therefore the number of simultaneously active features in the function approximator). The exploration rate was $\epsilon = 0.05$. The eligibility time constant was chosen $\tau_{\text{elig}} = 0.127$s with corresponding trace discounting factor $\lambda = 0.674$.

Reward function

The agent receives a reward of 100 for completing the task, i.e., to touch the step with both the heel and the toe. Every step, a time penalty of -3.75 is given, hence time is punished with -75s^{-1}. In this way, solutions that take 1.33s or less result in a positive total reward per episode. Since $\hat{Q}(s,a)$ was initialized around 0, the agent will prefer unexplored actions over actions that solve the task in more than 1.33. This stimulates exploration.

Training program

The robot begins an episode with its leg in the straight down position and its foot parallel to and above the floor. An episode ends when the task is completed, i.e., when the robot's heel and toe are in contact with the step[4]. The robot's leg is then brought back into its initial position by a pre-programmed controller. To avoid that the agent spends too much time in uninteresting parts of the state space, an episode is discontinued after 60 seconds, but not through a terminal

---

[4] In one experiment, the contact sensor in the toe was broken and always returned 'on'. The robot discovered that it was possible to press the heel sensor by curving its leg *backward*, hitting the lower torso. After a coffee break, we found that the robot learned to kick its own behind.

absorbing state (see Section 2.1.1). Regarding the state of the robot after 60s as terminal absorbing would result in an unexpected and unpredictable event of defining $r_{k+1} = 0$ and $Q(s_{k+1}, a_{k+1}) = 0$. Instead, learning updates are discontinued (or 'paused indefinitely'), eligibility traces are cleared and the robot continues to learn from its initial configuration state.

Results

The result of learning the stairs step-up task both in simulation and on the real robot is presented in Figure 3.13, showing the average episode length (i.e., task completion time) against learning time. The learning curve from simulation shows the average over 48 runs; the learning curve from the real robot shows the average over 12 runs. Unfortunately, the learning runs on the real robot were ended prematurely after approximately 13 minutes or more due to inaccurate time bookkeeping in the software (while the runs had a fixed duration, the regular motor cooling pauses were accidentally not subtracted from this duration). We can observe that the robot learns to perform the task in approximately 15 minutes. The learning curves from simulation and from the real robot do not differ significantly, at least up to 13 minutes, which is an indication that the simulation environment is sufficiently accurate to predict learning results for this task. The learning runs on the prototype are plotted individually in Figure B.1 to illustrate the learning progress of individual runs.

Some learning runs on the robot had to be terminated due to hardware failure. The gearbox in the hip motor was damaged a few times. Furthermore, the foot sensor in the toe once got 'stuck' due to a friction problem in the mechanical construction that presses and releases the sensor. These runs were omitted from the results.

3.3.2 Hardware changes

From the robustness test that was reported in Section 3.1.4 and from the robot results described in Section 3.3.1, we discovered that the gearboxes of the joint motors formed the weakest link of the hardware. During these experiments and during preliminary tests with learning to walk (see Section 3.3.3-3.3.5), we regularly observed gears with broken teeth, despite the fact that they are fully made of metal. It is expected that sudden accelerations of the joint angle, e.g., due to foot impact or due to random changes in actuation voltage when exploring, are especially harmful for the gearbox. Therefore, to reduce impacts on the gearbox, we placed elastic couplings between the motor shaft and the robot brackets for the knee and hip joints; see Figure 3.14. By varying the material of the elastic element within the coupling, the stiffness of the coupling can be varied. Unfortunately, the elastic coupling introduces angular deviations between the motor position and the joint position, which are not measured; only the motor position is measured. The couplings used in the knees have a static torsional stiffness of

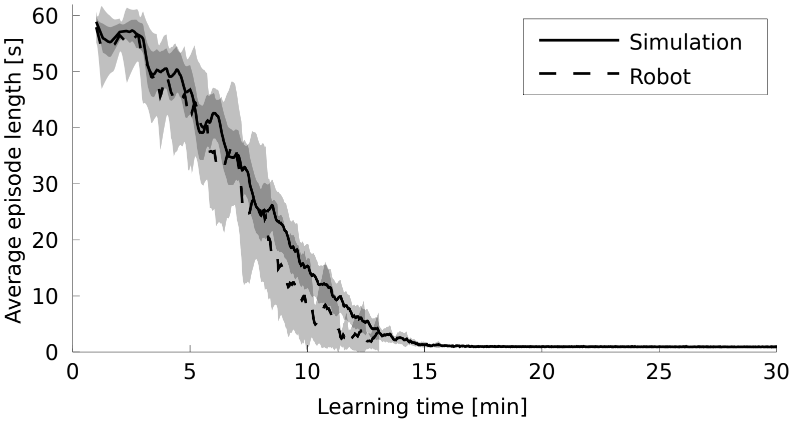


Figure 3.13: *Robot Leo learning the stairs step-up task, both in simulation and on the real robot. The average episode length (lower is better) is plotted against learning time. The learning curve from simulation shows the average over* 48 *runs; the learning curve from the real robot shows the average over* 12 *runs, which were, unfortunately, prematurely ended after approximately* 13 *minutes of learning.*

150Nm/rad and a dynamic torsional stiffness of 300Nm/rad (manufacturer specifications). In a static situation where the motor exerts its maximum stall torque of approximately 2.5Nm at 10.7V, this results in an angular deviation of approximately $1.7 \cdot 10^{-2}$rad (one degree) between the motor shaft and the robot bracket. Such deviations are in principle a violation of the Markov property. However, they are of the same order of magnitude as the backlash (approximately $1 \cdot 10^{-2}$) and the sensor accuracy (approximately $1 \cdot 10^{-2}$). While these couplings sufficiently reduced the mean time between failures for the gearboxes in the knees – they would not break for hours of experimentation time – the gearboxes of the hip joints would typically break after $5 - 10$min of learning to walk on the prototype. Therefore, we used couplings with a lower stiffness in the hip joint – the static torsional stiffness being 53Nm/rad and the dynamic torsional stiffness being 106Nm/rad. This increased the mean time between failures of the hip motors to about 30min. Unfortunately, this causes a violation of the Markov property with as of yet unknown effects. It was, however, the only adjustment we could make without doing a complete redesign of the robot, which would be required when switching to different actuators and/or transmission. The elastic couplings were used in the experiments with the prototype described below, but they were not modeled in simulation.

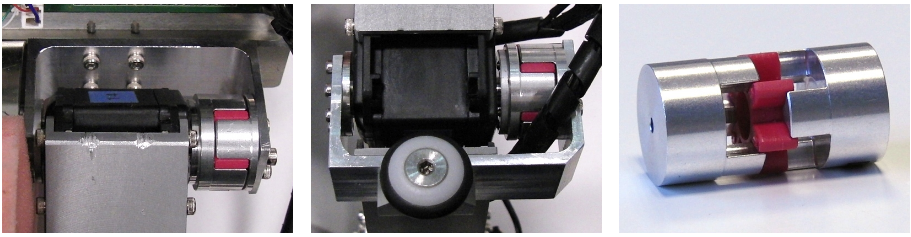

(a) Improved hip joint (b) Improved knee joint (c) Elastic coupling

Figure 3.14: *To reduce impacts on the gearboxes in Leo's hip and knee motors, elastic couplings were added between the motor shaft and the robot. Servo motors (black) are connected to their bracket via two rigid elements with an elastic element (red) in between. The coupling has a diameter of* 25*mm.*

3.3.3 Learning to walk

For the task of learning to walk, we used SARSA(λ) with the ϵ-greedy policy (2.9), combined with tile coding function approximation. Because SARSA is an on-policy algorithm, the estimated Q-values take exploration into account, which helps the robot choose actions that perform well on the real system while continuing to explore. The MDP design, learning parameters and other details are discussed below.

State-action space

We used a state space description that allows to exploit the robot's mirror-symmetry between its left and right side as illustrated in Section 2.3.6. This is implemented by labeling the leg that touches the floor as stance leg and the other leg as swing leg – regardless whether it is the left or right leg. When both legs touch the floor, the stance leg is the leg whose foot is in front of the other. This approach effectively reduces the memory required to store $\hat{Q}(s,a)$ by a factor of two.

In order to make this task feasible in terms of computational requirements and learning time, we restricted the number of actuators controlled by the learning agent to three: the hip motors of the stance leg and the swing leg, and the swing leg's knee motor. The remaining four actuators for the stance knee, shoulder and both ankle joints are controlled using simple position controllers (voltage control) of the form (3.25), where the desired angular positions are kept constant in time. In this way, the pre-programmed controllers pose a virtual constraint on these joints; the desired stance knee angle $\varphi_{\text{st.knee,ref}} = 0$ keeps the stance leg stretched, the desired shoulder joint angle $\varphi_{\text{shoulder,ref}} = 0$ keeps the arm parallel to the torso and the desired ankle angles $\varphi_{\text{st.ankle,ref}} = \varphi_{\text{sw.ankle,ref}} = 0.065\text{rad}$ keep the feet almost perpendicular to the lower leg, with the toes slightly tilted upward. The proportional gains were chosen as $P_{\text{ankle}} = 42$, $P_{\text{st.knee}} = 21$ and $P_{\text{shoulder}} = 21$. The control law, its gains and desired angles resulted from manual tuning in simulation and on the real robot with the goal to obtain simple and stable controllers that resulted in the desired virtual constraints. A Markovian state vector $\boldsymbol{s}$ was composed of the angle and angular rate of the torso, stance hip, swing hip, stance knee and swing knee – a total of 10 dimensions:

$$\boldsymbol{s} = \begin{pmatrix} \varphi_{\text{torso}} \\ \dot{\varphi}_{\text{torso}} \\ \varphi_{\text{st.hip}} \\ \dot{\varphi}_{\text{st.hip}} \\ \varphi_{\text{sw.hip}} \\ \dot{\varphi}_{\text{sw.hip}} \\ \varphi_{\text{st.knee}} \\ \dot{\varphi}_{\text{st.knee}} \\ \varphi_{\text{sw.knee}} \\ \dot{\varphi}_{\text{sw.knee}} \end{pmatrix} \tag{3.20}$$

The shoulder and ankles are omitted from the state space due to their virtual constraints[5]. The torso angle is absolute and thereby defines the pose of the robot with respect to the gravity vector. The robot is assumed to always make contact with the floor with at least one point; if this assumption would be violated, the state signal should include the robot's height and velocity vector with respect to the floor as well. The floor is assumed to be level, flat and homogeneous across the robot's walking circle. While floor height differences do exist in the real setup, as is shown in Figure 3.15, the robot's position within the walking circle is not included in the state space. Therefore, floor height differences are regarded as noise by the agent. The influence of the floor height differences was tested in a separate simulation experiment by adding a floor model based on the measured floor height data from Figure 3.15.

We chose the primitive action $\boldsymbol{a}$ to be a vector of desired constant voltages for all three motors for the next sampling period h:

$$\boldsymbol{a} = \begin{pmatrix} U_{\text{st.hip}} \\ U_{\text{sw.hip}} \\ U_{\text{sw.knee}} \end{pmatrix} \tag{3.21}$$

While Leo's actuators are servo motors, we chose not to use their internal position controllers, but to directly set desired voltages. Especially in the double stance phase of the robot, when the system is overactuated, position control of the joints could lead to undesirable behavior. To summarize, the state-action space consists of 10 state dimensions and 3 action dimensions.

---

[5] The angle and angular rate of the stance knee are included in the state space, because $\varphi_{\text{st.knee}}$ often significantly deviates from $\varphi_{\text{st.knee,desired}}$ during operation, e.g., when the robot makes a footstep with a bent swing leg, which then becomes the new stance leg.

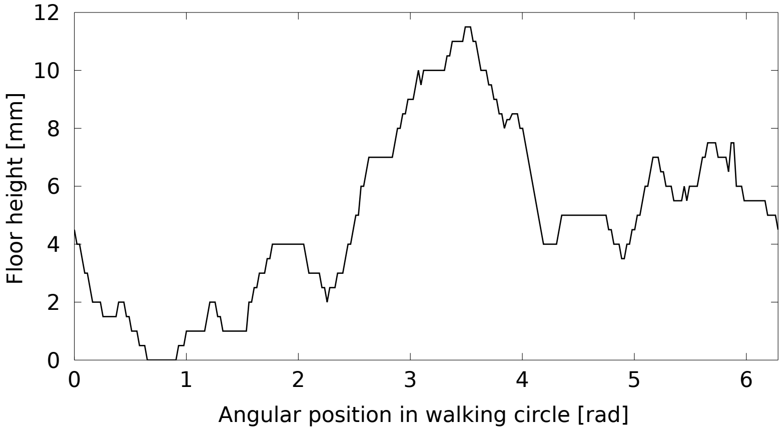


Figure 3.15: *The height of the floor along the walking circle of robot Leo. The circle has a perimeter of approximately* 11*m. The height is plotted relative to the lowest measured point. The estimated measurement error is* 1*mm.*

Function approximation

The tile coding function approximator used contained 16 tilings, displaced according to (2.22). It generalizes between states as well as between actions, i.e., a 13-dimensional function approximator is used for the state-action space as a whole. The tile widths used for all state and action variables, i.e., the generalization widths, can be found in Table 3.5. The resolution for the torso state information was doubled because of the torso's relatively large influence on the dynamics for small angular displacements, caused by its large mass and inertia and its inherent instability during walking. The feature parameters are trained in a gradient descent fashion according to (2.15) and (2.18). The memory required for storing the feature parameters was reduced through hashing, as described in Section 2.3.6. To stimulate exploration and to break ties at the beginning, we initialized $\hat{Q}(s, a)$ with random values between 0 and 0.01. Each time the experiment was repeated, the randomization was different, which caused the robot to learn a slightly different walking strategy in each experiment.

Learning parameters

We discretized the voltage range $[-10.7, 10.7]$ of each actuator controlled by the agent into 7 discrete voltages, giving a total of $7^3 = 343$ elements in A. We chose a sampling period of $h = \frac{1}{30}$s. These choices are the result of maximizing h by trial and error, while ensuring that the resulting learning controller could still let the robot walk stably; an increase in h results in an exponential decrease in the size of the solution space, which results in faster learning (see Section 2.1.2). The number

Table 3.5: *Tile widths (generalization widths) used in the tile coding function approximation of $Q(s, a)$ for robot Leo's task of learning to walk.*

	φ	$\dot{\varphi}$	U
Torso	0.14 rad	5 rad/s	-
Stance hip	0.28 rad	10 rad/s	6.7V
Swing hip	0.28 rad	10 rad/s	6.7V
Stance knee	0.28 rad	10 rad/s	-
Swing knee	0.28 rad	10 rad/s	6.7V

of elements in A should be limited, because when using (2.9) as action selection policy, the computational time and resulting control delay rises exponentially with the number of allowed discrete voltages per motor. The measured control delay of the RL agent on the robot was $T_\mathrm{d} = 10\mathrm{ms}$ on average – 30% of the sampling period. In Section 2.4, it is shown that when the control delay becomes too large, it can significantly violate the Markov property and therefore cause convergence problems. Therefore, the simulation experiments were done with and without delay to verify its influence. The learning rate was $\alpha = 0.20/16$ (where 16 is the number of active features in the function approximator). The exploration rate was $\epsilon = 0.05$. We estimated the characteristic time constant of this task to be $\tau_\mathrm{task} = 8.7\mathrm{s}$ as to represent a time horizon of several footsteps, which typically take between 0.5s and 1.0s. This resulted in a time discounting factor $\gamma = 0.996$ (see Section 2.1.2). The eligibility time constant was chosen $\tau_\mathrm{elig} = 0.22\mathrm{s}$ – roughly 25-40% of the footstep time, depending on the gait – with corresponding trace discounting factor $\lambda = 0.859$.

Reward function

We used a reward function that has the goal to let the robot walk forward, where time is punished to promote walking speed, and where energy usage is punished to promote efficiency. The reward function is a dense function (feedback every time step) and has the following components. To reward forward movement of the robot, the displacement of the swing foot is measured every time step and rewarded $300\mathrm{m}^{-1}$ for positive displacement and $-300\mathrm{m}^{-1}$ for negative displacement. When walking forward, this results in a total reward of $600\mathrm{m}^{-1}$ because alternatingly, both feet are moving forward. Every time step, a reward of -1 is given, so that time is punished with $-30\mathrm{s}^{-1}$. When the robot falls, it receives a reward of -125. Energy usage is punished every time step with $-3\mathrm{J}^{-1}$ proportional to the total positive electrical work W^+_total, which is the sum of W^+ of both hip motors and both knee motors:

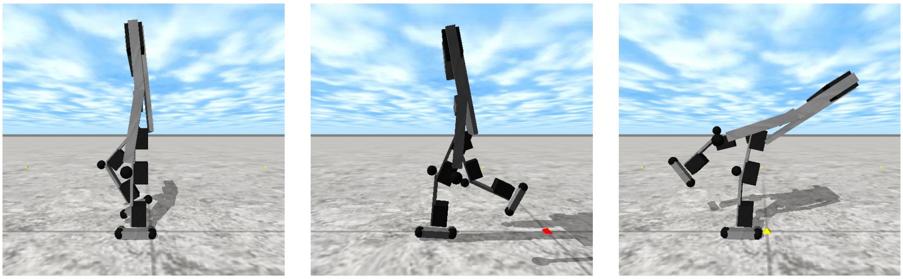
$$\begin{aligned}
W_k &= hU_kI_k \\
&= hU_k(U_k - K_\tau G\overline{\omega}_k)/R \\
&= hU_k(U_k - K_\tau G(\omega_k + \omega_{k+1})/2)/R && (3.22) \\
W_k^+ &= \text{MAX}(0, W_k) && (3.23) \\
W_{\text{total},k}^+ &= W_{\text{st.hip},k}^+ + W_{\text{sw.hip},k}^+ + W_{\text{st.knee},k}^+ + W_{\text{sw.knee},k}^+ && (3.24)
\end{aligned}$$

where U_k is the motor voltage, $K_\tau = 9.92 \cdot 10^{-3}$Nm the torque constant, $G = 193$ the gearbox ratio, $R = 8.6\Omega$ the winding resistance and $\overline{\omega}_k$ the averge joint angular rate during time step k, which is approximated by averaging ω_k and ω_{k+1} – the joint angular rates measured at time $t = kh$ and $t = (k+1)h$, respectively.

The values for the components of the reward function were chosen in such a way that for the resulting walking behavior, on average, the punishments for energy and for time were of the same order of magnitude as to learn a tradeoff between spending time and energy. The positive reward for forward movement was chosen larger than the sum of the negative rewards for time and energy, so that the net reward of walking was positive (i.e., larger than the initialization value of $\hat{Q}(s,a)$). The punishment for falling was chosen larger than the maximum obtainable reward of a specific local maximum: completely extending the swing leg, immediately followed by a fall. If the punishment for falling was chosen lower than the rewards collected during this act, the robot would typically spend large amounts of time exploring this local maximum while not exploring typical walking.

Training program

The robot was trained as follows. The robot starts in an initial configuration with zero angular rates and the following angles: $\varphi_{\text{torso,ic}} = -0.10$rad, $\varphi_{\text{st.hip,ic}} = 0.10$rad, $\varphi_{\text{sw.hip,ic}} = 0.82$rad, $\varphi_{\text{st.knee,ic}} = 0$rad, $\varphi_{\text{sw.knee,ic}} = -1.27$rad, $\varphi_{\text{st.ankle,ic}} = 0$rad, $\varphi_{\text{sw.ankle,ic}} = 0$rad. This configuration is relatively easy to accomplish on the real robot – the robot is balanced – and swinging the swing leg is facilitated by the slightly lifted swing foot. This is illustrated in Figure 3.16(a). An episode ends when the robot is doomed to fall, which is considered the case when the torso angle becomes too large, $|\varphi_{\text{torso}}| > 1.0$rad, or when the stance leg angle becomes too large, $|\varphi_{\text{torso}} + \varphi_{\text{st.hip}}| > 1.13$rad, see Figure 3.16(c). The latter is especially useful when the robot finds itself in a split, while its torso is still upright, thereby not able to walk any further. To ensure regular practice of starting to walk from the initial condition, the robot is put back in this configuration after 25s of successful walking. Such an episode is not terminated with a terminal absorbing state (see Section 2.1.1), but simply discontinued. Regarding the state of the robot after 25s as terminal absorbing would result in an unexpected and unpredictable event of defining $r_{k+1} = 0$ and $Q(s_{k+1}, a_{k+1}) = 0$. Instead, learning updates are discontinued (or 'paused indefinitely'), eligibility traces are cleared and the robot continues to learn from its initial configuration state.



(a) Initial configuration (b) Learning to walk (c) Doomed to fall

Figure 3.16: *Robot Leo in simulation, illustrating several situations during the process of learning to walk.*

Simulation results

The task of learning to walk was performed in simulation in three situations; on a flat floor without control delay, on a flat floor with control delay of $T_{\mathrm{d}} = 10$ms, and on a model of the real floor with control delay of $T_{\mathrm{d}} = 10$ms. The resulting learning curve is shown in Figure 3.17(a), where the average distance walked per episode, i.e., in max. 25s or until the robot falls, is plotted against learning time. The curves show averages over 48 runs. It can be observed that without control delay and on a flat floor, the robot learns to walk in 3 hours or less. Control delay has a negative influence on the learning time, increasing it to approximately 5 hours, but does not prevent the robot from reaching the same final performance. If a realistic model of the floor is used as well, based on the measured floor height data from the real robot setup, the learning time is further increased to approximately 6 hours – twice the learning time needed on a flat floor and without control delay.

From the variance of the performance and after inspecting individual runs, we observed that with the current learning parameters, the robot tends to find a locally optimal solution that does not further improve within the duration of the experiment. Given the size of the state space and the length of the experiment, this is not surprising, for we did not fulfill one of the requirements for convergence to the global optimum: visiting each state-action pair infinitely often. This can be improved upon by altering the exploration strategy so that the robot more easily explores a larger part of the state-action space, and by increasing the experiment length. It can also be noted that the specific reward values used here are the result of extensive experimentation, aiming for a short learning time and a natural walking gait. Different values for the rewards would typically lead to different walking gaits, e.g., a very 'lazy' gait with lots of foot scuff when the energy penalty was relatively high, and a very inefficient gait in which the swing leg was raised very high when the energy penalty was relatively low. Also, some combinations of rewards, specifically high penalties for time and energy, could increase the learning time by sometimes a factor of five or more if they would hinder the initial exploration process of making the first footsteps.

From Figure 3.17, it can be seen that the number of times the robot falls before it learned to walk is estimated to be at least 5000. Unfortunately, our current prototype will not withstand that many falls. We decided to reduce the floor height differences by creating a height map of the floor and leveling it with layers of foam and a laminate top layer, as illustrated in Appendix A. This resulted in a maximum floor height difference of approximately 3mm – an acceptable value compared to the robot's leg length. Below, we explore a method that avoid the initial period of learning from scratch on the prototype, while still being able to test the viability of the proposed RL approach, in order to further reduce the number of falls the prototype has to withstand.

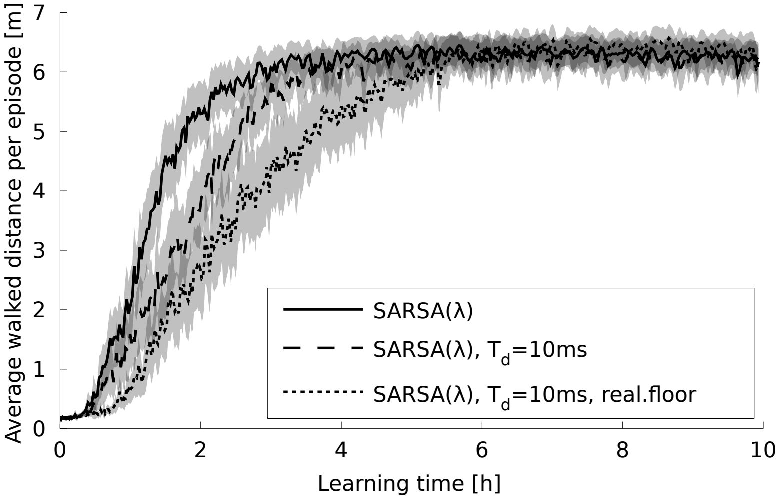
3.3.4 Pre-programmed limit cycle walking

To verify that Leo is physically capable of walking, a manually programmed limit cycle walking controller (Hobbelen, 2008) was used. The most prominent characteristic of the limit cycle walking paradigm is that the robot is allowed to be unstable at every instant in time; however, over the course of several footsteps, its behavior is stable as a whole and converges to a limit cycle in the form of periodic walking. There is no pre-described trajectory in time for all joints that needs to be accurately tracked by the controller to guarantee stability. Individual footsteps are allowed to differ significantly from each other without jeopardizing stability, which is obtained at the time scale of multiple footsteps instead of a single footstep. When the robot's walking gait is too fast, this will lead to larger footsteps, which dissipate more energy at heelstrike. This automatically reduces the walking speed and stabilizes the gait. Because joint angles are allowed to be less strictly controlled with this paradigm, a limit cycle controller usually results in less energy usage compared to control approaches that aim for full local controllability, such as 'Zero Moment Point' based control (Vukobratovic, Frank, and Juricic, 1970; Vukobratovic and Borovac, 2004). However, the choice of the control paradigm is subordinate in this experiment; we merely want to verify the robot's ability to walk.

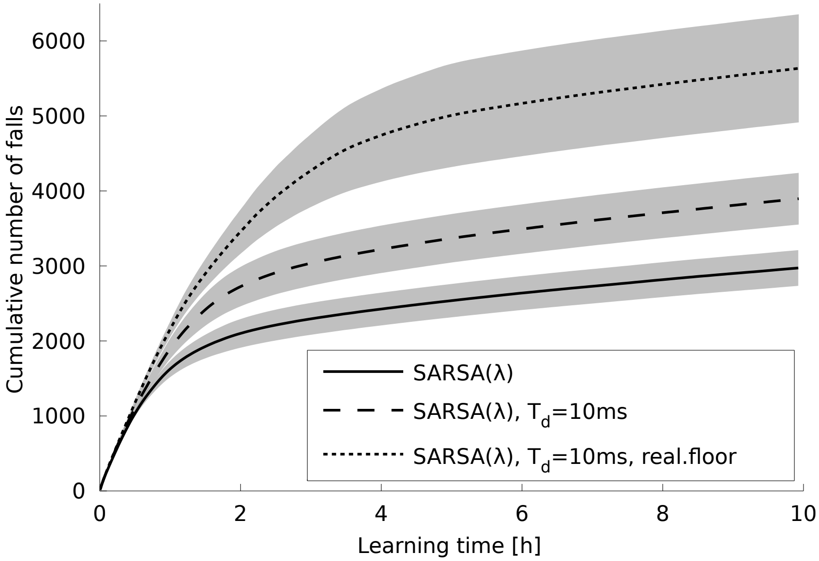
Controller parameters

The limit cycle walking controller is composed of several simple position controllers of the form

$$U = P(\varphi_{\mathrm{ref}} - \varphi) \tag{3.25}$$



(a) Average distance walked per episode against learning time



(b) Cumulative number of falls against learning time

Figure 3.17: *Robot Leo learning to walk in simulation. The learning performance is shown for three situations: on a flat floor without control delay, on a flat floor with control delay of* $T_\mathrm{d} = 10ms$*, and on a model of the real floor with control delay of* $T_d = 10ms$*. The learning curves show the average over* 48 *runs. Both the control delay and the realistic floor have a negative influence on the learning speed.*

where the motor voltage U is proportional with gain P to the error between a measured angle φ and a desired (relative) constant joint angle φ_{ref}. The individual control rules and their purpose for all motors are as follows. The stance hip motor controls the torso angle towards a fixed value, $\varphi_{\text{torso,ref}} = -0.09\text{rad}$. Both hip motors control the inter-hip angle $\varphi_{\text{interhip}} = \varphi_{\text{sw.hip}} - \varphi_{\text{st.hip}}$ towards a fixed value, $\varphi_{\text{interhip,ref}} = 0.68\text{rad}$. The control laws for the hip motors are:

$$\begin{aligned} U_{\text{st.hip}} &= P_1(\varphi_{\text{interhip,ref}} - \varphi_{\text{interhip}}) + P_2(\varphi_{\text{torso,ref}} - \varphi_{\text{torso}}) \\ U_{\text{sw.hip}} &= P_3(\varphi_{\text{interhip,ref}} - \varphi_{\text{interhip}}) \end{aligned} \tag{3.26}$$

with $P_1 = -3.4$, $P_2 = -59$ and $P_3 = 17$. The stance knee motor keeps the stance leg stretched, $\varphi_{\text{st.knee,ref}} = 0\text{rad}$. In the early swing phase, defined as the first 0.184s after heel strike, the swing knee motor bends the knee as quickly as possible using the maximum allowed voltage. After the early swing phase, the swing knee motor stretches the swing leg again towards $\varphi_{\text{sw.knee,ref}} = 0\text{rad}$. The control laws for the knee motors are:

$$\begin{aligned} U_{\text{st.knee}} &= P_4(\varphi_{\text{st.knee,ref}} - \varphi_{\text{st.knee}}) \\ U_{\text{sw.knee}} &= \begin{cases} -10.7 & \text{, early swing} \\ P_5(\varphi_{\text{sw.knee,ref}} - \varphi_{\text{sw.knee}}) & \text{, late swing} \end{cases} \end{aligned} \tag{3.27}$$

with $P_4 = 21$ and $P_5 = 65$. The feet are kept almost perpendicular to the lower legs, with the toes slightly tilted upward, $\varphi_{\text{st.ankle,ref}} = \varphi_{\text{sw.ankle,ref}} = 0.065\text{rad}$. The arm is kept parallel to the torso, $\varphi_{\text{shoulder,ref}} = 0\text{rad}$. The control laws for the ankles and shoulder are:

$$\begin{aligned} U_{\text{st.ankle}} &= P_6(\varphi_{\text{st.ankle,ref}} - \varphi_{\text{st.ankle}}) \\ U_{\text{sw.ankle}} &= P_7(\varphi_{\text{sw.ankle,ref}} - \varphi_{\text{sw.ankle}}) \\ U_{\text{shoulder}} &= P_8(\varphi_{\text{shoulder,ref}} - \varphi_{\text{shoulder}}) \end{aligned} \tag{3.28}$$

with $P_6 = P_7 = 42$ and $P_8 = 21$.

Results

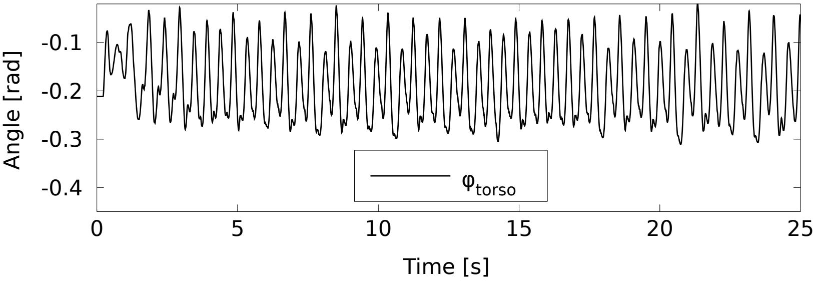
The typical behavior of the prototype while being controlled by the limit cycle walking controller is illustrated in Figure 3.18, in which the angles of the torso, the hips and the knees are plotted against time. The prototype walked on the leveled floor as described in Appendix A. One can observe that the prototype walks stably. A slight asymmetry is visible between left and right footsteps; the left and right hip angles have different minimum values, and the torso angle has a different maximum value for left and right footsteps. This can be caused by the fact that the robot is connected to the boom construction with one side, and due to differences in friction between the joints. We can also observe that during the 25s shown, in which the robot walked along approximately 75% of its walking circle, its behavior varies only slightly, indicating that remaining floor height differences do not have a big impact on the prototype's behavior. This contrasts with the prototype's behavior on the original, unleveled floor, on which it would show significant variations in walking speed, frequently leading to a standstill or fall at specific locations.

3.3.5 Initializing the learning controller with a pre-programmed solution

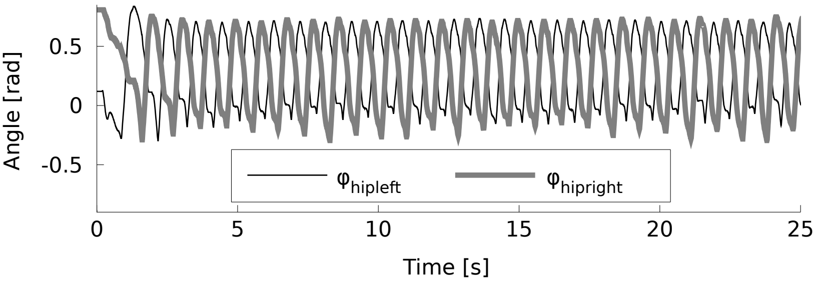
An apparent and often used approach to initialize the learning controller on a real robot is to estimate the action-value function in simulation and transfer it to the real robot, where it continues to be updated using real experience. This approach avoids the tedious initial period of the learning process, where movements are mostly random and frequently lead to falling, which is potentially harmful for the prototype. However, sometimes, an accurate simulation model of a robot is not available or hard to obtain. From our experience with some prototypes, e.g., robot Flame (Hobbelen, De Boer, and Wisse, 2008), we learned that creating an accurate and reliable simulation model can be more difficult than programming its controller. In this experiment, we present an alternative approach with the same goal, which does not need a simulation environment. Instead, it is assumed that a pre-programmed policy π_{pp} is available, which may be suboptimal.

The learning process is as follows. During a short initial period – the observation period – the prototype is controlled by a pre-programmed policy, while the learning agent estimates the action-value function $\hat{Q}(s,a)$ under that policy. After the observation period, actuation is continued according to the ϵ-greedy policy based on $\hat{Q}(s,a)$, which is further updated using SARSA(λ). It is expected that even after relatively short observation periods, the policy derived from $\hat{Q}(s,a)$ is good enough to avoid the frequent falling that occurs when learning from scratch. Because the initialization of the action-value function can be arbitrary without changing the global optimum that it converges to, there are no additional requirements for the pre-programmed policy. If π_{pp} is deterministic, controlling the robot purely according to π_{pp} during the observation period would expose the system to only a very small part of the state-action space; the system would show nearly (or in simulation, exactly) the same state evolution each time it starts a trial from the same initial condition. Since this would lead to a relatively poor initialization of $\hat{Q}(s,a)$, we use a policy similar to the ϵ-greedy policy during the observation phase in which actions according to π_{pp} are alternated with random actions: $\pi_{\epsilon-\text{pp}}$. Learning updates are applied to $\hat{Q}(s,a)$ to estimate $Q^{\pi_{\epsilon-\text{pp}}}(s,a)$, i.e., the action-value function under $\pi_{\epsilon-\text{pp}}$. This is done by using the original SARSA(λ) update rule (see (2.11) and (2.12)) where a_k and a_{k+1} are computed according to $\pi_{\epsilon-\text{pp}}$.

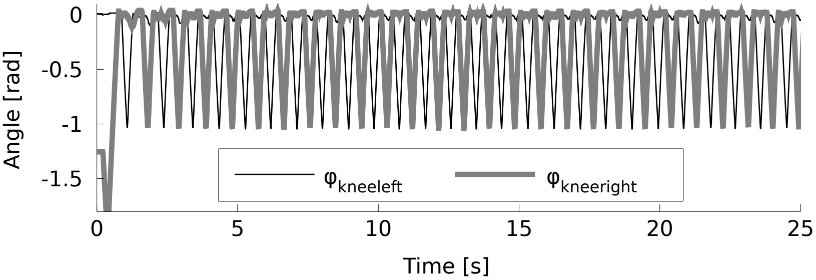
We tested our approach in simulation first, followed by repeating the experiment on the prototype. For the pre-programmed policy π_{pp}, we chose the policy presented in Section 3.3.4, which proved to result in good walking behavior on the prototype.



(a) Torso angle.



(b) Hip angles.



(c) Knee angles.

Figure 3.18: *Evolution of the joint angles of prototype Leo while being controlled by a pre-programmed limit cycle walking controller. One can observe that the prototype walks stably, with a slight asymmetry between left and right.*

Simulation results

For this simulation experiment, we chose the observation period to be 3 minutes during with the robot was controlled by $\pi_{\epsilon-\text{pp}}$ with $\epsilon = 0.05$. We ignored control delay and assumed the floor is flat and level. The results are presented in Figure 3.19, showing the average over 48 runs. It can be seen that for a short period after the observation period, the agent performs worse than π_{pp}. This is probably the result of the limited number of updates that were performed while following π_{pp}, which were not enough to let $\hat{Q}(s,a)$ converge. When the agent switches to executing the ϵ-greedy policy based on $\hat{Q}(s,a)$, this leads to a temporary drop in performance. Furthermore, when following π_{pp}, actions are not discretized as is the case for the ϵ-greedy policy. This means that Q-values for the discretized actions may be inaccurate because they are rarely selected. After approximately 1 hour of learning, the initial drop in performance is turned into a significant increase in performance over the pre-programmed policy π_{pp}. After approximately 5 hours of learning, however, this increase becomes insignificant, although the agent never performs significantly worse than π_{pp}. The solution found is also significantly better than the average performance of the policies learned from scratch, which is added to Figure 3.19 for reference. It can be noted that when learning from scratch, individual runs would occasionally achieve a performance comparable to the pre-programmed one, albeit with quite a different gait. However, on average, learning from scratch performed worse. This illustrates that guiding the robot to interesting parts of the state space can significantly increase the average performance achieved in a limited time span. Nonetheless, the fact that learning from scratch *can* result in a solution that is comparable to the pre-programmed solution, which embodies extensive expert knowledge on bipedal walking, illustrates the potential of the applied methods.

Prototype results

From the simulation results, it was observed that an initial drop in performance occurred. With the aim to make this dip smaller, we increased the observation period to 5 minutes and kept the other parameters the same as in simulation. Due to the high resource costs in terms of materials and repair time, the experiment was performed only once for a duration of 4.5h. The result is shown in Figure 3.20. One can observe that, initially, the robot shows a dip in its performance (as was observed in simulation), after which it learns to walk approximately as well as the pre-programmed controller, showing occasional periods in which it performs better than the pre-programmed controller. During the experiment, the prototype broke on average every 30min, usually due to a failing gearbox in one of the hip joints.

Unfortunately, due to the lack of additional learning runs and the stochastic nature of the experiment, we cannot draw conclusions on the significance of the demonstrated performance, nor on the convergence of the learning process on

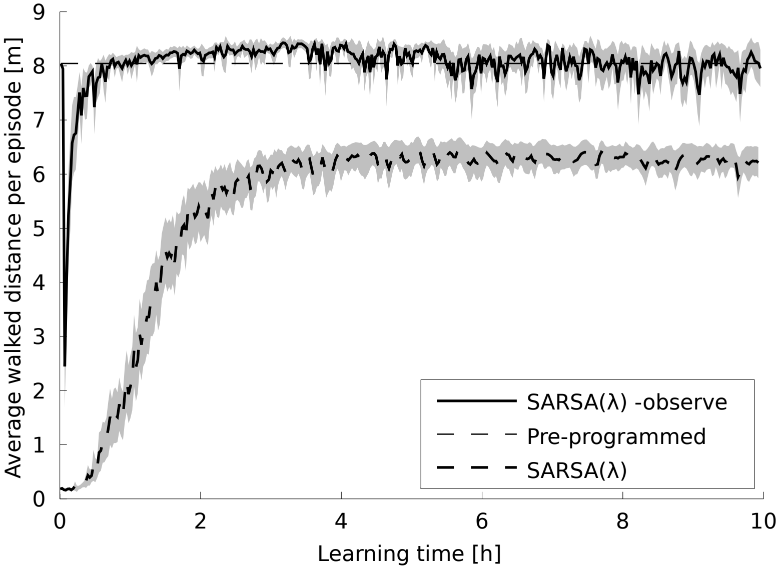


Figure 3.19: *Robot Leo learning to walk in simulation. During an initial period of* 3 *minutes,* $\hat{Q}(s,a)$ *is estimated for a pre-programmed controller, after which actuation and learning is continued by the learning agent. The average walked distance per episode is plotted against simulation time and compared with the SARSA(λ) approach from Section 3.3.3. The learning curves show the average over* 48 *runs.*

the prototype. However, we can conclude that this initial result shows that the prototype and the presented RL techniques form a promising start for further experimentation; for the duration of the experiment, in which it performed more than $4 \cdot 10^5$ learning updates, learning did not divergence, despite the fairly large values for the exploration rate ($\epsilon = 0.05$) and the learning rate ($\alpha = 0.2/16$ with 16 being the number of tilings).

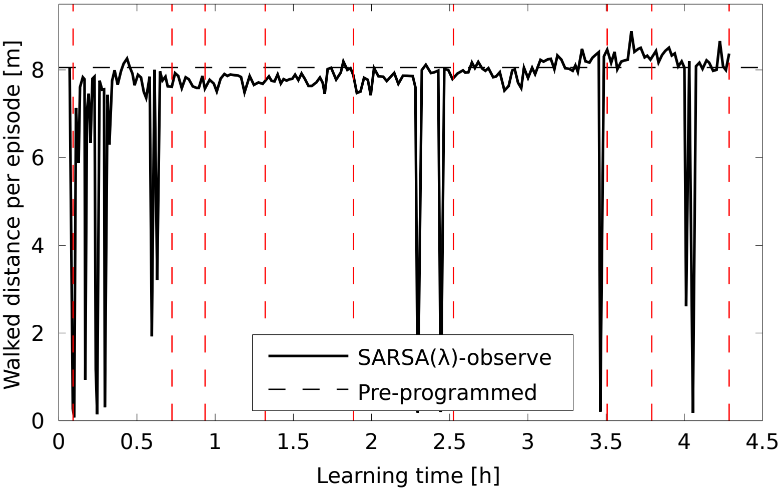


Figure 3.20: *Robot Leo learning to walk using the real prototype. During an initial period of* 6 *minutes,* $\hat{Q}(s,a)$ *is estimated for a pre-programmed controller, after which actuation and learning is continued by the learning agent. The walked distance per episode is plotted against learning time. The initial performance of the pre-programmed controller is shown by the horizontal dashed line. The vertical dash-dotted lines indicate moments of hardware repair; usually, a hip motor was replaced due to a broken gearbox.*

3.3.6 Conclusions

In this section, we presented learning results obtained with robot Leo (described in Section 3.1 and 3.2) in simulation as well as on the prototype. We started with a relatively simple learning task that involved only one leg (the other body parts were statically mounted): the stairs step-up task. In this task, which had 6 state dimensions and 2 action dimensions, the robot had to learn to place its foot on a plateau. We showed that the robot is able to learn this task from scratch in approximately 15 minutes. The results obtained in simulation and in hardware did not differ significantly. This showed that the prototype's hardware and software are suitable for RL experiments of short duration. It also revealed a weakness in the prototype's construction: the gearboxes of the actuators quickly wore down due to the large (alternating) forces they were exposed to. Because the gearboxes wore down even faster in preliminary experiments of learning to walk on the real robot, we decided to add elastic couplings between the actuators and the robot brackets to reduce the peak forces on the gears. This significantly increased the mean time between failures.

Next, we defined the MDP of learning to walk – a task with 10 state dimensions and 3 action dimensions for our prototype – and presented simulation results. The robot was able to learn to walk from scratch in 3 hours or less when control delay and floor height differences were ignored. When control delay was included in the simulation, the learning time increased to approximately 5 hours. When the floor height differences measured in the real robot setup were included in the simulation, the learning time further increased to approximately 6 hours. Additionally, the number of times the robot fell before learning to walk approximately doubled due to the floor height differences. While our prototype was kept small to reduce the impact of falling, ironically, this decision heavily increased the number of falls since existing floor height differences become more significant in relation to the small legs. Therefore, the floor was leveled using a laminate floor, which reduced the floor height differences to acceptable values. We verified that the prototype with elastic couplings and leveled floor was able to walk by controlling it with a pre-programmed controller, which resulted in a stable walking gait.

The simulation results showed that the prototype would have to withstand thousands of falls before learning to walk from scratch. Unfortunately, in its current form, the prototype was not deemed capable of withstanding that many falls. To be able to study the presented RL techniques of learning to walk on the current prototype, we initialized the learning controller with a pre-programmed solution, thereby avoiding the initial learning period in which the robot falls frequently. During a short period of time, the robot was controlled by a pre-programmed controller, for which it estimated the action-value function. After this initial observation period, the robot continued the learning process. The robot learned to walk at least as well as with the pre-programmed controller, while the number of falls was greatly reduced compared to learning from scratch. After obtaining these positive results in simulation, the experiment was repeated on the prototype. This showed that the prototype was able to learn a suboptimal policy in a matter of minutes by observing the pre-programmed controller, after which it was able to improve its policy with the same techniques used when learning to walk from scratch in simulation. Unfortunately, significant results on the average performance and convergence could not be obtained due to the large resource costs of the experimental setup in its current form. Therefore, this result can best be regarded as a proof of concept.

3.4 Conclusions

In this chapter, we presented the hardware and software design of robot Leo and presented learning results in simulation and on the prototype. The robot was able to perform learning experiments in real-time, on its embedded hardware, without human support or intervention – with the exception of hardware failures, which, unfortunately, limited the maximum consecutive experimentation time with the prototype to approximately 30 minutes. Numerous design improvements have greatly increased its fitness with respect to the hardware requirements derived from the RL framework used in this thesis. To improve the state signal, we increased the stiffness of the boom construction to reduce vibrations, replaced potentiometers with magnetic encoders, applied low-pass filtering to the signals from the joint position encoders and leveled the floor to reduce the floor height differences. To withstand the peak forces from foot impact, random actuation signals due to exploration and frequent falls, we improved the motherboard mounting, added damping foam in key locations, increased the thickness and material strength of the brackets and added elastic couplings between the actuators and the robot brackets to protect the gearboxes. To make actuation more predictable, we compensated the actuation signals for the temperature of each motor. It was striking to see how an initial version of the prototype would last 8 hours before breaking down when walking with a 'traditional', pre-programmed controller, while it would last only 5 minutes when applying learning control to learn to walk. Eventually, after the aforementioned improvements, the prototype would typically have a mean time between failures of 30 minutes when learning to walk, with the weakest link being the gearboxes of the motors in the hip joints.

The motion control software developed for the prototype contained a clear abstraction layer, which made it possible to write controller code that could be compiled without modification for both the simulation environment and the real-time environment on the robot, including facilities for logging to screen and file. Using a publish/subscribe architecture, state information was distributed to the controller and additional modules such as logging and visualization services. By means of a separate actuation service, which allowed changing the actuator signals at any time, control delay could be kept to a minimum. To achieve deterministic timing on the prototype, we used Linux with real-time Xenomai extension. The simulation environment, based on the fast and stable Open Dynamics Engine, was extended with convenient XML configurability and allowed controllers – both conventional and learning – to be tested safely before being deployed onto the robot.

The first RL experiment on the prototype consisted of a relatively simple learning task that involved only one leg (the other body parts were statically mounted) – the stairs step-up task – and showed that the robot was able to learn from scratch to place its foot on a plateau in approximately 15 minutes. The results obtained in simulation and in hardware did not differ significantly. This showed that the prototype's hardware and software were suitable for RL experiments of short duration. Subsequently, we defined the MDP of learning to walk – a task with 10 state dimensions and 3 action dimensions for our prototype – and presented simulation results. In simulation, the robot learned to walk from scratch in 3 hours or less. We showed that both control delay and floor height differences increased the learning time (up to a factor of two), as well as the number of falls occurring during the learning process. To reduce the latter, we adjusted the robot setup by leveling the floor. The simulation results also showed that the prototype would have to withstand thousands of falls before learning to walk from scratch. Since our prototype was not robust enough for that, we applied a method to speed up the initial learning period in which the robot falls frequently, in order to still be able to study the presented RL techniques of learning to walk on the current prototype. During an initial period of a few minutes, the prototype was controlled by a pre-programmed controller. For this demonstrated solution, the action-value function was estimated on-line, which then served as an initialization for the remainder of the learning process. After successfully testing this method in simulation, we executed the method on the prototype for 4.5 hours, in which it learned to perform at least as well as the demonstrated pre-programmed solution. To the best of our knowledge, this is the first demonstration of Temporal Difference learning in real-time, on embedded robot hardware involving a non-trivial task and a high-dimensional state-action space.

Chapter 4

The effects of large disturbances on learning to walk

An important difference between the real world and simulation is the presence of disturbances. In Reinforcement Learning, disturbances are typically regarded as stochastic variations of state transitions and actions. However, large and infrequent disturbances do not fit well in this framework; essentially, they are outliers and not part of the underlying (stochastic) Markov Decision Process. In this chapter, we investigate the effects of large and infrequent disturbances on the on-line learning process of a simple walking robot in simulation. This section is based on (Schuitema et al., 2010c).

4.1 Introduction

In an attempt to close the gap between RL in simulation and on real robots, this chapter investigates the effects of large disturbances on the RL task of learning to walk. For our prototype Leo – see Chapter 3 – we have found the main sources of disturbance to be sensor noise, sampling period irregularities, the temperature dependence of the actuators and unpredictable interactions with the environment such as floor height irregularities. Disturbances can be (virtually) instantaneous, such as sensor noise and sampling time irregularities, or lengthier such as wind, changing floor slope, or (temporary) sensor drift.

Within RL, the learning agent and its environment are usually modeled as a Markov Decision Process or MDP; see Section 2.1. It is common practice for RL algorithms to allow stochastic state transitions (and rewards) in the MDP (Bertsekas, 1987; Jaakkola, Jordan, and Singh, 1994; Tsitsiklis, 1994). Usually, averaging over sufficient experiences will allow the algorithm to find an optimal solution to such a stochastic MDP. However, not all disturbances can be considered to be part of the stochastic nature of the problem. Some disturbances, especially large and infrequent ones, should be considered as *outliers*. While some algorithms – particularly off-line ones – are robust against outliers, to our knowledge no in-depth study has been made of the effect of realistic outliers on learning a control policy for real-time dynamic systems. In this chapter, we focus on on-line Temporal Difference (TD) learning as described in Section 2.2.

Dealing with outliers involves several steps. The first step is to *detect* the outlier. Once it is detected, the system can *reject* the outlier, excluding it from the learning process. A possible further step is *correction*, i.e., to try and counter the disturbance while it is active, which is only possible when the disturbance has a significant duration. Appropriate correction requires *classification* of the outlier to predict its evolution. The final step is *recovery*, during which the system abandons the undesirable state (region) that resulted from the disturbance. Deliberately *including* disturbances in the learning process might result in a solution that is more robust against disturbances. However, this is only possible if they occur often enough, i.e., if they are part of the stochastic nature of the system.

Several techniques have been studied to make RL more robust against disturbances (Singh et al., 1994; Kretchmar et al., 2001; Doya, 2001; Anderson et al., 2007). While this work offers important and useful techniques, the difference in effects of various types of disturbances remains unknown. We explore the influence of several types of outliers on the learning process of a simple simulation[1] model of a walking robot – the simplest walker model (see Section 4.3.1) – in order to assess the need for further steps such as detection and classification. We use this simple model instead of the fairly complex simulation model of robot Leo because its low computational complexity allows us to more easily produce statistically significant results. To illustrate the benefit of being able to detect outliers, we test the effect of excluding the outliers from the learning process – a relatively simple operation in TD learning. We do not look into the ability of the learning agent to withstand stochastic disturbances, e.g. normally distributed disturbances of moderate size; we only consider large, infrequent disturbances that can best be regarded as outliers. We also do not look into slow and permanent changes in the system or its environment, because this requires different properties of the learning algorithm: the ability to adapt to new situations.

4.2 Large disturbances during RL

For a real-time dynamic system such as a humanoid robot, the main sources of outliers are sudden changes in the dynamics, sensor and actuator noise and sampling time irregularities. We now discuss the effect of each of these categories of outliers on the learning process of RL, as well as disturbance rejection and detection.

---

[1] While in simulation one can choose when and how to apply a disturbance, on a real robot, this happens mostly involuntarily. Therefore, we use simulations throughout this section to show the effects of several types of large disturbances.

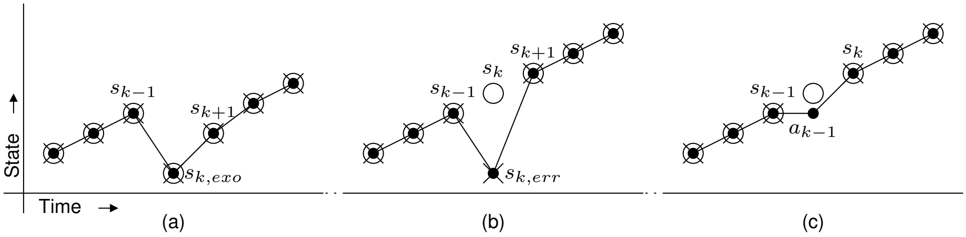


Figure 4.1: *Effect of disturbances.* ○ *is the actual state,* × *is the measured state, and* • *is the state on which the effective action at that timepoint is based. (a) Instantaneous push. (b) Erroneous sensor reading. (c) Missing sample (incidental sampling period doubling).*

4.2.1 Outliers due to unexpected interactions with the environment

An external disturbance such as an instantaneous push or a step-up or step-down in the floor can cause an outlier in the state transitions and bring the learning agent into an exotic state $s_{k,exo}$; see Figure 4.1a for a one-dimensional example. In the learning updates that follow, the agent will erroneously relate $Q(s_{k-1}, a_{k-1})$ to $Q(s_{k,exo}, a_k)$ and a reward related to the transition to $s_{k,exo}$. Especially when $s_{k,exo}$ has an associated negative reward or will eventually but surely lead to negative rewards, states on good solution paths might be linked to the negative results of the disturbance. The agent cannot relate and thus not reward its behavior before the disturbance to the reward that it actually deserves.

When the agent almost never suffers from such disturbances, it probably does not have enough experience to recover from the resulting exotic states. However, when it is disturbed frequently, it is able to practice in these regions of the state space as well, so that it automatically learns to recover from these outliers. Note that this is likely to lead to a longer learning time, since the robot visits a larger part of the state space.

The effect of actuator noise is similar. However, actuator noise is likely to bring the system in a state that it also could have visited under normal conditions, e.g. due to exploration. Therefore, this type of outlier is less interesting and we do not further look into it.

In summary, the learning problem becomes larger because a larger part of the state space is visited. Furthermore, with every outlier, parts of the state space are connected that are in principle not related, which can negatively influence learned behavior in frequently visited parts of the state space.

4.2.2 Outliers due to sensor noise

In case of an outlier in the sensor reading, the agent will perceive a state $s_{k,err}$ and reward $r_{k,err}$ that are different from the actual s_k and r_k. Here, r_k (and thus $r_{k,err}$) is computed by the controller from s_{k-1}, a_{k-1} and s_k. Because of this, the agent will experience two erroneously perceived state transitions; $(s_{k-1}, a_{k-1}) \rightarrow s_{k,err}$ and $(s_{k,err}, a_k) \rightarrow s_{k+1}$, see Figure 4.1b for a one-dimensional example. When the agent is estimating the action-value function of the MDP, the result is that $Q(s_{k-1}, a_{k-1})$ will be related to $Q(s_k^{err}, a_k)$, the latter most probably having a Q-value unrelated to the problem with $s_{k,err}$ being an outlier. Next, $Q(s_{kerr}, a_k)$ is related to $Q(s_{k+1}, a_{k+1})$. When using temporal difference (TD) learning with eligibility traces, however, it is likely that the subsequent TD updates (see (2.11)) largely cancel each other out. When using SARSA, for example, the two subsequent errors in the TD updates are:

$$\begin{aligned} \Delta\delta_{\mathrm{TD,k-1}} &= \delta_{\mathrm{TD,k-1}}^{err} - \delta_{\mathrm{TD,k-1}}^{correct} &&= \gamma(Q(s_k, a_k) - Q(s_k^{err}, a_k)) + r_k^{err} - r_k \\ \Delta\delta_{\mathrm{TD,k}} &= \delta_{\mathrm{TD,k}}^{err} - \delta_{\mathrm{TD,k}}^{correct} &&= Q(s_k^{err}, a_k) - Q(s_k, a_k) \end{aligned} \tag{4.1}$$

When using eligibility traces, *both* incorrect TD updates are applied to all $Q(s, a)$ in the trace, with maximum net error Δ_{max} for $Q(s_{k-1}, a_{k-1})$ (all other (s, a) pairs have smaller values of $e(s, a)$):

$$\begin{aligned} \Delta_{max} &= \alpha(\Delta\delta_{\mathrm{TD,k-1}} + \gamma\lambda\Delta\delta_{\mathrm{TD,k}}) \\ &= \alpha(r_k^{err} - r_k + \gamma(1-\lambda)(Q(s_k, a_k) - Q(s_k^{err}, a_k)) \end{aligned} \tag{4.2}$$

In the special case of using a simple reward function in which r_k is constant in large parts of the state-action space (this is true, e.g., when r_k equals a constant time penalty except for goal states and terminal states), r_k^{err} and r_{k+1} are often equal. For such simple reward functions, the remaining error approaches 0 when λ approaches 1, i.e., for lengthy eligibility traces.

However, the agent will base a_k on $Q(s_{k,err}, *)$, which results in a suboptimal action. This action can remove the agent from its optimal path - but not worse than when an exploratory action was chosen. Unlike with a dynamics disturbance, the agent has a fair chance to successfully continue its episode and receive the reward it deserves based on its behavior before the disturbance.

In summary, the most prominent effect of a sensor outlier are two learning updates with an erroneous TD-error (with eligibility traces, the effect is usually very small) and one extra suboptimal action.

4.2.3 Outliers due to sampling time irregularities

When the sampling time is suddenly disrupted, e.g., due to calculations taking longer than normal, the dynamics of the system evolve longer (or shorter) than normally. Depending on the type of action that the agent executes, the action itself will be shortened or extended, which results in a different resulting state. This is true for actions like motor torque or voltage when they are maintained until other actions overwrite them. However, unless the disruption lasts a large multiple of the sampling time, the effect is expected to be limited. See Figure 4.1c for a 1-dimensional example.

In summary, an outlier in sampling time is expected to have a limited effect, unless it lasts a large multiple of the sampling time.

4.2.4 Disturbance rejection and detection

For on-line TD-learning, simply skipping the learning update for state transitions that include outliers is enough to exclude the outlier from the learning process. When eligibility traces are used, the traces can simply be cleared once an outlier is detected. This skips the faulty learning update. Note that clearing eligibility traces can slow down the learning process.

In this work, we do not focus on the subject of outlier detection. In our simulations, we simply signal the learning algorithm that an outlier was detected at the moment we apply the disturbance. For on-line disturbance detection on robotics systems, a state transition model of the robot and its environment is needed, preferably learned on-line. Model learning techniques that might be appropriate for such systems are Locally Weighted Learning (LWL) (Atkeson, Moore, and Schaal, 1997), a local linear regression technique, and SmartSifter (Yamanishi et al., 2000). Once a model is available, every state transition can be compared to the expected state transition based on the model. If a measured state differs significantly from its prediction, the measurement can be labeled as an outlier. Prediction intervals can serve as a significance measure and are easily calculated for LWL.

4.3 Experimental setup

In order to evaluate the effect of several types of outliers on the learning process and the efficacy of skipping the learning update, we have performed simulations of a simple two-dimensional system - the simplest walker - that learns to walk using SARSA(λ). To simulate a disturbance, the walking system was severely perturbed for a single time step. Because we do not focus on the outlier detection aspect, we simply signal the learning algorithm of the presence of the outlier when the disturbance was applied. To reject the outlier, the learning update for this time step is not performed and eligibility traces are cleared. We tested three types of disturbances:

1. An ‘instantaneous’ push, which is a perturbation of the real state due to an unexpected interaction with the environment.
2. An erroneous sensor reading (spike noise).
3. A sampling time irregularity, resulting in a sample that takes longer to acquire.

4.3.1 The simplest walker

The simulated system is a compass walker (Garcia et al., 1998) consisting of two rigid legs of unit length connected by a frictionless hinge at the hip (Figure 4.2). A mass of unit size is located in the hip of the walker. The legs are massless, while the feet contain an infinitesimally small mass. This results in pendulum-like behavior of the swing leg. We allow the swing foot to be briefly below floor level during its swing, which is inevitable for a walker without knees; the second time the swing foot is at floor level height, the walker makes a step and the swing leg becomes the new stance leg. The system is described by the following equations of motion:

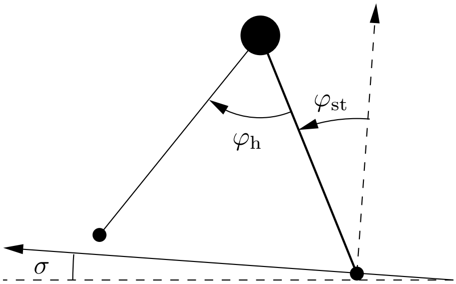


Figure 4.2: *The simplest walker – the most elementary model that describes walking behavior.*

$$\begin{bmatrix} \ddot{\varphi}_{\rm st} \\ \ddot{\varphi}_{\rm h} \end{bmatrix} = \begin{bmatrix} \sin(\varphi_{\rm st} - \sigma) \\ \sin(\varphi_{\rm h})(\dot{\varphi}_{\rm st}^2 - \cos(\varphi_{\rm st} - \sigma)) + \sin(\varphi_{\rm st} - \sigma) \end{bmatrix} \tag{4.3}$$

in which $\varphi_{\rm st}$ is the angle between the stance leg and the floor normal, $\varphi_{\rm h}$ is the relative hip angle and σ is the floor's slope angle. We used 4th order Runge-Kutta to integrate (4.3) with a time step of 0.0125s. At heel strike, the collision with the floor causes the system to lose energy, and the swing leg becomes the new stance leg and vice versa. The impact is modeled as an instantaneous velocity change from the pre-collision state(−) to the post-collision state (+) by:

$$\begin{bmatrix} \dot{\varphi}_{\rm st}^{+} \\ \dot{\varphi}_{\rm h}^{+} \end{bmatrix} = \begin{bmatrix} \cos(2\varphi_{\rm st}^{-}) \\ \cos(2\varphi_{\rm st}^{-})(1 - \cos(2\varphi_{\rm st}^{-})) \end{bmatrix} \dot{\varphi}_{\rm st}^{-} \tag{4.4}$$

This unactuated system is able to passively and stably walk down a slope in a gravity force field with unit magnitude. The slope angle σ is 0.004rad. For this passive walking gait, a footstep takes around 4 seconds (20 time steps).

4.3.2 Learning to walk

Its stability and walking speed can be increased by adding actuation. Because the legs are virtually massless, the action consists of an acceleration of the swing leg instead of a torque. This acceleration has no effect on the movement of the stance leg, only on the swing leg. The agent can choose its action from the range $[-1.2, 1.2]$rad/s^2 in 15 uniformly spaced steps. The state space of the learning agent is spanned by φ_{st}, $\dot{\varphi}_{\mathrm{st}}$, φ_{h} and $\dot{\varphi}_{\mathrm{h}}$. At the start of an episode, the walker is randomly set to an initial condition that is known to contain enough energy to start walking (but not necessarily leads to a stable walking pattern). An episode ends when the walker fell down or after 100 seconds. The rewards are -1 for every action, 50 per meter of footstep length at every footstep and -50 when it falls. By rewarding traveled meters and punishing time, the walker will optimize towards maximum walking speed. In most gaits we observed, a footstep took around 1.6 seconds.

The relevant learning parameters are the learning rate $\alpha = 0.4/16$ where 16 is the number of tilings in the tile coding function approximator used to estimate $Q(s, a)$, the exploration rate $\epsilon = 0.05$ (discounted such that it is 0.01 after 30 simulated hours), time discounting factor $\gamma = 0.99$ and trace decay rate $\lambda = 0.92$. The sampling period was 0.2s. We kept the learning rate constant, which is realistic for a learning robot; it can then continuously adapt to possible slow changes in the environment or its own dynamics, such as changes in friction due to wear and tear of the system.

4.3.3 Test scenarios

We analyzed three types of large disturbances. In each test, we added a specific type of disturbance to the system at random time intervals with an average of one disturbance every 50 time steps.

The first type of large disturbance is a physical perturbation of the system in the form of an instantaneous push, which leads to an outlier in the state transitions (Figure 4.1(a)). In our simulation this was effected by applying a random, instantaneous change in angular velocity of the stance leg, drawn from a uniform distribution over the ranges $[-0.044, -0.038]$rad/s and $[0.038, 0.044]$rad/s. This corresponds to a change in its velocity of roughly 25%-100% depending on the moment of application.

The second type of large disturbance is sensor spike noise. Potentiometers commonly applied to measure joint angles (as a cheaper alternative to magnetic or optical angular sensors) can occasionally lose contact, resulting in spikes in the measurements. Such a disturbance only changes the state reported to the learning module, while the physical system itself remains unchanged (Figure 4.1(b)). In our simulation, an outlier was implemented by replacing the sensor reading of the hip angle by a random value distributed uniformly over the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$rad. In addition, we changed the perception of the hip angular velocity as well, as if it resulted from differentiating the faulty position signal. Note that this results in two samples with a faulty state signal.

The third type of large disturbance is sampling time irregularity. While the motor control policy of most robots runs on a proper real-time operating system, samples may still be delayed or lost due to communication errors or algorithmic calculations taking occasionally longer than normal. This was simulated by omitting samples at random, causing the last chosen action to take twice as long.

4.4 Results and discussion

We present our test results as learning performance graphs, showing the average traveled distance of the walker versus simulation time. After every 20 episodes, a series of 11 test runs – each 100 seconds long – was performed. By measuring traveled distance, an increase in performance in terms of faster walking shows an increase in traveled distance. On the other hand, equally fast but unstable walking results in the walker falling easily, resulting in on average a decrease in traveled distance. Each point in the graph is an average of 20 tests and includes the 95% confidence interval of the average. Usually, the walker quite quickly learns to walk, after which it (slowly) continues to optimize for walking speed. It must be noted that an episode ends immediately when the walker falls, resulting in a (very) low traveled distance. Because these runs are averaged with successful runs, the variance of the performance graphs is quite large.

The performance of learning to walk using the SARSA algorithm without disturbances is shown as baseline in, e.g., Figure 4.8. We see that the walker learned to walk after about 1.5 hours, after which it slowly keeps increasing its average walking speed, which increases its traveled distance during the 100 second test runs. We now compare this result to our different disturbance scenarios.

4.4.1 Push

We first compared a learning process disturbed with random pushes – but undisturbed during the test runs – to undisturbed learning. This allows us to ascertain how well we do on learning the underlying undisturbed problem. In Figure 4.3 we can see that by skipping the learning update for state transitions involving an outlier as described in Section 4.2.4, we do indeed learn the optimal policy. Without rejection however, the performance is significantly worse. Apparently, Q-values of states that regularly visited during walking are severely affected. Figure 4.4 shows the more realistic scenario we get when we include disturbances in the test runs. Again, undisturbed learning and outlier rejection perform similarly (but worse than testing without disturbances, of course), while regular learning in the disturbed system has a lower performance. This indicates that the SARSA(λ) algorithm is unable to treat the infrequent state perturbations as stochastic variations (at the learning rate used here, which proved to be appropriate for undisturbed learning). When disturbing the test runs, learning without disturbances performs slightly worse than learning with disturbances and rejecting them. This can be explained by the fact that the disturbed learner visits a larger part of the state space (i.e., more exotic states), from which it learns to recover, than the undisturbed learner. Note that it might be expected that *not* rejecting outliers could eventually improve performance in the disturbed test runs over rejection, because the system could possibly learn a more cautious walking gait (at the cost of walking speed). In our simulations this was not the case. This effect is more likely to appear when using a lower learning rate and for smaller, more frequent disturbances.

Although we motivated our choice of keeping the learning rate constant (see Section 4.3.2), we tested the effect of slowly reducing the learning rate over time (results not shown here). The low learning rate at the end of the runs allowed regular SARSA(λ) to perform enough averaging to endure the disturbances and reach the same level of performance as outlier rejection. However, it converged more slowly. We also compared the performance of Q(λ)-learning and SARSA(λ), but could not find a significant difference between the two, indicating that outlier rejection is equally applicable to both algorithms.

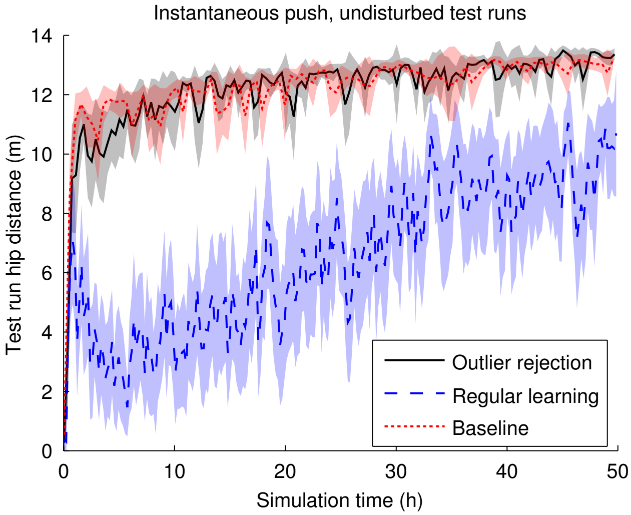


Figure 4.3: *Performance comparison for the push scenario using SARSA and undisturbed test runs. Learning with outlier rejection is compared to regular learning and to the baseline (i.e., undisturbed learning). Outlier rejection restores optimal performance over regular learning for undisturbed test runs.*

4.4.2 Sensor spike noise

Our second testing scenario is sensor spike noise. In Figure 4.5 we see that one spike every 50 time steps (on average) does not significantly reduce the performance from the baseline. As described in Section 4.2.2, the use of eligibility traces is likely to mask the disturbances. The situation changes when we increase the

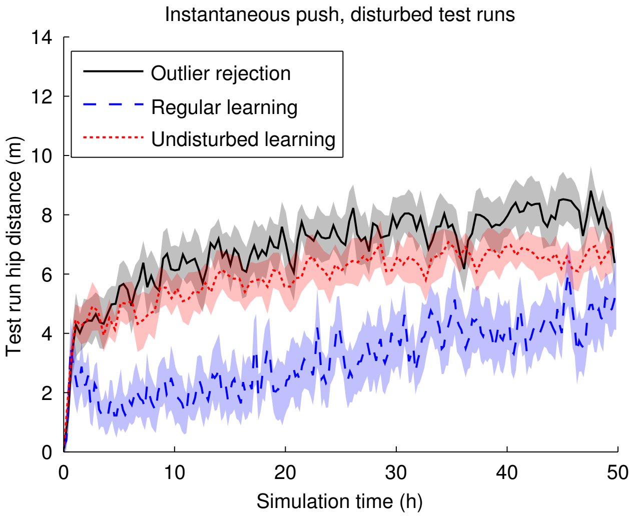


Figure 4.4: *Performance comparison for the push scenario using SARSA and disturbed test runs. Learning with outlier rejection is compared to regular learning and to undisturbed learning. Outlier rejection performs better than undisturbed learning and regular learning. However, all learning schemes show a significantly lower performance compared to undisturbed test runs.*

frequency of the spikes to once every 5 time steps on average: overall performance drops, and only regular learning learns to deal with the disturbances (Figure 4.6). Both outlier rejection and undisturbed learning plateau at a significantly lower performance level. Additionally, the convergence of learning with outlier rejection is slowed by the clearing of the eligibility traces. This shows that for certain disturbances, outlier rejection can actually have a negative effect.

4.4.3 Sampling time irregularity

The final scenario involved omitting samples. As can be seen in Figure 4.7, there is again no significant difference from the baseline. More interestingly, even in the unrealistic case of one lost sample every 5 time steps (on average), the regular learning process achieves only a slightly lower performance than the baseline (Figure 4.8). This indicates that our walking system is quite robust against sampling time irregularity by default and becomes slightly more robust when learning under the influence of disturbances. The frequency of the disturbance now allows it to be treated as stochastic noise.

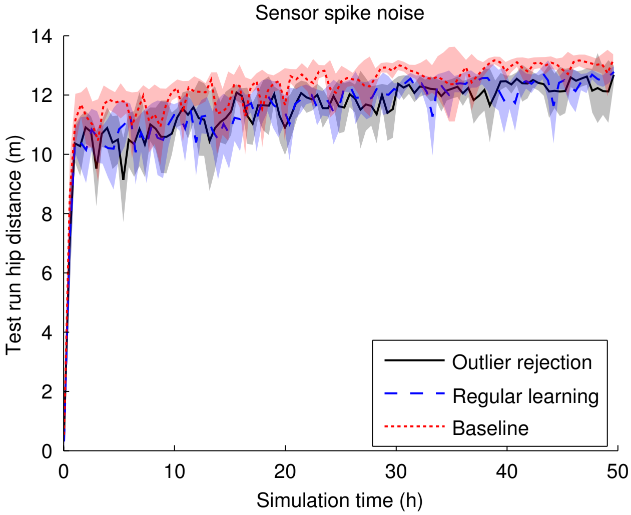


Figure 4.5: *Performance comparison for the sensor spike noise scenario using SARSA and disturbed test runs. Learning with outlier rejection is compared to regular learning and to the baseline (undisturbed learning and test runs). The disturbances have a small effect on the learning performance.*

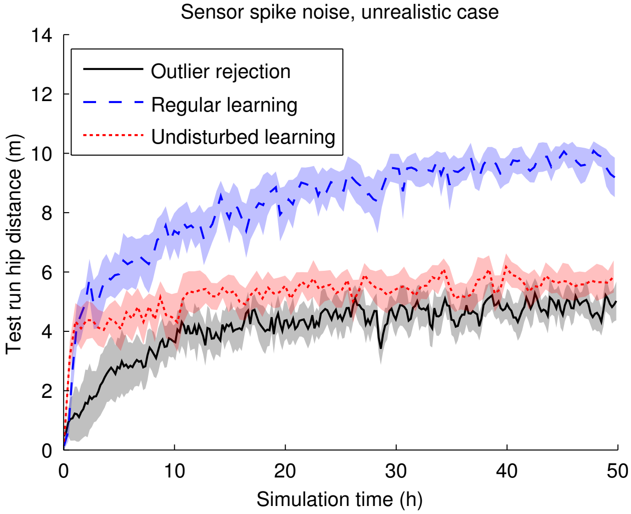


Figure 4.6: *Performance comparison for the sensor spike noise scenario using SARSA and disturbed test runs. Disturbances occurred unrealistically often. Learning with outlier rejection is compared to regular learning and to undisturbed learning. Rejecting outliers performs* worse *than regular learning. Regular learning performs better than undisturbed learning.*

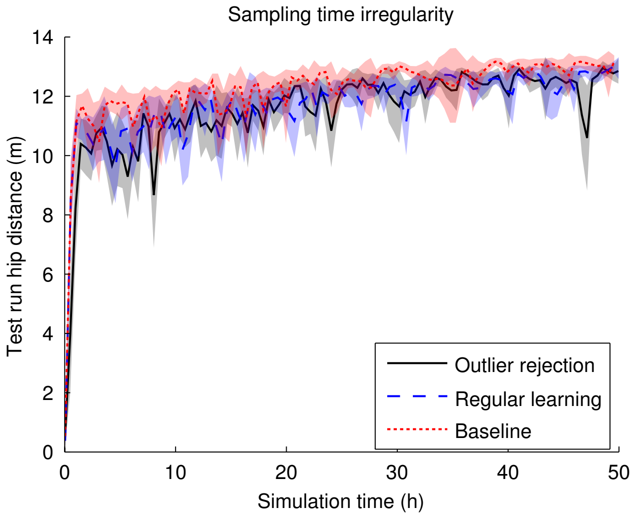


Figure 4.7: *Performance comparison for the sampling time irregularity scenario using SARSA and disturbed test runs. Learning with outlier rejection is compared to regular learning and to the baseline (undisturbed learning and test runs). The disturbances have an insignificant effect on the learning performance.*

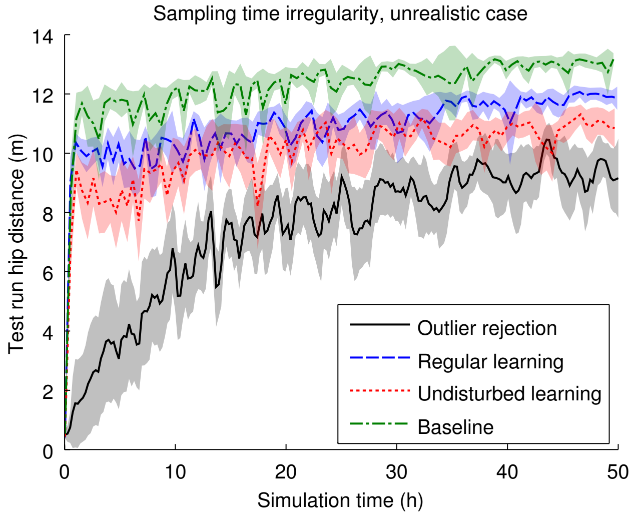


Figure 4.8: *Performance comparison for the sampling time irregularity scenario using SARSA and disturbed test runs. Disturbances occurred unrealistically often. Learning with outlier rejection is compared to regular learning, undisturbed learning and to the baseline (undisturbed learning and test runs). Outlier rejection has the worst performance, while regular and undisturbed learning perform surprisingly well.*

4.5 Conclusions

Stochastic system behavior is part of the stochastic MDP framework and poses no problem for most learning algorithms, other than that it usually results in the need to average over more experience (i.e., using a lower learning rate) and thus longer learning times. However, the effect of large and infrequent disturbances – or outliers – is relatively unknown. Every real system will suffer from outliers to some degree. They can occur in sensor readings, timing or in unexpected interactions with the environment. In this chapter, we evaluated the effects of outliers on a simple simulation model of a walking robot, which learned to walk using SARSA(λ). We tested the effects of three types of outliers: an instantaneous push, a sensor reading outlier, and a sampling time irregularity.

Pushing the walker at random moments, on average once in approx. 6 footsteps, had a dramatic effect on the learning time and system performance. Rejecting the outliers by excluding the faulty state transitions from the learning process completely restored the performance of the walker. After an equal number of practicing hours, the 'ignorant' walker that included the outliers in the learning updates performed roughly half as well as the outlier rejecting walker. This contrasts with the possible expectation that learning under the influence of disturbances would produce a more robust policy; apparently, the size and frequency of the disturbances in this experiment did not allow for them to be treated as stochastic variations of the underlying MDP.

The introduction of random spike noise on the sensor reading of the hip angle, on average once every 50 measurements, had an undetectable effect on the learning agent. When their frequency was increased ten times (unrealistic), outlier rejection actually resulted in a *decrease* in learning speed. This can be explained by the fact that we excluded outliers in SARSA(λ) by clearing the eligibility traces, thus on average once in 5 samples, which slowed down learning. Doubling the sampling time randomly, on average every 50th sample, also had an undetectable effect on the learning agent. When increasing their frequency ten times (unrealistic), the effect became noticeable but was still surprisingly small. Again, rejecting outliers by clearing the eligibility traces led to a large drop in learning speed. The rejection process had a much more negative impact on the learning performance than the outliers themselves.

We can conclude that for this simple model, large disturbances of the actual system state through unexpected interaction with the environment have by far the largest influence on the learning process, compared to timing and sensor outliers.

Chapter 5

Using Independent Learners for individual actuators

Section 2.4 showed that control delay can slow down the learning process and at worst make it diverge. The experiments in Section 3.3 on simulation of robot Leo confirmed this. In this chapter, we take a multiagent RL approach in which each actuator of the robot is controlled by a separate learning agent. Because each agent has a smaller, one dimensional action space, consulting the policy requires much less computation, which reduces the control delay in real systems. In addition, memory requirements are reduced. This chapter is based on (Troost, Schuitema, and Jonker, 2008).

5.1 Introduction

A well known problem in scaling RL towards more complex systems, like humanoid robots, is the large number of inputs (sensors) and outputs (actuators) that are spanning the state-action space of these systems. Generally, the larger the state-action space of a system, the longer it takes and the more memory is required to find a satisfactory control policy with RL. For RL methods that derive the policy directly from the (action-)value function, a large action space possesses an additional computational disadvantage. Suppose a robot has M motors, each of which can be controlled in N discrete actuation steps. Selecting the best action in a certain state involves evaluating the (action-)value function N^M times. This can become computationally heavy, especially when using computationally intensive function approximation techniques to learn in continuous state spaces. A lengthy computation causes control delay – delay between the state measurement and the execution of the selected action – which was shown to be detrimental to the learning process (Section 2.4).

In this chapter, we present an approach in which the action space is *decomposed* by assigning a learning agent to each individual actuator. These heterogeneous agents with differing action spaces learn to achieve the overall system goals by implicitly cooperating to achieve their common goal: the global reward function. Together the agents form a multiagent system (MAS). Each agent's state space consists of the full state space as would be used in single agent solutions, but does not include any information on action selection done by the other agents. Claus and Boutilier (1998) call this approach Independent Learners (IL). During policy evaluation, each agent selects its own action, without any form of negotiation or central coordination. In our example of M motors with N discrete actions each, this will lead to evaluating and storing $M \cdot N$ values instead of N^M; an increase that is linear in the number of actuators instead of exponential. Thereby, this MAS approach offers a reduction in computational and memory requirements. Furthermore, it is suitable for parallel or distributed computing implementations, where the agents learn and select actions simultaneously (Stone and Veloso, 2000).

In the IL approach, the state transitions for each agent do not only depend on the actions taken by that agent itself, but also on the actions of the other agents. Since these are not included in the agent's state signal, the state transition function for each independent agent is in effect non-stationary (Laurent, Matignon, and Fort-Piat, 2011), thereby violating the Markov property. Therefore, problems can be expected when individual agents are using an RL algorithm that assumes a stationary MDP. Nevertheless, this approach has been successfully applied to a number of simulation domains, including elevator group control (Crites and Barto, 1998), the control of a two-link manipulator (Buşoniu, De Schutter, and Babuška, 2006) and distributed micro-manipulation (Matignon et al., 2010).

We test the IL approach on three different simulations of robotic setups, with the aim to reduce computational and memory requirements: a two-link manipulator, and bipedal walking robots META and Leo. Whereas the two-link manipulator and META have two motors, Leo has three actuators under RL control[1]. We compare classical (single-agent) Q(λ) and SARSA(λ) with their multiagent counterparts using the IL approach. We also test two modifications to IL: using synchronized exploration, where explorative actions of all agents coincide, and using Lenient Learning (Panait, Sullivan, and Luke, 2006), a method for multiple agents to be lenient to each other's actions. Since all three systems have continuous state spaces, we use tile coding function approximation to estimate action-value functions.

This chapter is organized as follows. In Section 5.2, we explain the theory of multiagent RL (MARL) and the proposed IL method. In Section 5.3, we present the results obtained with the simulation of a two-link manipulator. In Section 5.4, the results for simulated bipedal walking robot META are presented and Section 5.5 presents the results obtained with the simulation of Leo. We finally present a discussion in Section 5.6 and our conclusions in Section 5.7.

---

[1] Leo has 7 servo motors, of which 3 are controlled by RL and 4 by conventional control laws.

5.2 Theory

In this section, we outline our approach, formalize it and discuss the available theory.

5.2.1 Approach

Reinforcement Learning (RL) is designed to find optimal control policies for Markov Decision Processes (MDPs, see Section 2.1). We propose to decompose the action space of the system as defined in the original MDP. We do this by assigning a learning agent to each actuator, thereby creating a multiagent system (MAS). The agents act in a fully cooperative setting, i.e., they have the same objective by means of the single, global reward function from the original MDP; all agents receive the same reward, which depends on the actions of all agents together. However, each agent learns to control its actuator without communicating with the other agents. Therefore, coordination between the agents – needed to select the globally optimal action – is not explicit, as can be achieved through communication or social convention, but needs to emerge through learning and interaction. Each agent receives the full state of the system (which excludes the actions chosen by the other agents).

We are considering (robotic) systems with a continuous state space and M actuators, each allowing a set of discrete action commands A^m. A single agent would have to learn to control the joint action space containing $|A^1| \cdot |A^2| \cdot \ldots \cdot |A^M| = \prod\limits_{m=1}^{M} |A^m|$ action combinations. To this end, the agent needs to store an equal number of Q-values and evaluate these Q-values when consulting its policy. In the multiagent case, where each actuator is controlled by a separate agent, there are M agents that together store and consult $|A^1| + |A^2| + \ldots + |A^M| = \sum\limits_{m=1}^{M} |A^m|$ Q-values. Memory requirements [2] and computation time thereby become linear instead of exponential in M. Memory requirements and computation time during action selection are reduced by a factor:

$$\frac{\prod\limits_{m=1}^{M} |A^m|}{\sum\limits_{m=1}^{M} |A^m|} \tag{5.1}$$

We now formalize this idea and discuss temporal difference learning algorithms for this setting.

---

[2] At this point, we do not consider function approximation with generalization between actions.

5.2.2 MMDP

For our proposed approach, the MDP framework needs to be extended to a multiagent version – the MMDP – in which several learning entities are active in the same environment. This is done by Boutilier (1996) by including all agents in the tuple:

$$\langle S, M, \{A^1, ..., A^M\}, T, R \rangle, \tag{5.2}$$

where M is the number of agents, A^m is the action space available to agent m, $T : S \times A^1 \times \ldots \times A^M \times S \to [0, 1]$ is the state transition probability density function and $R : S \times A^1 \times \ldots \times A^M \times S \to \mathbb{R}$ is the real-valued reward function.

One can attempt to solve this MMDP by solving a set of local decision processes, one for each agent m, defined by a 4-tuple containing partial action space A^m, the entire set of states S, a time-dependent state transition probability density function $T_k^m : S \times A^m \times S \to [0, 1]$ and a time dependent reward function $R_k^m : S \times A^m \times S \to \mathbb{R}$:

$$\langle S, A^m, T_k^m, R_k^m \rangle. \tag{5.3}$$

While T and R are global, stationary functions, the individual functions T_k^m and R_k^m are non-stationary; they depend on time step k for the following reason. When the system state s_k transitions to s_{k+1}, this is the result of the *joint* action of all agents. However, because the agents' policies are continually adjusted during learning, the actions of the agents vary over time. Therefore, from a single agent's perspective, state transitions and rewards also change over time, even though they are generated by the global and stationary functions T and R. These 4-tuples, thus, do not possess the Markov-property (Laurent, Matignon, and Fort-Piat, 2011). Below we propose learning algorithms for these agents and discuss whether convergence can be expected.

5.2.3 MA-Q(λ) and MA-SARSA(λ)

Q-learning (see Section 2.2) is an off-policy method, which can be extended to a multiagent version – MA-Q – with the aim to solve the previously proposed MMDP. We propose to let a set of M Independent Learners (IL) (Claus and Boutilier, 1998) estimate their own action-value function $Q^m(s, a^m)$, spanned by $S \times A^m$, using a local Q-learning update rule. Each agent calculates its local TD-error $\delta^m_{\mathrm{TD_Q}}$ according to:

$$\delta^m_{\mathrm{TD_Q},k+1} = r_{k+1} + \gamma \max_{a' \in A^m} Q_k^m(s_{k+1}, a') - Q_k^m(s_k, a_k^m) \tag{5.4}$$

This approach is equivalent to Distributed Q-learning (Lauer and Riedmiller, 2000). To simplify the discussion, we assume that each $Q^m(s, a^m)$ is estimated using linear function approximation. This can be done by updating every element $\theta^{m,i}$ of each agent-specific feature parameter vector $\boldsymbol{\theta}^m$ every time step according to:

$$\theta_{k+1}^{m,i} = \theta_k^{m,i} + \alpha \delta_{\mathrm{TD},k+1} \phi^{m,i}(s_k, a_k) \tag{5.5}$$

with $\boldsymbol{\phi}^m(s_k, a_k)$ the vector of agent-specific features (basis functions) and $\theta_0^{m,i}$ arbitrarily initialized. Eligibility traces can be implemented analogously to (2.20) and (2.21) – thereby creating MA-Q(λ) – by using an agent-specific vector $\boldsymbol{e}^m$ of eligibility traces.

SARSA, an on-policy algorithm, estimates the total expected sum of rewards of choosing action a in state s and following the *current* policy afterwards. A multiagent, IL version – MA-SARSA – can be defined by using (5.5) with the following TD-error definition:

$$\delta^m_{\mathrm{TD_Q},k+1} = r_{k+1} + \gamma Q^m_k(s_{k+1}, a^m_{k+1}) - Q^m_k(s_k, a^m_k) \tag{5.6}$$

MA-SARSA can be extended with eligibility traces analogously to (2.20) and (2.21) to create MA-SARSA(λ).

5.2.4 Policy

Several action selection policies can be used during learning to incorporate different exploration strategies. We used the ϵ-greedy policy, denoted $\pi_{\epsilon-\mathrm{greedy}}$, with exploration rate ϵ according to (2.9). In the case of MA-Q(λ) and MA-SARSA(λ), each agent m has its own policy and thus independently decides which action a^m to take. Therefore, exploration is also independent. An IL version of the ϵ-greedy policy is:

$$\pi^m_{\epsilon-\mathrm{greedy}}(s_k, a^m_k) = \begin{cases} 1 - \epsilon + \epsilon/|A^m| & \text{if } a^m_k = a^m_{k,\mathrm{greedy}} \\ \epsilon/|A^m| & \text{if } a^m_k \neq a^m_{k,\mathrm{greedy}} \end{cases} \tag{5.7}$$

with $a^m_{k,\mathrm{greedy}}$ the agent-specific greedy action at time step k:

$$a^m_{k,\mathrm{greedy}} = \arg\max_{a' \in A^m} Q^m(s_k, a') \tag{5.8}$$

In practice, action a^m_k is selected as follows:

$$\begin{aligned} & x^m \sim U(0,1), \\ a^m_k = & \begin{cases} \sim U(A^m) & \text{if } x^m < \epsilon \\ a^m_{k,\mathrm{greedy}} & \text{if } x^m \geq \epsilon \end{cases} \end{aligned} \tag{5.9}$$

where $x^m \in [0, 1]$ is an agent-specific random number drawn from a uniform distribution. Multiagent Q-learning for stateless systems with the ϵ-greedy policy is described by Gomes and Kowalczyk (2009).

Alternatively, one may choose to *synchronize* exploration between the agents by always exploring simultaneously:

$$\begin{aligned} & x \sim U(0,1), \\ a^m_k = & \begin{cases} \sim U(A^m) & \text{if } x < \epsilon \\ a^m_{k,\mathrm{greedy}} & \text{if } x \geq \epsilon \end{cases} \end{aligned} \tag{5.10}$$

with $x \in [0, 1]$ a random number drawn once per time step. Strictly speaking, synchronizing exploration breaks the independence of the agents. However, the goal of our approach is not the independence of the agents per se, but the computational benefits this approach offers, which are unchanged by this modification.

5.2.5 Problems of ILs

Theoretical literature on heterogeneous ILs mainly focuses on single state MMDPs with two or three actions available to each agent (Claus and Boutilier, 1998; Lauer and Riedmiller, 2000; Panait, Sullivan, and Luke, 2006; Panait, Tuyls, and Luke, 2008). The problems faced and sometimes solved in these papers may also (at least to some extent) occur in our proposed method. Because the learning problem for each agent is not stationary (see Section 5.2.2), each agent is faced with a moving target learning problem: the best policy depends on the other agents' policies. If there is no coordination between the agents, which is the case for ILs, some specific situations may give rise to convergence problems. Below we give an example in which convergence to a suboptimal solution is likely to occur.

One problem often occurring in multiagent RL is the problem of obtaining full state information. Most literature on multiagent RL is about multiple, often homogeneous robots having to work together to solve a problem (see, e.g., Panait and Luke (2005)). Obtaining full state information involves (synchronized) communication between all robots and can be cumbersome. In our approach, this problem is not present. We use heterogeneous Independent Learners that are active within a single robot; obtaining full state information is not more difficult than in single agent approaches.

Convergence to suboptimal solutions

To illustrate the problem of convergence to suboptimal solutions, we borrow an example from (Claus and Boutilier, 1998): the 'climbing game'. It is a deterministic, stateless (i.e., single state) learning system with two agents. The two agents – one with action space a_1, a_2, a_3, the other with action space b_1, b_2, b_3 – choose an action independently, resulting in a global reward for both agents, after which the game ends. The reward scheme is given in Table 5.1.

In this game, two Nash equilibria exist: an optimal one at (a_1, b_1) (with reward 11) and a suboptimal one at (a_2, b_2) (with reward 7). A Nash equilibrium (Nash Jr, 1951) in this context is formed by a set of agent policies such that no individual agent can improve its policy if the other agent's policy remains the same. Instead of always reaching the optimal equilibrium, the IL agents can converge to a suboptimal solution, depending on the specific learning parameters used. Suppose that both agents start with policies in which all actions have equal probabilities (i.e., uniformly random action selection). Learning will quickly lead to new policies in which (a_3, b_3) is the preferred joint action, since it results in the highest average reward under the agent's initial policies. From this new

Table 5.1: *The rewards belonging to actions in the 'climbing game': a stateless learning problem with two agents that is known to give convergence problems when solved by Independend Learners. One agent has action space* a_1, a_2, a_3*, the other has action space* b_1, b_2, b_3.

	a_1	a_2	a_3
b_1	11	-30	0
b_2	-30	7	6
b_3	0	0	5

equilibrium, a unilateral change in the second agent's policy from b_3 to b_2 is an improvement, which can be easily reached by exploration. Similarly, from this new policy, a unilateral change in the first agent's policy from a_3 to a_2 is again an improvement. However, because of the high penalties of -30 at (a_1, b_2) and (a_2, b_1), it is unlikely that the agents will further change their policies. Unless both agent's policies *simultaneously* change into preferring (a_1, b_1), the agents will keep preferring (a_2, b_2). This change is unlikely to happen, especially for small values of the exploration rate and learning rate. This illustrates that ILs can have a tendency to 'settle' for suboptimal solutions. Depending on the initial policies and the learning parameters, the specific suboptimal solution reached may vary. Situations similar to the 'climbing came' can occur in multi-state (i.e., sequential decision making) systems as well.

The optimistic assumption

Lauer and Riedmiller (2000) adjust the IL with the aim to cope with such problems without adding coordination. Instead of the update rule in (5.5), an 'optimistic' assumption is made where Q-values are only updated if the TD-error is positive, thereby ignoring mistakes by other agents that lead to a lower expected return. This idea can be extended to systems with multiple states and function approximation. Updating feature parameter values with this optimistic assumption can be done as follows:

$$\theta_{k+1}^{m,i} = \begin{cases} \theta_k^{m,i} + \alpha\delta_{\mathrm{TD},k+1}\phi^{m,i}(s_k, a_k) & \text{if } \delta_{\mathrm{TD},k+1} > 0 \\ \theta_k^{m,i} & \text{if } \delta_{\mathrm{TD},k+1} \leq 0 \end{cases} \tag{5.11}$$

This takes care of the problem in the climbing game. Lauer and Riedmiller (2000) describe a way to keep track of the first best policy in each agent to make sure that in a penalty game, both agents choose the same (the first encountered) optimum. With these two additions, convergence to the optimal solution is again guaranteed under the same conditions as with the single-agent MDP.

Within a stochastic MMDP, fluctuations of Q-values occur due to stochastic state transitions and stochastic rewards. Such fluctuations, i.e., positive and negative adjustments, cannot be distinguished from fluctuations due to the changing policies of the other agents. Using the optimistic assumption of (5.11) in this case leads to an overestimation of the total expected reward, resulting in a loss of convergence (Lauer and Riedmiller, 2000).

Panait, Sullivan, and Luke (2006) propose Lenient Learning for stateless problems. This is a combination of the update rules (5.5) and (5.11). In the beginning of the learning trial, the optimistic assumption is made to make sure optimal equilibria are found. After these are discovered, the lenience towards the other agents is tuned down, returning the update to the original one (5.5). This transition is smoothly made with a Boltzmann probability that depends on the number of times a state-action pair is visited. Here we extend Lenient Learning to a multi-state method with linear function approximation as follows:

$$\begin{gathered} x \sim U(0,1), \\ \theta_{k+1}^{m,i} = \begin{cases} \theta_k^{m,i} + \alpha\delta_{\mathrm{TD},k+1}\phi^{m,i}(s_k,a_k) & \text{if } \delta_{\mathrm{TD},k+1} > 0 \text{ or } x > \ell_k^{m,i} \\ \theta_k^{m,i} & \text{otherwise} \end{cases} \end{gathered} \tag{5.12}$$

with $x \in [0,1]$ a random variable, and $\ell_k^{m,i}$ the feature dependent lenience defined as:

$$\begin{aligned} \ell_k^{m,i} &= 1 - e^{-\kappa\tau_{\ell,k}^{m,i}} \\ \tau_{\ell,k}^{m,i} &= \beta\tau_{\ell,k-1}^{m,i} \end{aligned} \tag{5.13}$$

where κ is the lenience parameter and $\tau_\ell^{m,i}$ the lenience temperature of feature $\phi^{m,i}$, which is decreased with a discounting factor $\beta \in [0,1]$ each time $\phi^{m,i}$ is visited. At initialization, $\tau_{\ell,0}^{m,i} = 1$.

Implications for our robots

For multi-state problems such as robots, convergence to the optimal solution is not guaranteed when using ILs due to the problems just described. Especially in situations where an improvement in the expected return requires a simultaneous change of all agents' policies, whereas a unilateral policy change of a single agent would decrease the expected return, it is probable that the ILs converge to a suboptimal solution. Therefore, care needs to be taken when implementing ILs as a substitute for single agent RL. At this point, however, we cannot fully predict the complications of ILs in multi-state systems; the behavior of multi-state ILs is a topic of recent research (Vrancx, Tuyls, and Westra, 2008; Hennes, Tuyls, and Rauterberg, 2008; Hennes, Tuyls, and Rauterberg, 2009). In the following sections, we show the differences between ILs and single agent RL empirically by means of experiments with several robot simulations. To help solve the possible convergence problems discussed in this section and in Section 5.2.5, Lenient Learning according to (5.12) is tested on one of the test setups: the two-link manipulator.

5.3 Two-link manipulator

In this section, we compare the performance of MA-Q(λ) and MA-SARSA(λ) against their single agent counterparts for the two-link manipulator, a robotic setup with two actuators, shown in Figure 2.7. We show the performance of synchronized exploration and Lenient Learning as well. Whereas Buşoniu, De Schutter, and Babuška (2006) applied decentralized value iteration to this setup, here we apply temporal difference learning. Its fourth-order non-linear dynamics are given in (2.34); see, e.g., (Buşoniu et al., 2007) for a more detailed description of the system. We numerically integrated (2.34) using the Runge-Kutta algorithm with time step $T_i = 0.01$s.

5.3.1 Learning

In our learning setup, we define two agents, one for each motor. Both agents receive the complete system state, but no information about the other agent's action. Their state vectors $\boldsymbol{s}^i$ are:

$$\boldsymbol{s}^1 = \boldsymbol{s}^2 = \begin{pmatrix} \varphi_1 \\ \dot{\varphi}_1 \\ \varphi_2 \\ \dot{\varphi}_2 \end{pmatrix} \tag{5.14}$$

Each agent controls a single motor through their action a^i:

$$a^1 = \tau_1 \tag{5.15}$$

$$a^2 = \tau_2 \tag{5.16}$$

The task of the system is to accomplish $\varphi_1 = \varphi_2 = \dot{\varphi}_1 = \dot{\varphi}_2 = 0$ as fast as possible. To this end, a reward is given when the angles and angular velocities are within a small region around 0: $|\varphi_j| < 0.17 \wedge |\dot{\varphi}_j| < 0.2,\ \forall\ j \in \{1, 2\}$. Furthermore, each time step a time penalty is given. The reward r^i_k is equal for both agents:

$$r^1_k = r^2_k = \begin{cases} 100, & \text{if } |\varphi_{j,k}| < 0.17 \wedge |\dot{\varphi}_{j,k}| < 0.2,\ \forall\ j \in \{1,2\} \\ -1, & \text{all other cases (time penalty)} \end{cases} \tag{5.17}$$

The agents perform a Q-learning update rule simultaneously at each time step according to (5.5). The the sampling period $h = 0.05$s. The learning rate $\alpha = 0.4$, the exploration rate $\epsilon = 0.05$, the discount factor $\gamma = 0.98$ and the trace discount factor $\lambda = 0.92$. Both agents use tile coding function approximation with 16 tilings to approximate $Q^i(s, a^i)$. The action space is discrete. There is no generalization between actions, only between states. The tile widths for the state space are 1/12rad in the φ_1 and φ_2 dimensions and 1/6rad/s in the $\dot{\varphi}_1$ and $\dot{\varphi}_2$ dimensions. The learning results will be compared with the single-agent case in which one agent controls both motors, i.e., where $\boldsymbol{a} = (\tau_1, \tau_2)^T$.

5.3.2 Results

In order to test the learning performance of the two-link manipulator task, we defined a test set of 15 different initial conditions from which the manipulator has to complete its task. We regularly let the system perform all the tasks from the test set and monitor the number of successfully completed tasks. During test runs, exploration is disabled. In the next sections, we test various learning algorithms and settings and compare their performance. The learning process is repeated several times with different random initializations of the action-value function.

Single-agent vs. multiagent Q(λ)

In this test we compare the performance of single-agent Q(λ) against multiagent Q(λ) on the two-link manipulator task. Each actuator's action space is uniformly discretized into 7 steps. The result can be found in Figure 5.1. The learning curve of the SA case is slightly, but barely better than the MA case. In terms of performance, MA-Q(λ) proves to have no significant disadvantage over SA. The advantage of MA-Q(λ) in terms of reduced memory requirements is illustrated in Figure 5.2, showing the relative memory requirements for several sizes of the action space, where the required memory is computed from the number of function approximation feature parameters involved in the learning updates. Because we

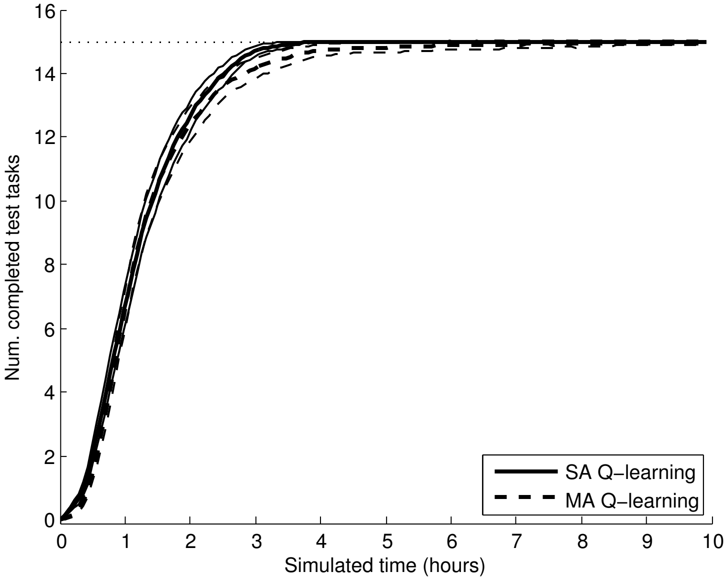


Figure 5.1: *Single-agent (SA) Q(λ) compared with multiagent (MA) Q(λ) for the two-link manipulator task. The average over 400 independent runs is plotted, including* 95% *conficence bounds on the average.*

use an equal number of discrete actions m for both agents, according to (5.1) we can expect a memory reduction of $\frac{m}{2}$ in the multiagent case compared to the single-agent case. As can be seen from the figure, the simulations follow the expected memory reduction factor. The second advantage of the MA approach, a reduction of computation time, becomes clear in Table 5.2, where the relative computation time of an average learning step is shown for SA and MA learning. It follows the same trend as for the memory reduction, with a bias for the overhead of other calculations than the best action search.

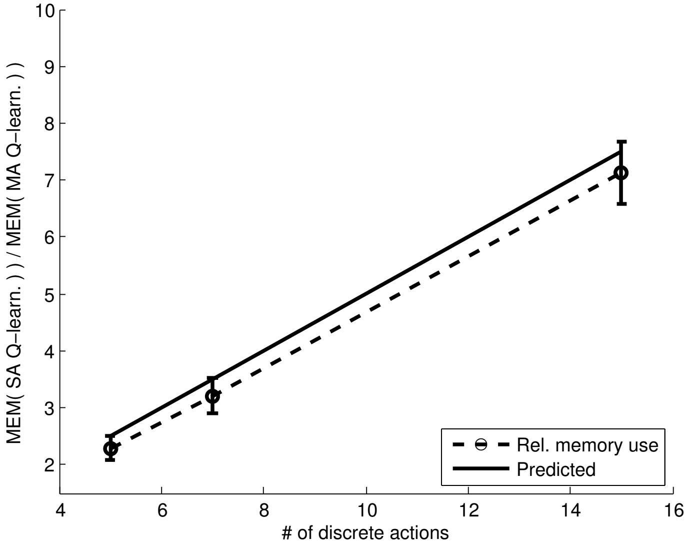


Figure 5.2: *Relative memory use of single-agent (SA) Q(λ) versus multiagent (MA) Q(λ) setup for the two-link manipulator task, for several sizes of the action space (equal for both agents). The average over 25 independent runs is shown, including* 95% *error bars. In this system with two actuators, the MA approach realizes a memory reduction that is linear in* $|A^i|$.

Table 5.2: *Relative computation time of the learning step of SA-Q(λ) versus MA-Q(λ) for the two-link manipulator with m actions for each agent.*

	$m = 5$	$m = 7$	$m = 15$
SA/MA	2.1	3.2	7.1

MA-SARSA(λ) vs. MA-Q(λ)

It could be that using on-policy SARSA(λ) for each agent increases its ability to perform and learn under the changing policy of the other agent. A comparison between single-agent SARSA(λ) and multiagent SARSA(λ) can be found in Figure 5.3. As with Q(λ), SA-SARSA(λ) is slightly, but significantly faster in reaching the final performance. Furthermore, the effect of using SARSA instead of Q-learning in the MA case can be seen in Figure 5.4. There is no significant difference between SARSA and Q-learning.

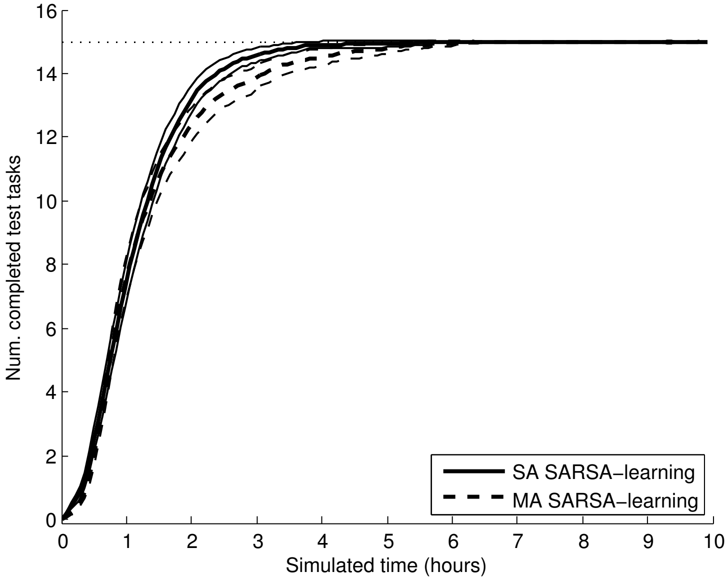


Figure 5.3: *Single-agent (SA) SARSA(λ) compared with multiagent (MA) SARSA(λ) for the two-link manipulator task. The average over 400 independent runs is shown, including* 95% *confidence bounds on the average.*

Synchronized exploration

In the Independent Learner MA setup, all agents explore independently according to (5.9). However, it could be beneficial to synchronize exploration by letting both agents solely perform explorative actions simultaneously using (5.10). This possibly reduces the uncertainty in the policy of the other agents. Additionally, chances of escaping from a local maximum (for instance in situation like the 'climbing game' in Table 5.1b.) are possibly increased when all agents deviate from their current policy simultaneously. We compared the independent exploration strategy with the synchronized exploration strategy. The result can be found in Figure 5.5. It however, did not result in significantly better results. Apparently,

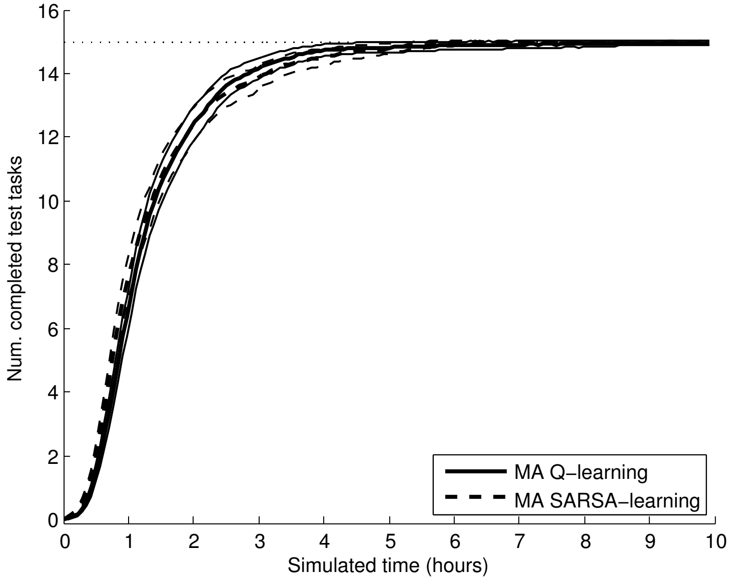


Figure 5.4: *Multiagent (MA) Q(λ) compared with multiagent (MA) SARSA(λ) for the two-link manipulator task. The average over 400 independent runs is shown, including 95% confidence bounds on the average.*

the uncertainty in each agent's policy due to learning is more prominent than the exploration component, at least at the start of learning where performance is still climbing rapidly. Another explanation could be that there are not many suboptimal (Nash-)equilibria in this action-value function.

Lenient learning

As explained in Section 5.2.5, Lenient Learning might help the learning process by ignoring negative value updates in the beginning of learning. This would prevent one mistake of an agent to discourage the right policy of the other agent. A comparison between multiagent Q(λ) and multiagent Lenient Q(λ), with a lenience factor $\ell(s,a)$ for each state-action pair, with $\kappa = 2.0$ and a lenience temperature discounting factor $\beta = 0.95$, discounting at each visit, can be found in Figure 5.6. As can be seen, Lenient Learning can result in a significant improvement. Tuning of the lenience parameters could potentially increase this improvement even further.

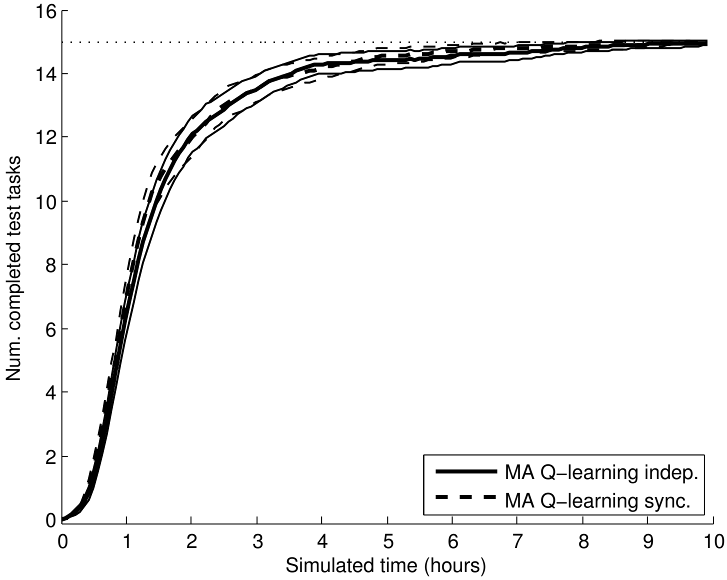


Figure 5.5: *Multiagent (MA) Q(λ) on the two-link manipulator comparing independent exploration and synchronized exploration between the agents. The average over 400 independent runs is shown, including the* 95% *confidence bounds on the average. Synchronized exploration has no significant benefit over independent exploration.*

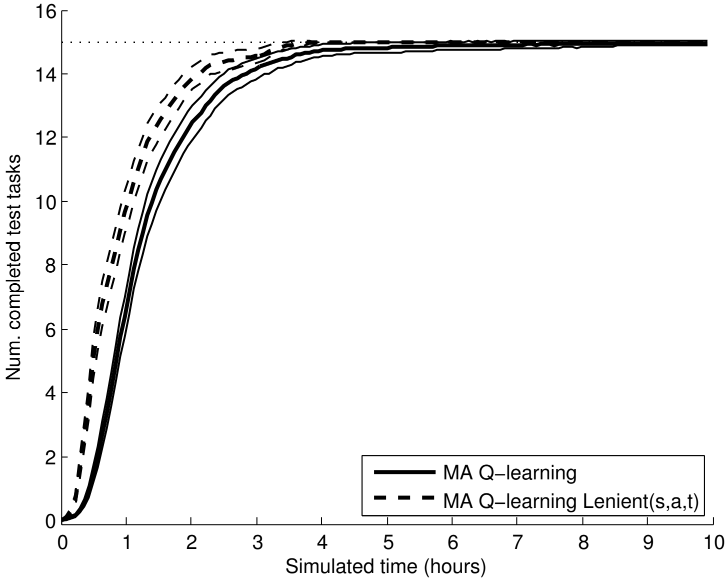


Figure 5.6: *Multiagent (MA) Q(λ) compared with multiagent (MA) Q(λ) with lenience for the two-link manipulator task (average over 400 independent runs).*

5.4 Meta: a bipedal walking robot

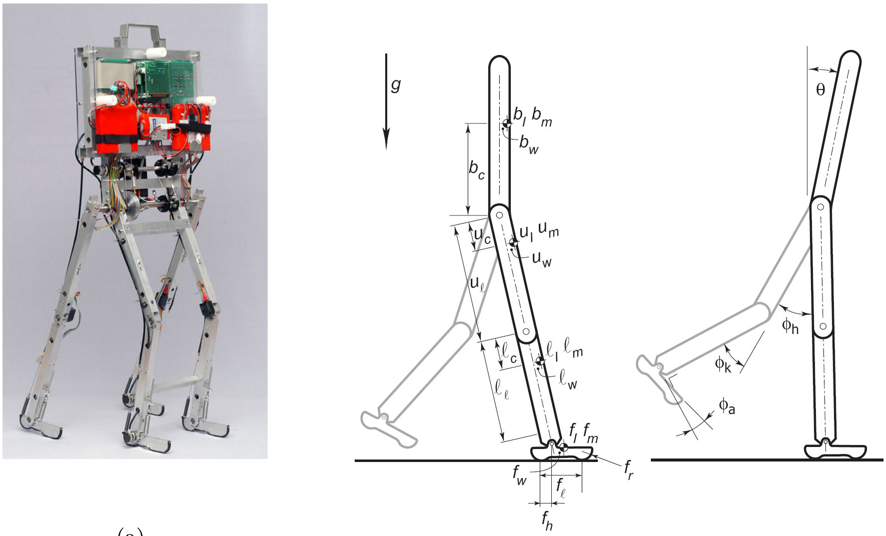
The second test setup on which we evaluate ILs is the simulation of META, a limit cycle walking prototype. Like the two-link manipulator, META has two actuators. However, its state space is of much higher dimensionality and its dynamics are more complex.

5.4.1 Model

The construction of META (Schuitema et al., 2005), see Figure 5.7(a), is based on the concept of limit cycle walking (Hobbelen, 2008). With the concept of limit cycle walking, it is possible to construct a fully passive walking robot that can walk down a shallow slope (for energy input) without any actuation or control (Collins, Wisse, and Ruina, 2001; McGeer, 1990), by carefully choosing the mass distributions and leg lengths. By adding actuation, the robustness increases and the energy input can come from an external source so that the robot can walk on flat terrain. Because META was designed according to the limit cycle walking concept, walking is a natural movement for the robot. META is effectively a two-dimensional walking robot by using two pairs of parallel legs, which remove the sideways stability problem. The version of META that was modeled in (Schuitema et al., 2005) had one hip motor and a special mechanical construction that always kept the upper body upright, at an angle that bisects the angle between both upper legs. The prototype we model here has undergone hardware modifications to equip it with two hip motors. One motor controls the joint between the upper body and the left upper leg, the other motor controls the joint between the upper body and the right upper leg. Each motor can apply a torque to its joint between −10Nm and +10Nm. The prototype was modeled in the Open Dynamics Engine rigid body simulator (Smith et al., 2006) as a 7-link 2D model, see Figure 5.7(b). The joints are modeled by stiff spring-damper combinations. The knees are provided with a hyperextension stop and a locking mechanism which is released just after the start of the swing phase (i.e. right after making a footstep). Contact between the foot and the ground is also modeled by a tuned spring-damper combination which is active whenever part of the foot is below the ground. The model of the foot mainly consists of two cylinders at the back and the front of the foot. A set of physically realistic parameter values were derived from the prototype, see Table 5.3.

5.4.2 Learning

The full state space of the robot consists of the angle and angular velocity of all seven body parts, i.e. 14 state dimensions. Because the feet have relatively small mass and inertia, we assume that the state transitions and the rewards of the system do not significantly depend on the angles and angular velocities of the feet. Therefore, we do not include them in the state space, which leaves 10



(a)

(b)

Figure 5.7: *Bipedal walking robot META. a. The prototype. b. Two-dimensional 7-link model; left the parameter definition, right the degrees of freedom (DoFs). Only the DoFs of the swing leg are given, which are identical to the DoFs of the other leg.*

Table 5.3: *Physical parameters of the model of META.*

	Body(b)	Up.leg(u)	Lo.leg(ℓ)	Foot(f)
Mass m [kg]	8	0.7	0.7	0.1
Mom. of in. I [kgm^2]	0.11	0.005	0.005	0.0001
Length l [m]	0.45	0.3	0.3	0.06
Vert. offset CoM c [m]	0.2	0.15	0.15	0
Hor. offset CoM w [m]	0.02	0	0	0.015
Foot radius f_r [m]	-	-	-	0.02
Foot hor. offset f_h [m]	-	-	-	0.015

input dimensions. We assign a separate learning agent to each motor to create a multiagent system with two agents. Each agent has the same state space consisting of 10 input dimensions and an action space consisting of one motor torque, discretized in 7 steps between −10Nm and +10Nm.

The task of META is to learn to walk with the highest possible forward velocity by actuating both hip motors. This requires the simultaneous coordination of the legs to make correct footsteps, as well as the coordination of the upper body; a task that is much more difficult than the learning task solved in (Schuitema et al., 2005).

The following rewards are used. Whenever the robot makes a footstep, a reward is given that is proportional to the length of the footstep. Together with a time discount factor γ close to 1, the robot will optimize for progressing as many meters in as little time as possible. A footstep is defined as the moment when the foot of the swing leg touches the ground while the hip angle is between 0.1rad and 0.61rad. These values are the minimum and maximum size of a step that allow walking in our model. Furthermore, a penalty is given when the robot falls. The rewards are based on the performance of the system as a whole, not on the behavior of a single-agent. To make a footstep, cooperation between both agents is required. Both agents receive the same reward r_k:

$$r_k^1 = r_k^2 = \begin{cases} 500 \text{ per meter}, & \text{if footstep made, } 0.1 < |\varphi_{hip,k}| < 0.61 \\ -10, & \text{if the robot falls} \end{cases} \tag{5.18}$$

The agents perform a Q-learning update rule simultaneously at each time step according to (5.5). The sampling period $h = 0.018$s. The learning rate $\alpha = 0.5$, the exploration rate $\epsilon = 0.05$, the time discount factor $\gamma = 0.995$ and the trace discount factor $\lambda = 0.92$. Both agents have their own tile coding function approximator with 16 tilings. Generalization is present between states as well as between actions. The tile widths are the same for both agents; for the upper leg angles: 1/6rad, the upper leg angular velocities: 1/2rad/s, the lower leg angles: 1/2.1rad, the lower leg angular velocities: 1/1.65rad/s, the body angle: 1/5.5rad, the body angular velocity: 1/1.2rad/s and the output torque: 5Nm.

The learning results will be compared to the single-agent case in which one agent controls both motors, i.e., $\boldsymbol{a} = (\tau_1, \tau_2)^T$.

5.4.3 Results

In order to test the learning performance of META's walking task, we regularly perform a walking run and monitor the number of footsteps the robot made. When 16 footsteps are made, we assume that it can walk stably and we end the trial. Therefore, the maximum performance of a test run is 16 footsteps. During test runs, exploration is disabled. In the next sections, we test various learning algorithms and settings and compare their performance. The learning process is repeated several times with different random initializations of the action-value space. Contrary to the results obtained with the two-link manipulator in Section 5.3, preliminary experiments with Lenient Learning for Meta have not yet led to an improvement of the results.

Single-agent vs. multiagent Q(λ)

In this test we compare the learning result of META's walking task in the single-agent case (SA) with the multiagent case (MA), where all agents use Q(λ). The result can be found in Figure 5.8. First of all, we can conclude that the walking task is successfully learned in both the single-agent Q(λ) and multiagent Q(λ) case. The robot is able to walk after about 15 hours. Second, we can conclude that the difference in learning speed between the SA case and the MA case is negligible. However, in the end, the performance in the multiagent case is slightly less than in the single-agent case.

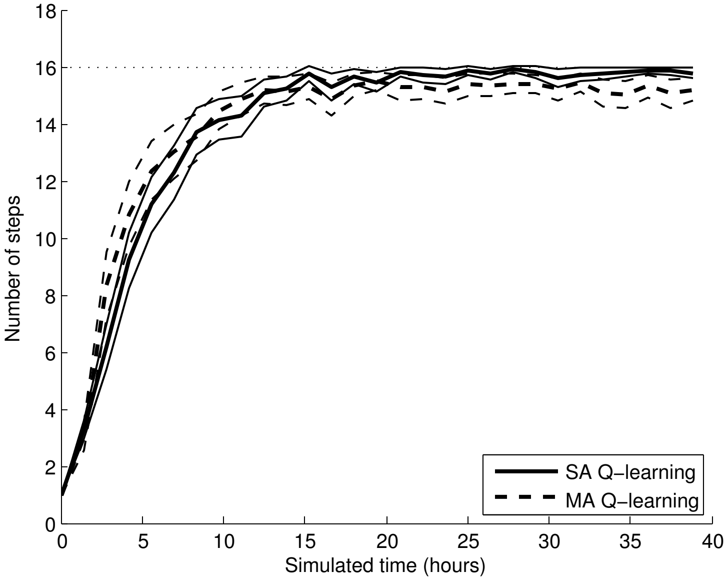


Figure 5.8: *Single-agent (SA) Q(λ) compared with multiagent (MA) Q(λ) for META. The average over 96 independent runs is shown, including the* 95% *confidence bounds on the average.*

Multiagent SARSA(λ) vs. multiagent Q(λ)

Because SARSA is an on-policy learning algorithm, it could be the case that when the agents learn with SARSA, each agent is better able to deal with the changing policy of the other agent. A comparison between the MA case using SARSA(λ) and the MA case using Q(λ) can be found in Figure 5.9. The learning curves do not differ significantly and the SARSA algorithm has no added value in this case.

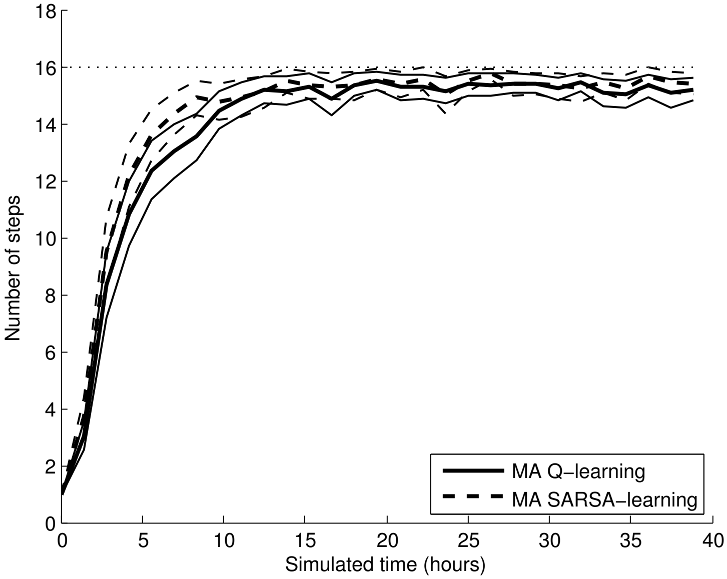


Figure 5.9: *Multiagent (MA) Q(λ) compared with multi-agent (MA) SARSA(λ) for META. The average over 96 independent runs is shown, including the* 95% *confidence bounds on the average.*

Synchronized exploration

In the Independent Learner MA setup, all agents explore independently according to (5.9). However, it could be beneficial to synchronize exploration by letting both agents solely perform explorative actions simultaneously using (5.10). Perhaps the chances of escaping from a local maximum as in the 'climbing game' of Section 5.2.5 increase, when all agents deviate from their current policy simultaneously. We compared the independent exploration strategy with the synchronized exploration strategy. The result can be found in Figure 5.10. The learning graphs do not differ significantly. In this case, synchronized exploration does not increase learning speed or performance.

Reduced state space

Now that we have created a multiagent setting in which each agent controls one actuator, it might be the case that each agent does not need the full state of the robot in order to control its own actuator. Note that we then violate the

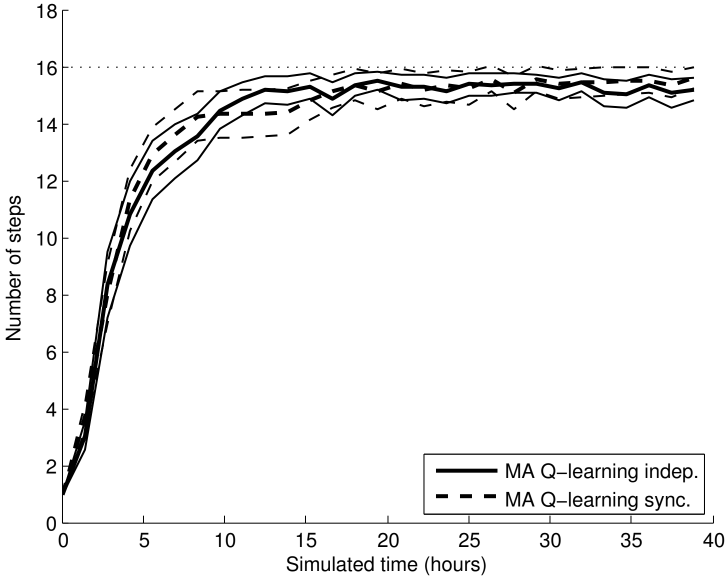


Figure 5.10: *Multiagent (MA) Q(λ) for META, comparing independent exploration with synchronized exploration between the agents. The average over 96 independent runs is shown, including the* 95% *confidence bounds on the average. Synchronized exploration has no significant benefit over independent exploration.*

Markov property not only because all agent-specific transition functions are time-dependent (see Equation (5.3)), but also because we have hidden state variables. In this test, the agent that controls the joint between the upper body and the stance leg does not include the angular velocity of the lower swing leg in its state space. The agent that controls the joint between the upper body and the upper swing leg still uses the full state space of 10 dimensions. The result can be found in Figure 5.11. From this graph, it is clear that the performance of the reduced state space case is far worse than the original MA case, but not failing completely. It is clear that divergence issues occur later on in the graph. Apparently, the incomplete state space is violating the Markov property too much.

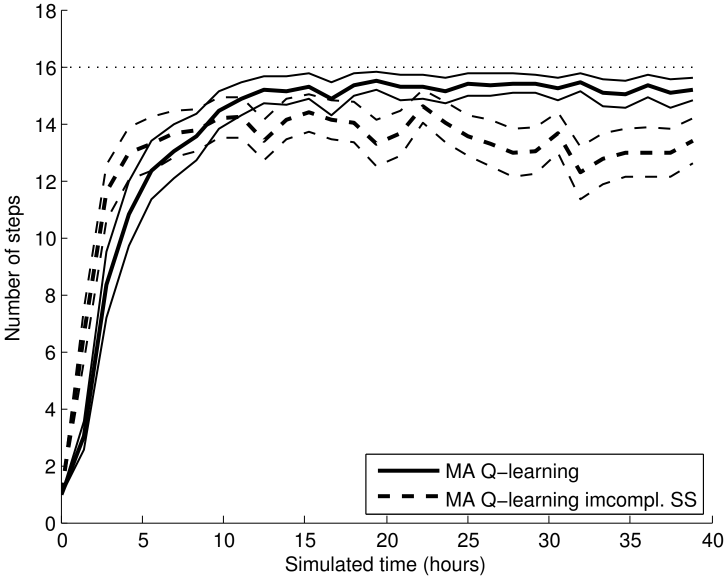


Figure 5.11: *Multiagent (MA) Q(λ) for META. The original MA setting is compared with a MA setting in which one agent has an incomplete state space. The average over 96 independent runs is shown, including the* 95% *confidence bounds on the average. The incomplete state space results in decreased performance and divergence.*

5.5 Leo

The third and last test setup on which we evaluate ILs is the simulation of Leo, a bipedal walking robot, shown in Figure 3.5. Its state space is comparable to that of META. However, Leo has three actuators, whereas the two-link manipulator and META have only two. A detailed description of the simulation model of Leo can be found in Section 3.2.5. In Section 3.3.3, it became clear that with regular SARSA(λ), control delay had a negative effect on the learning process. The control delay is dominated by the action selection calculation; the learning update itself (i.e., changing the feature parameter values) can be performed afterwards, while the action is being executed. Therefore, for this test setup, we focus on the learning performance of MA-SARSA(λ) and the reduction in computation time compared to SARSA(λ); we omit the evaluation of MA-Q(λ), synchronized exploration and lenient learning.

5.5.1 Learning

A detailed description of Leo's task of learning to walk can be found in Section 3.3.3. The single agent learning setup uses SARSA(λ). The multiagent case uses three agents to control Leo: one for the stance hip motor, one for the swing hip motor and one for the swing knee motor. Within each agent, the Q-function is approximated using tile coding with 16 tilings. Generalization is present between states as well as between actions. The action space A^i of each actuator consists of 7 discrete actions. Therefore, according to (5.1), the reduction in action selection computation time is predicted to be a factor $\frac{7^3}{3 \cdot 7} \approx 16.3$.

5.5.2 Results

The learning performance of MA-SARSA(λ) is compared with SARSA(λ) in Figure 5.12, showing the average walked distance per episode of 25s (or until falling) against simulated time. We can observe that MA-SARSA(λ) initially learns slower, but reaches its final system performance approximately as fast as SARSA(λ). With MA-SARSA(λ), the system's final performance is slightly, but significantly lower than when using SARSA(λ).

We compared the computation time that MA-SARSA(λ) and SARSA(λ) require to perform a learning step, as well as the computation time needed to consult the policy. Only the latter contributes to the control delay. The results are shown in Table 5.4. It can be observed that the policy consulting time has been reduced with a factor 12.4. This factor deviates somewhat from the theoretical factor of 16.3 due to overhead in the calculation. Furthermore, the computation time of the total learning step is reduced with a factor 7.4 by the multiagent approach, despite the fact that three agents need to update their feature values instead of one.

We implemented MA-SARSA(λ) on the real robot as well. This reduced the control delay of 10ms using SARSA(λ) to 2.5ms using MA-SARSA(λ), on average; a reduction of 75%. Note that the control delay is not only caused by consulting the policy, but also by other factors, such as the communication with the motors over a serial bus.

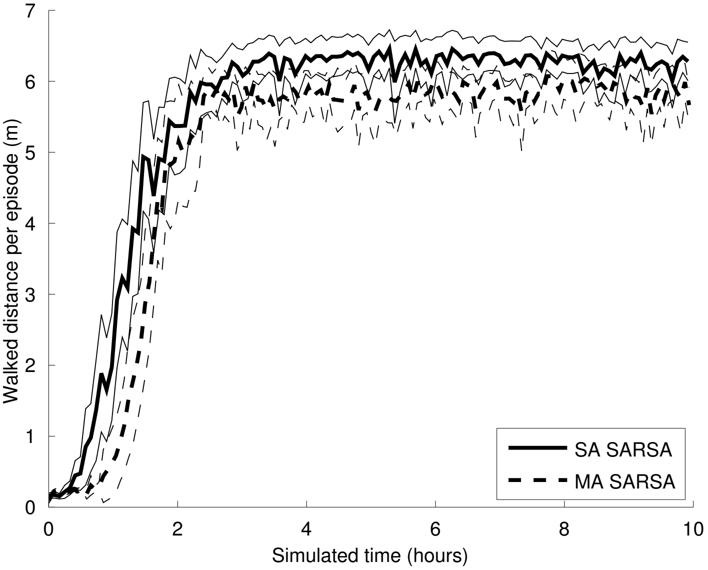


Figure 5.12: *Single-agent (SA) SARSA(λ) compared with multiagent (MA) SARSA(λ) for the simulation of Leo. The average over 24 independent runs is shown, including the 95% confidence bounds on the average. MA-SARSA(λ) learns slightly slower and has slightly lower final performance.*

Table 5.4: *Relative computation time of the total learning step and the policy consulting step of single-agent (SA) SARSA(λ) versus MA-SARSA(λ) for the simulation of Leo.*

	Total learning step	Policy consulting step
SA/MA	7.4	12.4

5.6 Discussion

In this chapter, we used a multiagent approach to make reinforcement learning more scalable for learning complicated robotic tasks. When adding actuators to a learning agent, its action space increases exponentially, requiring a longer learning time, more storage space and more computation time for action selection. The latter can cause control delay on real robots, which can slow down the learning or let it diverge. Decomposing the combined action space of these actuators by splitting it up over different agents solves the last two problems: instead of exponentially increasing in size of the action space, it now becomes linear in the number of actuators. Unfortunately our approach is not proven to converge to the optimal solution. Current theory on convergence, focusing on single state problems, shows that certain situations can cause ILs to converge to a suboptimal solution. We showed however that in our simulations these problems did not result in bad performance.

We implemented Independent Learner algorithms for three simulated robots: a two-link manipulator, and bipedal walking robots META and Leo. The two-link manipulator had to learn to return its two links to the zero position with zero velocity. META and Leo had to learn to walk. Where the two-link manipulator and META had two actuators, and thus two agents in the multiagent approach, Leo had three. In a direct comparison between single-agent and multiagent Q-learning, the performance of cooperative, heterogeneous independently learning agents was not very different from the single-agent case, while memory requirements and action selection computation time decreased with a factor $\frac{m}{2}$ for the two-link manipulator and META, and with a factor $\frac{m^2}{3}$ for Leo, in which m is the number of discrete actions per actuator (equal for all actuators).

With single-agent learning, SARSA(λ) and Q(λ) performed equally well on both the two-link manipulator and META. Because SARSA has an on-policy update rule, we tested the hypothesis that it would perform better than Q-learning with multiagent learning. However, for the two-link manipulator and META, SARSA(λ) and Q(λ) showed similar learning performance.

We proposed an idea to get out of local, suboptimal (Nash) equilibria more quickly by synchronizing the exploration of the agents. However, this did not result in significantly different results for the two-link manipulator and META. This could be because in their state-action value functions there are not many suboptimal (Nash)equilibria.

In recent literature the Lenient Learning algorithm was proposed as a method to learn under the influence of suboptimal actions of other agents in stochastic environments. It was implemented for the two-link manipulator and a significant improvement was found in learning speed and performance. Tuning of the lenience parameters could potentially further increase this improvement. For Meta and Leo, good results with Lenient Learning have not yet been achieved. This subject deserves further attention.

5.7 Conclusions

We successfully employed Independent Learners to control the individual actuators of three robotic setups in simulation: a two-link manipulator, and the two bipedal walking robots META and Leo. In all simulations, learning speed and final system performance using the multiagent approach were comparable to the classical, single-agent approach, while the computational time needed for learning updates and the amount of memory needed to store the state-action space were significantly decreased; from an exponential problem in the number of actuators, it became a linear problem. For robot Leo, the time to complete a learning step was reduced with a factor 7.4, while the policy consulting time was reduced with a factor 12.4. This reduced the control delay on the real robot by 75%, from 10ms to 2.5ms in the current implementation. In addition, in the simulation setup of the two-link manipulator robot, we showed that using Lenient Learning for the independent learners has the potential to significantly increase learning speed compared to single agent learning. The downside of the proposed method is the lack of convergence guarantees; in certain situations, Independent Learners are known to converge to a suboptimal solution. We can conclude that the proposed method is promising and performed well on the three robotic setups tested in this chapter, but more research remains be done on its convergence behavior to make it a reliable method. The method also offers good perspective for future parallel implementations, such as on multi-core processors and robots with distributed computing.

Chapter 6

Reducing system damage using Modular Reinforcement Learning

In Section 3.3, the experiments showed that the learning process was very straining for the hardware. In this chapter, we propose a method based on Modular Reinforcement Learning to reduce the exposure of the robot to risky situations, such as a fall. Initial work on this topic has been done by Van Diepen (2011). The work presented here has been done in cooperation with Wouter Caarls[1].

6.1 Introduction

Reinforcement Learning allows a robotic system to autonomously learn new tasks. With techniques such as temporal difference (TD) learning, this can be done from scratch and model-free. During learning, the system regularly takes suboptimal and sometimes completely random actions due to intentional exploration and as long as it has not reached the optimal solution. A particular challenge for real robots using RL is how to cope with this trial-and-error nature of the learning process, which can be stressful and harmful for the robot's hardware and its environment, especially when learning from scratch. When this explorative behavior causes frequent collisions, for example, this will eventually damage the robot or its environment. Controlling and limiting the strain on the hardware and its environment is an important prerequisite for RL on real robots. Besides the existence of states that are certain to lead to damage – so called failure states or error states – some states and actions might only increase wear and tear or have a *chance* of failure. Therefore, we strive to minimize the *cumulative risk exposure* during learning, where risk, expressed as a scalar negative reward, is a measure for the chance of and the severity of future damage, similar to the definition by Geibel

---

[1] Wouter Caarls is with the Delft Biorobotics Lab, Delft University of Technology, Faculty Mechanical, Maritime and Materials Engineering, The Netherlands.

(2001). Note that this is different from the notion of risk as the variance on the return, where the learning agent runs the risk of not getting a positive return (see, e.g., (Mihatsch and Neuneier, 2002)). We also do not strive to completely avoid risk; we believe taking risk is an inherent component of trial-and-error learning from scratch. We merely want to reach the learning objective with minimum risk of damage. In this work, we focus on the task of learning to walk for a bipedal robot. For our prototype (see Chapter 3), this task has no 'fatal' states, but has many states, such as collisions, that increase wear and tear and will eventually lead to failure.

Traditionally, RL algorithms are judged on how fast the system's behavioral performance (e.g., expressed as the average return or in a physical performance measure) grows against learning time, expressed either in seconds or in numbers of trials. For real robots that can learn without human assistance, however, system damage might be more 'expensive' than pure time, when down time and repair costs are unwanted. From this perspective, we propose to consider cumulative risk exposure as an additional measurement axis for RL on real robots and evaluate RL algorithms based on their behavioral performance against cumulative risk exposure.

In this work, we propose a model-free RL technique based on Modular Reinforcement Learning (MRL), which can reduce cumulative risk exposure without the need for prior knowledge on task solutions. We employ two learning agents, one of which quickly, but imprecisely, learns the expected return of large negative rewards that indicate risk, while another agent learns the return of performance related positive rewards with higher precision, at a slower pace. The expected return of negative rewards is learned faster by using a coarser state space representation for this agent when approximating the action-value function. The goal of this approach is to have the system quickly and coarsely learn to avoid risky states and let it learn the actual task solution in more detail in a safer manner. We test our approach on simulations of two systems: the simplest walker model (see Chapter 4.3.1), and a simulation of bipedal walking robot Leo (see Chapter 3). To empirically prove that our proposed modular approach is effective, we compare it to similar techniques that use a coarser state space representation to learn faster, but which do not make a distinction between positive and negative rewards.

6.2 Related work

The challenge of reducing risk during learning can be approached from several angles. Here, we assume that the robot's hardware design contains as much mechanical and electrical protection as is possible and appropriate for the robot's purpose, and restrict ourselves to solutions from a RL perspective. Furthermore, we restrict the discussion to RL techniques that do not require prior knowledge on the task solution, with the aim to create robots able to learn various tasks without human assistance. This excludes policy restricting methods such as policy gradient RL (Peters, Vijayakumar, and Schaal, 2003; Kohl and Stone, 2004; Tedrake, Zhang, and Seung, 2004), which do provide a convenient way of limiting the solution space to possibly safe and stable solutions, but at the price of requiring prior knowledge on the task solution. Within these boundaries, we identified two apparent approaches to risk minimization; learning faster, and smart exploration.

Learning faster makes the uncertain and explorative initial learning period shorter, which is likely to reduce the cumulative risk exposure. Without meaning to give a complete overview of how to speed up the learning – a general goal in RL research – we give several relevant approaches within our context. Most speedup methods make better use of individual observations, i.e., they increase the sample efficiency, usually at the cost of lengthier computation. More sample efficient methods that do not require an a priori model are eligibility traces (Sutton and Barto, 1998), experience replay (Lin, 1993), the Dyna architecture and prioritized sweeping (Sutton, 1991; Moore and Atkeson, 1993; Sutton et al., 2008), LSPI (Lagoudakis and Parr, 2003) and fitted Q-iteration (Ormoneit and Sen, 2002). A speedup can also be accomplished by using more generalization in the state-action space, i.e., by using state aggregation; information is quickly shared with similar system states and actions. The disadvantage of that approach is that it is likely to reduce the performance of the learned policy due to the loss of resolution in the action-value function[2]. To avoid this, an adaptive-resolution (AR) method could be applied, which starts with a coarse resolution of the action-value function for a quick start, while gradually increasing it as learning progresses for more accuracy. Grzes and Kudenko (2010) apply this in the context of reward shaping (Ng, Harada, and Russell, 1999; Marthi, 2007), where a shaping reward is derived from a value function that is estimated in parallel using aggregated states. They also introduce a mixed resolution case, in which both large and small features are used within a single agent. Both methods increase learning speed, with their combination having the best performance. This approach is closely related to variable resolution discretization (see, e.g., Munos and Moore (2002) and Whiteson (2010)), where states are automatically split in regions where the policy needs improved resolution. Bernstein and Shimkin (2010) use the specific scheme of splitting states based on the number of visits. While all mentioned speedup methods are likely to reduce risk exposure, they leave room for *additional* methods that are specifically designed to reduce risk exposure. Therefore, from here on, we focus on such additional methods and consider it advisable to apply general speedup techniques whenever possible.

The second approach – smart exploration – has the goal to focus exploration on states that are considered safe. In ‘safe exploration’ (Hans et al., 2008) and ’safe RL’ (Garca and Fernndez, 2011), a safety function is either defined (using prior knowledge) or learned, which indicates the safety of an action in a particular

---

[2] When using tile coding function approximation, a high resolution can be combined with large generalization by using many tilings. This, however, has a computational cost, which could lead to intolerable control delay (see Chapter 2).

state. Additionally, a backup policy, either pre-programmed or learned, has the function to return the system to a safe state when the agent is not able to reliably choose a safe action. Although promising, learning both the safety and the backup functions from scratch is expected to be difficult. For inherently unstable robots such as walking robots, many states are risky as long as the agent has not yet learned to keep its balance from these states, making learning the safety function difficult. Learning the backup policy is similar to learning the task of walking itself; it is far from trivial to learn a safe walking gait faster than to learn the optimal gait according to the reward scheme.

6.3 Approach

In this work, we propose a method that specifically aims at reducing cumulative risk exposure, without reducing learning speed or final system performance. This is done by making only mild additional assumptions on the learning problem. The main assumption is that the reward at each time step is composed of several *partial* rewards, some of which provide feedback on the level of success of the robot (usually positive rewards), while others provide feedback on risky behavior (usually large negative rewards). In addition, it is assumed that the agent can deduce from its rewards (either given in composed or decomposed form) whether its state and action are related to risky behavior (e.g., by relating rewards below a certain threshold to risk exposure, such as in (Hans et al., 2008)). This distinction in rewards creates the opportunity for the robot to learn to avoid risky states more quickly – and perhaps less accurately – while learning to accomplish the actual goal(s) of the task with more precision in more time, thereby reducing cumulative system damage. To this end, we apply Modular Reinforcement Learning (MRL) (Uchibe, Asada, and Hosoda, 1996; Sprague and Ballard, 2003; Russell and Zimdars, 2003; Samejima, Doya, and Kawato, 2003; Bhat, Isbell, and Mateas, 2006) – a generic approach with various applications – in the specific way described below.

The common approach in RL is to not make any distinction between rewards for different events, but to learn a single action-value function of the expected total sum of future rewards. In contrast, MRL aims to solve a complex task through a set of learning agents that each optimize a subset of a composed reward signal (i.e., the reward is a superposition of partial rewards), while control actions result from coordination between the agents. Each agent has its own (reduced) state space and (reduced) reward function. Using MRL, we create one agent with a coarse, more generalizing representation of the state space, learning the return of rewards of risk imposing states, and another agent with a finer representation of the state space that learns the return of rewards related to the task's goal(s). The coarser representation of the agent learning risk can result in faster learning, as well as provide an 'early warning' that risk is near, thanks to its broad generalization. All modules share the same action space, and action selection is done based on the sum of the expected return of all agents. This removes the need for negotiation techniques. We compare the proposed MRL method to a simple AR method, as well as to several other multi-resolution approaches that are directly related to the MRL method we propose.

6.4 Theory

The learning process is modeled as an MDP as described in Section 2.1. The goal of the learning agent is to find an optimal policy π^* by estimating the action-value function $Q(s,a)$. The reward function $R : S \times A \times S \rightarrow \mathbb{R}$ is assumed to consist of a superposition of P partial reward functions $R^{(p)} : S \times A \times S \rightarrow \mathbb{R}$, so that the reward r_{k+1} at time $k+1$ becomes

$$r_{k+1} = \sum_{p=1}^{P} R^{(p)}(s_k, a_k, s_{k+1}) = r_{k+1}^{(1)} + r_{k+1}^{(2)} \ldots + r_{k+1}^{(P)} \tag{6.1}$$

Each $R^{(p)}$ returns feedback on a different *aspect* of the learning task. Examples of rewards of different nature for a walking robot are a large negative reward for a risky state such as falling, a positive reward for a (sub-)goal such as a footstep, and a small negative reward for time or energy usage. The action-value function can now be rewritten as

$$\begin{aligned}
Q^{\pi}(s,a) &= \mathrm{E}_{\pi}\left\{ \sum_{i=0}^{\infty} \gamma^i r_{k+i+1} \middle| s_k = s, a_k = a \right\} \\
&= \mathrm{E}_{\pi}\left\{ \sum_{i=0}^{\infty} \gamma^i \sum_{p=1}^{P} r_{k+i+1}^{(p)} \middle| s_k = s, a_k = a \right\} \\
&= \sum_{p=1}^{P} \mathrm{E}_{\pi}\left\{ \sum_{i=0}^{\infty} \gamma^i r_{k+i+1}^{(p)} \middle| s_k = s, a_k = a \right\} \\
&= \sum_{p=1}^{P} Q^{(p),\pi}(s,a)
\end{aligned}$$

in which each $Q^{(p),\pi}(s,a)$ returns the estimated total discounted sum of rewards from partial reward function $R^{(p)}$ under the global policy π. This reformulation allows each partial Q-function to be estimated in a separate way. In principle, one can choose a different state space representation and, if desirable, a different learning algorithm[3] for estimating each $Q^{(p),\pi}(s,a)$.

---

[3] A possible scenario that could benefit from different learning parameters for each $Q^{(p),\pi}(s,a)$ is a scenario in which some rewards are more stochastic than others; a $Q^{(p)}$ that estimates the return of highly stochastic rewards could benefit from a lower learning rate. In this work, however, we only consider deterministic rewards.

We restrict the discussion to linear function approximation for continuous state spaces, in which we use different features for the representation of each $Q^{(p),\pi}(s,a)$ by assigning a vector of basis functions $\boldsymbol{\phi}^{(p)}(s,a)$ and corresponding parameter vector $\boldsymbol{\theta}^{(p)}$ of length $n^{(p)}$ to each $Q^{(p)}(s,a)$ (we drop the dependence of Q on the policy π in the notation and write $Q(s,a)$ instead of $Q^{\pi}(s,a)$ for the sake of clarity):

$$\begin{aligned} \hat{Q}_k^{(p)}(s,a) &= {\boldsymbol{\theta}_k^{(p)}}^T \boldsymbol{\phi}^{(p)}(s,a) = \sum_{i=1}^{n^{(p)}} \theta_k^{(p)i} \phi^{(p)i}(s,a) \\ \hat{Q}_k(s,a) &= \sum_{p=1}^{P} \hat{Q}_k^{(p)}(s,a) = \sum_{p=1}^{P} {\boldsymbol{\theta}_k^{(p)}}^T \boldsymbol{\phi}^{(p)}(s,a) \end{aligned} \tag{6.2}$$

More specifically, the agent could benefit from broader generalization – larger features – in the $Q^{(p)}$-function that estimates the return of large negative rewards coming from risky states, for which fast learning might be more important than accurate learning. Note that in order to obtain the same level of final system performance as in the regular, single Q-function solution, it is logical to use a value for each $n^{(p)}$ comparable to the number of features n chosen in the regular approach. This means that the modular approach is likely to increase the requirements for memory and computation time. Below we discuss a temporal difference (TD) algorithm to estimate the partial Q-functions: MRL-SARSA.

6.4.1 MRL-SARSA

For the widely used SARSA learning algorithm, a modular version is available called 'local Sarsa' (Russell and Zimdars, 2003). We can apply this algorithm, from hereon denoted as MRL-SARSA, to the linear function approximation case by defining a partial TD error $\delta^{(p)}_{\mathrm{TD}_{\mathrm{SARSA}}}$ for each $Q^{(p)}$:

$$\delta^{(p)}_{\mathrm{TD}_{\mathrm{SARSA}},k+1} = r^{(p)}_{k+1} + \gamma \hat{Q}^{(p)}_k(s_{k+1}, a_{k+1}) - \hat{Q}^{(p)}(s_k, a_k) \tag{6.3}$$

Using the partial TD error, every $\boldsymbol{\theta}^{(p)}$ is updated as follows:

$$\boldsymbol{\theta}^{(p)}_{k+1} = \boldsymbol{\theta}^{(p)}_k + \alpha \delta_{\mathrm{TD},k+1} \boldsymbol{\phi}^{(p)}(s_k, a_k) \tag{6.4}$$

When each $\hat{Q}^{(p)}$ uses the same features, i.e., $\boldsymbol{\phi}^{(p)} = \boldsymbol{\phi}^{(q)}$, $\forall p, q \in \{1..P\}$, MRL-SARSA estimates $\hat{Q}$ equivalently to SARSA, albeit stored in partial parameter vectors $\boldsymbol{\theta}^{(p)}$. For full equivalence, the learning rate should be chosen such that $\alpha_{\mathrm{MRL-SARSA}} = \frac{1}{P}\alpha_{\mathrm{SARSA}}$.

Eligibility traces (see, e.g., (Sutton and Barto, 1998)) can be implemented in a straightforward matter, creating MRL-SARSA(λ).

The greedy action, as used in, e.g., the ϵ-greedy action selection scheme, is selected using the *composed* Q-function from (6.2), which is also called "greatest mass" action selection (Sprague and Ballard, 2003):

$$a_{k,\mathrm{greedy}} = \arg\max_{a'} \hat{Q}(s_k, a') \tag{6.5}$$

Equivalently to MRL-SARSA, we can define MRL-Q-learning using the following partial TD error:

$$\delta^{(p)}_{\text{TD}_{\text{Q-learning}},k+1} = r^{(p)}_{k+1} + \gamma \hat{Q}^{(p)}_k(s_{k+1}, a_{k+1,\text{greedy}})) - \hat{Q}^{(p)}_k(s_k, a_k) \tag{6.6}$$

However, we do not further discuss or use this algorithm in the remainder of this chapter. Note that this update rule differs from 'local Q-learning' (Russell and Zimdars, 2003) – which is known not to converge – in the sense that $a_{k+1,\text{greedy}}$ is greedy with respect to $\hat{Q}$, whereas 'local Q-learning' uses the greedy action with respect to $\hat{Q}^{(p)}$ when computing the partial TD error.

Thus far, we assumed that the agent can make a distinction between different 'types' of rewards, which requires prior knowledge on the composition of the reward function. When this knowledge is not available to the agent, we can reduce the general scheme above to a slightly more specific one, in which rewards below a certain threshold r_{thres} are considered to indicate risk. For this type of scenario, we can define two Q-functions of which one, $Q^{(+)}(s,a)$, estimates the return of R^+ that contains rewards above r_{thres}, and another, $Q^{(-)}(s,a)$, estimates the return of R^- that contains rewards below r_{thres}:

$$Q(s,a) = Q^{(+)}(s,a) + Q^{(-)}(s,a) \tag{6.7}$$

using coarse features $\boldsymbol{\phi}^{(c)}$ for $Q^{(-)}(s,a)$ and fine features $\boldsymbol{\phi}^{(f)}$ for $Q^{(+)}(s,a)$. We use this scenario in our experiments in Section 6.5 and 6.6.

6.4.2 Related methods

The approach described above is expected to reduce the cumulative risk exposure during the learning process by making a distinction between rewards related to risk and other rewards. This is implemented using function approximation with features of different size. To verify that making a distinction between rewards is the key element to effectively reduce risk exposure, we compare our method to a series of methods that only capture one element of our MRL method, while using the complete reward function to estimate $Q(s,a)$:

1. Multi-resolution (MR)

Our proposed MRL approach uses features of different sizes. Possibly this is sufficient to reduce risk exposure. We verify this hypothesis by also testing the following method. We use SARSA(λ) to train a single function approximator to estimate $Q(s,a)$, containing the same coarse features $\boldsymbol{\phi}^{(c)}$ that are used in MRL to estimate $Q^{(-)}(s,a)$, as well as the fine features $\boldsymbol{\phi}^{(f)}$ used in MRL to estimate $Q^{(+)}(s,a)$:

$$\hat{Q}_k(s,a) = \boldsymbol{\theta}^{(c)\,T}_k \boldsymbol{\phi}^{(c)}(s,a) + \boldsymbol{\theta}^{(f)\,T}_k \boldsymbol{\phi}^{(f)}(s,a) \tag{6.8}$$

2. Coarse resolution (CR)

The use of larger features introduces broader generalization and faster learning. However, it is likely to reduce the accuracy of the final solution, which may result in worse final behavioral performance. To verify this, we compare our method to SARSA(λ) using a single function approximator to estimate $Q(s,a)$, using only the coarse features $\boldsymbol{\phi}^{(c)}(s,a)$:

$$\hat{Q}_k(s,a) = {\boldsymbol{\theta}_k^{(c)}}^T \boldsymbol{\phi}^{(c)}(s,a) \tag{6.9}$$

3. Adaptive resolution (AR)

In the MRL approach, the beginning of the learning process is dominated by the use of $Q^{(-)}(s,a)$ in action selection, due to mostly random and risky behavior. When the agent becomes more experienced, $Q^{(+)}(s,a)$ becomes dominant. Possibly, starting the learning with broad generalization, while reducing it as learning progresses, is the key effective element in the MRL approach. We verify this with the following alternative adaptive resolution approach. We use two function approximators to estimate $Q(s,a)$ *in parallel*, one using the coarse features $\boldsymbol{\phi}^{(c)}$ and one using the fine features $\boldsymbol{\phi}^{(f)}$; action selection is based on $\hat{Q}_k(s,a)$, which is the weighted sum of the coarse estimate $\hat{Q}_k^{(c)}(s,a)$ and the fine estimate $\hat{Q}_k^{(f)}(s,a)$:

$$\begin{aligned}
\hat{Q}_k^{(c)}(s,a) &= {\boldsymbol{\theta}_k^{(c)}}^T \boldsymbol{\phi}^{(c)}(s,a) \\
\hat{Q}_k^{(f)}(s,a) &= {\boldsymbol{\theta}_k^{(f)}}^T \boldsymbol{\phi}^{(f)}(s,a) \\
\hat{Q}_k(s,a) &= w_k(s,a)\hat{Q}_k^{(f)}(s,a) + (1 - w_k(s,a))\hat{Q}_k^{(c)}(s,a) \qquad (6.10)
\end{aligned}$$

The weight factor $w_k(s,a) \in [0,1]$ grows with the number of times the agent visited that point in state-action space: the relative weight of the smaller features is increased after every visit. For points in state-action space that are newly encountered or infrequently encountered, such as risky states, action selection is predominantly based on $\hat{Q}_k^{(c)}(s,a)$. For frequently visited points, such as the points on the optimal solution, $\hat{Q}_k^{(f)}(s,a)$ becomes dominant in the estimation. Because the state-action space is continuous, we approximate (and thus generalize) the number of visits $n_{\mathrm{v},k}(s,a)$ with a third parameter vector $\boldsymbol{\theta}^{(w)}$ using the fine features:

$$n_{\mathrm{v},k}(s,a) = {\boldsymbol{\theta}_k^{(w)}}^T \boldsymbol{\phi}^{(f)}(s,a) \tag{6.11}$$

We update $\boldsymbol{\theta}_k^{(w)}$ each time step such that $n_{\mathrm{v},k}(s_k,a_k)$ is incremented by 1 after visiting (s_k,a_k):

$$\boldsymbol{\theta}_{k+1}^{(w)} = \boldsymbol{\theta}_k^{(w)} + \frac{1}{||\boldsymbol{\phi}^{(f)}(s_k,a_k)||^2}\boldsymbol{\phi}^{(f)}(s_k,a_k) \tag{6.12}$$

with $\boldsymbol{\theta}_0^{(w)} = \mathbf{0}$. The changing weight factors continually change the approximation and therefore the policy as well, which is potentially detrimental to the convergence speed during this transient period. Therefore, we let $w_k(s,a)$ grow linearly in $n_{\mathrm{v},k}(s,a)$ until $w_k(s,a) = 1$:

$$w_k(s,a) = \max(\frac{n_{\mathrm{v},k}(s,a)}{\beta}, 1) \tag{6.13}$$

with β a visit count threshold. In this way, when all states and actions have been visited β times, $w_k(s,a) = 1, \forall s \in S, \forall a \in A$ and the approximation is completely described by the fine features.

6.5 Test setup 1: The simplest walker

We evaluate the effectiveness of MRL-SARSA(λ) in reducing risk exposure on a simple simulation system - the simplest walker - that learns to walk, and compare against the alternative methods presented in Section 6.4.2. A description of the simplest walker model is given in Section 4.3.1; a description of the MDP and learning parameters can be found in Section 4.3.2.

At the start of a learning episode, the walker is set to a random initial condition that is known to contain enough energy to start walking (but not necessarily leads to a stable walking pattern). This will cause the walker to fall frequently at the beginning of the learning process.

We set the reward threshold that splits R into R^+ and R^- to $r_{\mathrm{thres}} = -2$ so that R^- only contains the negative rewards of -50 for falling, while the time penalty of -1 for every action and the reward of 50 per meter at every footstep are returned by R^+. For now, we assign the same risk level to each fall, no matter the energy of the impact, which makes the number of falls accumulated during learning equivalent to the cumulative risk exposure. A more accurate representation of the amount of risk of each specific fall can make the reduction of risk exposure more effective, but this is outside the scope of this chapter.

6.5.1 MRL-SARSA(λ)

We compared regular SARSA(λ) with our MRL approach. For regular SARSA(λ), we approximated its single Q-function using tile coding function approximation with a feature size that was obtained experimentally and that resulted in satisfactory system performance and learning convergence[4]. This resulted in a feature size of $0.084\mathrm{rad} \times 0.111\mathrm{rad/s} \times 0.105\mathrm{rad} \times 0.222\mathrm{rad/s} \times 0.5\mathrm{rad/s}^2$ in the dimensions θ, $\dot{\theta}$, ϕ, $\dot{\phi}$ and τ. We used 16 tilings. Next, we applied MRL-SARSA(λ) using 16 tilings for approximating $Q^{(+)}(s,a)$ and 16 for approximating $Q^{(-)}(s,a)$. We used the feature size used for regular SARSA(λ) to estimate $Q^{(+)}(s,a)$, while scaling

---

[4] To the best of our knowledge, there exists no method to determine the feature size that gives the best trade-off between learning speed, system performance and memory usage.

the resolution for estimating $Q^{(-)}(s,a)$ with a factor $K_{\mathrm{r}} = 0.5$ in all dimensions (i.e., increasing the feature size). The result can be found in Figure 6.1, showing the average walked distance per episode (ending after 100s or when falling) against the cumulative number of falls that occurred during the learning process required to reach that performance. To discover the sensitivity of our approach with respect to the resolution scaling factor K_{r}, we tested three different values of K_{r} for estimating $Q^{(-)}(s,a)$: $K_{\mathrm{r}} = 0.8$, $K_{\mathrm{r}} = 0.5$ and $K_{\mathrm{r}} = 0.2$. The comparison can be found in Figure 6.2(a).

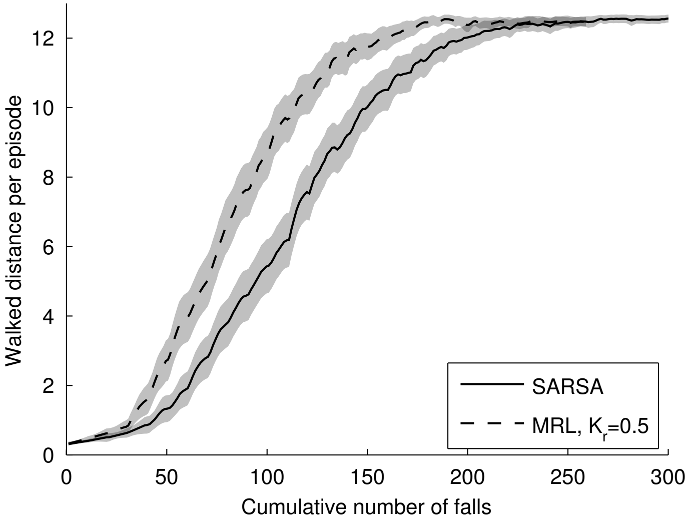


Figure 6.1: *Performance of the simplest walker learning to walk, expressed as the average walked distance in 100s (or until falling) plotted against the cumulative number of falls that occurred during the learning process required to reach that performance. The average of 200 runs is shown, including the 95% confidence bounds of the average. MRL-SARSA(λ) achieves superior system performance over SARSA(λ) after any number of falls.*

We can observe from Figure 6.1 that with MRL-SARSA(λ), the walker reaches a significantly larger average walked distance than with SARSA(λ) for any number of falls accumulated in the learning process, while the final performance is equal. Table 6.1 shows a 26% reduction in the number of falls required to reach $1 - 1/e \approx 63\%$ of the final performance. Figure 6.2(a) shows that using $K_{\mathrm{r}} = 0.2$ (i.e., very large features for estimating risk) resulted in faster learning at the beginning, but also in worse final performance. Using $K_{\mathrm{r}} = 0.8$ had no significant effect. Therefore, when comparing the MRL approach with related methods in the following section, we use $K_{\mathrm{r}} = 0.5$ as the default resolution scaling of coarse approximations.

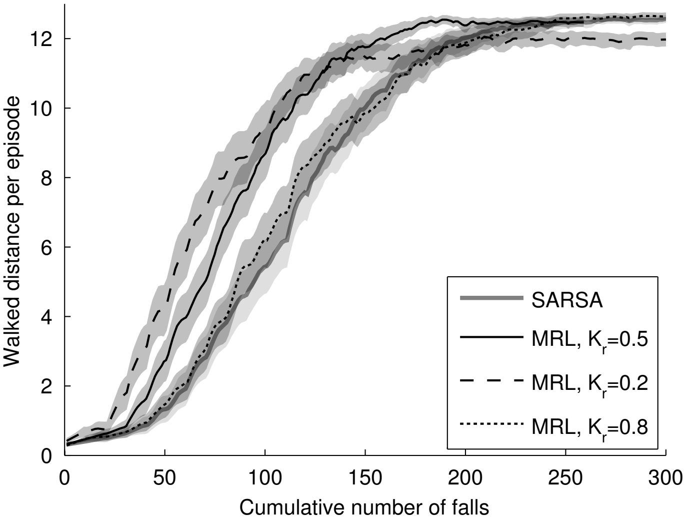
6.5.2 Related methods

We compared our approach to the related methods described in Section 6.4.2. The 'multi-resolution' (MR) method was applied using 32 tilings in total, of which 16 had the full resolution and 16 had a lower resolution ($K_\mathrm{r} = 0.5$). The 'coarse resolution' (CR) method was applied with 16 tilings with the lower resolution. The 'adaptive resolution' (AR) approach was implemented using a weight factor $w_k(s,a) = \max(\frac{n_{\mathrm{v},k}(s,a)}{40}, 1)$, i.e., the approximation is solely composed of fine features for states visited more than 40 times.

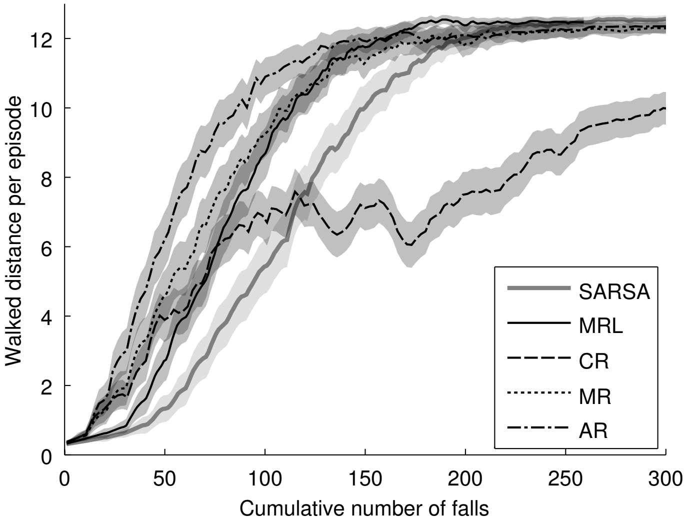
The results are presented in Figure 6.2 and Figure 6.3. We can observe that the hypothesis that merely using a coarser resolution (CR) is sufficient to reduce risk exposure, is false; CR results in poor final system performance. Apparently, the low resolution results in an inferior policy, resulting in high risk exposure (i.e., cumulative number of falls) during learning (Figure 6.2(b)) and many falls per unit time (Figure 6.3(b)). The MR approach performs better than MRL-SARSA(λ) at the beginning, but has slightly lower final performance (Figure 6.3(a)). This manifests itself as a higher fall rate per unit time than MRL-SARSA(λ) (Figure 6.3(b)). Therefore, the hypothesis that using features of different size is key in reducing risk exposure, is partially true; the large features help speed up the learning process in the beginning by also generalizing the expected return of positive rewards, but reduce the final system performance. The AR approach is superior in terms of system performance against cumulative risk exposure (Figure 6.2(b)). In addition, the AR approach results in faster learning (Figure 6.3(a)) than with MRL-SARSA(λ). The hypothesis that starting the learning with a coarser resolution and gradually increasing it is key in reducing risk exposure can therefore be accepted. Of all methods, however, MRL-SARSA(λ) has the lowest number of falls per unit time in the long run. A possible explanation is that the walker walks more cautiously due to the large generalization around risky states, while the estimate of performance related rewards is made using only fine features. This aspect is only present in the MRL approach. In addition, AR needs an appropriate weighting function, which we found much more difficult to tune than the reward threshold parameter used in MRL-SARSA(λ).

Table 6.1: *Cumulative number of falls the simplest walker required to reach* $(1-1/e) \approx 63\%$ *of its final performance.*

	No. of falls	Difference with SARSA(λ)
SARSA(λ)	118.3 ± 3.2	-
MRL-SARSA(λ)	87.9 ± 2.4	-26%
MR	72.2 ± 2.7	-39%
CR	80.1 ± 4.5	-32%
AR	64.5 ± 3.5	-45%

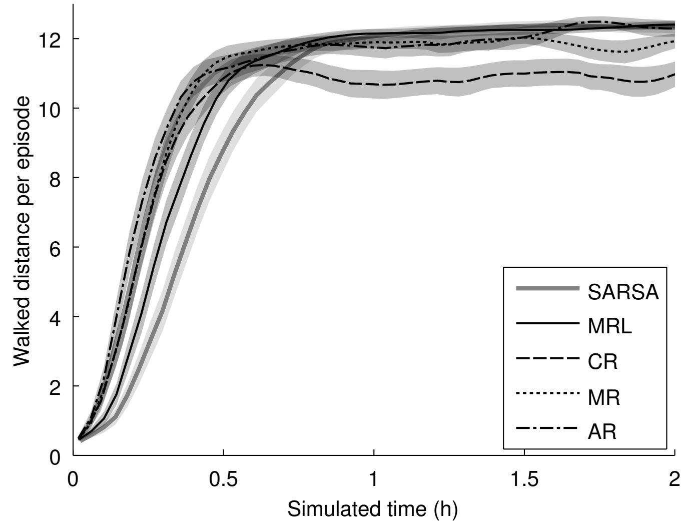


(a) Comparison of SARSA against MRL-SARSA for three values of the resolution scaling factor K_r.

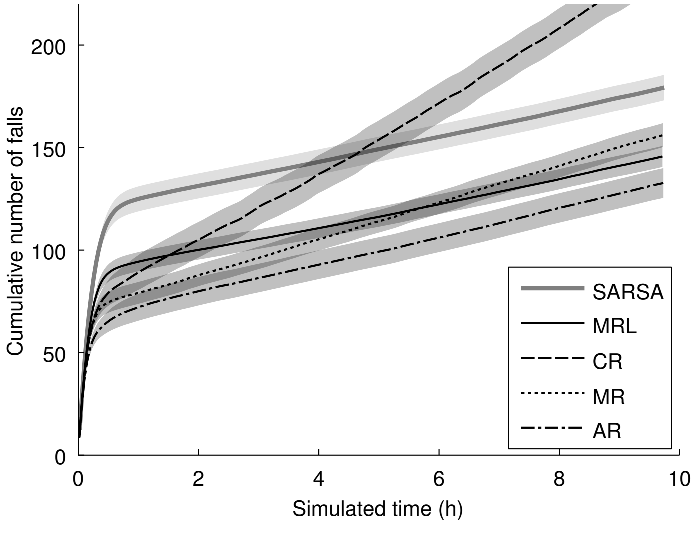


(b) Comparison of SARSA against MRL-SARSA ($K_\mathrm{r} = 0.5$) and SARSA with coarse resolution (CR), multi-resolution (MR) and adaptive resolution (AR).

Figure 6.2: *Performance of the simplest walker learning to walk, comparing regular SARSA(λ) against alternative approaches. The average walked distance in 100s (or until falling) is plotted against the cumulative number of falls. The average of 200 runs is shown, including 95% confidence bounds.*



(a) Comparison of SARSA against MRL-SARSA ($K_{\mathrm{r}} = 0.5$) and SARSA with coarse resolution (CR), multi-resolution (MR) and adaptive resolution (AR).



(b) Comparison of SARSA against MRL-SARSA ($K_{\mathrm{r}} = 0.5$) and SARSA with coarse resolution (CR), multi-resolution (MR) and adaptive resolution (AR).

Figure 6.3: *Performance of the simplest walker learning to walk, comparing regular SARSA(λ) against alternative approaches. The average walked distance in 100s (or until falling) and the cumulative number of falls are plotted against simulated time in (a) and (b), respectively. The average of 200 runs is shown, including 95% confidence bounds.*

6.6 Test setup 2: Leo

In Section 3.3.3, the setup is described of robot Leo learning to walk in simulation using regular SARSA(λ). Here, we compare these results with MRL-SARSA(λ) and the alternative methods from Section 6.4.2. As with the simplest walker, we assign the same risk level to each fall, no matter the energy of the impact, which makes the number of falls accumulated during learning equivalent to the cumulative risk exposure.

6.6.1 MRL-SARSA(λ)

To implement MRL-SARSA(λ) for Leo, we scaled the resolution for $Q^{(-)}$ with $K_{\mathrm{r}} = 0.7$ in all dimensions. For $Q^{(+)}$, we used the resolution of the Q-function of regular SARSA(λ) from Section 3.3.3. From the set of resolution scaling factors $\{0.5, 0.6, 0.7, 0.8\}$, best results were obtained with 0.7. We set the reward threshold that splits R into R^+ and R^- to $r_{\text{thres}} = -50$ so that $Q^{(-)}$ solely estimates the return of the fall penalty and $Q^{(+)}$ estimates the return of all other rewards (i.e., the time and energy penalties and the reward for forward movement).

In initial experiments, MRL-SARSA(λ) did not perform significantly better than regular SARSA(λ) with the MDP setup from Section 3.3.3. Visual inspection of the learning process revealed that the robot spent an unusual amount of time (compared to SARSA(λ)) in a local optimum where it would practically stand still, preferring balancing (i.e., not falling) over making footsteps. The robot seemingly learned to prevent falling quicker than it learned to make footsteps. Since it costs an initial 'investment' in the form of an energy penalty to explore making footsteps by swinging the leg forward, we decided to boost this exploration process by initializing both $\hat{Q}^{(+)}$ and $\hat{Q}^{(-)}$ pessimistically with random values in the range $[-20.01, -20]$. This would remove the disadvantage that already explored state transitions have over unexplored actions when they would lead to walking, but at the cost of an energy penalty. With pessimistic initialization, we obtained the results shown in Figure 6.4, showing the average walked distance per episode (ending after 25s or when falling) against cumulative risk exposure. We can observe that our MRL approach significantly reduces the number of falls over regular SARSA(λ). Table 6.2 shows a 47% reduction in the number of falls required to reach $1 - 1/e \approx 63\%$ of the final performance. The need for pessimistic initialization showed that the simultaneous processes of learning not to fall and learning to make footsteps have to occur at a balanced pace in order not to spend a long time in a local optimum.

6.6.2 Alternative methods

We applied the CR, AR and MR methods from Section 6.4.2 in the following way. The 'multi-resolution' (MR) method was applied with 32 tilings in total, of which 16 had the full resolution and 16 had a lower resolution ($K_{\mathrm{r}} = 0.7$). The 'coarse

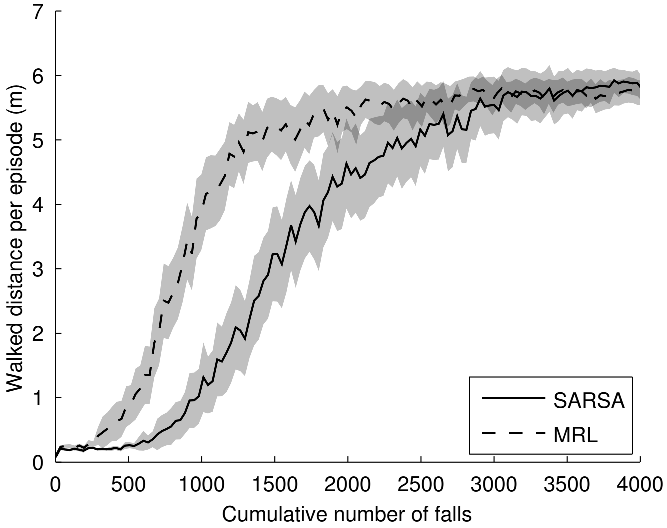


Figure 6.4: *Performance of Leo learning to walk in simulation, expressed as the average walked distance in 25s (or until falling) plotted against the cumulative number of falls that occurred during the learning process required to reach that performance. MRL-SARSA(λ) achieves superior system performance over SARSA(λ) after any number of falls. The average of 48 runs is shown, including 95% confidence bounds of the average.*

Table 6.2: *Cumulative number of falls Leo required to reach* $(1-1/e) \approx 63\%$ *of its final performance.* †*With the AR method, the final performance was not well defined; it did not stabilize, but dropped frequently.*

	No. of falls	Difference with SARSA(λ)
SARSA(λ)	1561 ± 105	-
MRL-SARSA(λ)	834 ± 59	-47%
MR	981 ± 59	-37%
CR	1012 ± 68	-35%
AR	$1076 \pm 105^\dagger$	-31%

resolution' (CR) method was applied using 16 tilings with the lower resolution. The 'adaptive resolution' (AR) approach was implemented using a weight factor $w_k(s,a) = \max(\frac{n_{\mathrm{v},k}(s,a)}{10}, 1)$.

The results can be found in Figure 6.5 and Figure 6.6. We can observe that the CR and MR methods perform comparable to MRL-SARSA(λ) in terms of system performance against cumulative risk exposure (Figure 6.5), while achieving a slightly, but significantly, higher final performance in terms of the average walked distance (Figure 6.6(a)). However, with MRL-SARSA(λ), Leo has the lowest number of falls per unit time in the long run (Figure 6.6(b)) compared to CR and MR. As with the simplest walker, a possible explanation is that the walker walks more cautiously due to the large generalization around risky states. This could also explain why on average, Leo walks slower (less traveled meters per episode) when learning with MRL-SARSA(λ) than with MR or CR. Again, these experiments confirm that merely using coarse features or a combination of coarse and fine features does not perform as well as the MRL approach.

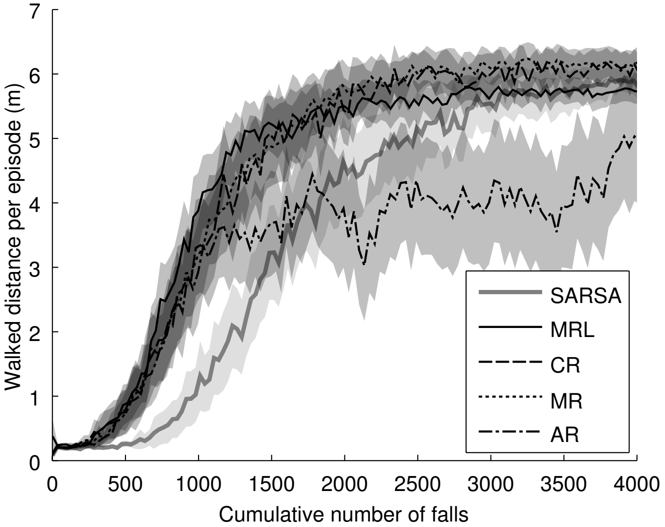
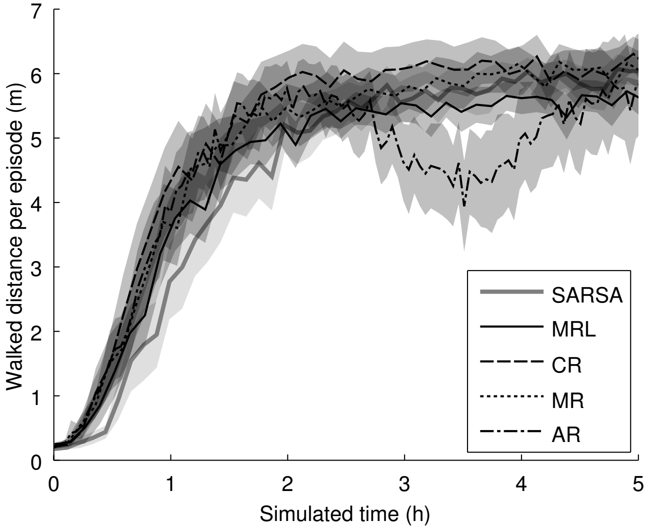
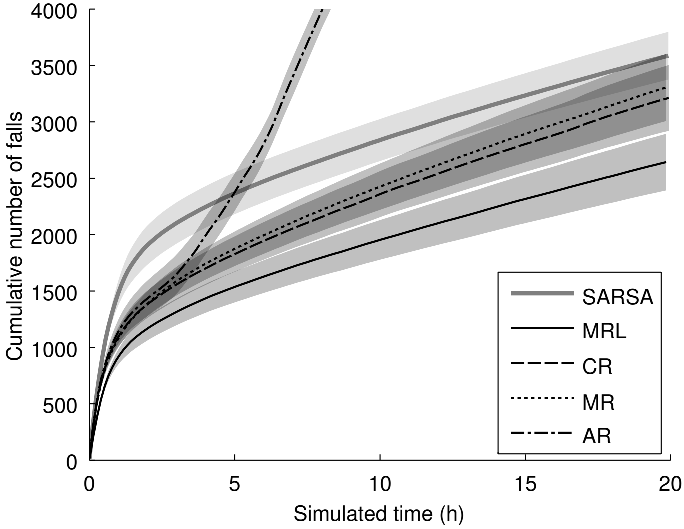


Figure 6.5: *Performance of Leo, comparing regular SARSA against MRL-SARSA ($K_{\mathrm{r}} = 0.7$) and SARSA with coarse resolution (CR), multi-resolution (MR) and adaptive resolution (AR). The average walked distance in 25s (or until falling) is plotted against the cumulative number of falls that occurred during the learning process required to reach that performance. The average of 48 runs is shown, including 95% confidence bounds of the average.*

Using AR, we could not obtain a stable solution within 20h learning, despite several attempts using different weight functions $w_k(s,a)$. Inspection of individual runs revealed that the robot regularly switched to a slightly different gait



(a)



(b)

Figure 6.6: *Performance of Leo, comparing regular SARSA against MRL-SARSA ($K_\mathrm{r} = 0.7$) and SARSA with coarse resolution (CR), multi-resolution (MR) and adaptive resolution (AR). The average walked distance in 25s (or until falling) and the cumulative number of falls are plotted against simulated time in (a) and (b), respectively. The average of 48 runs is shown, including 95% confidence bounds of the average.*

every several hours. This new gait would typically result in higher performance, but only after a period of reduced performance, which not seldom took just as long as the robot needed to learn the initial solution, i.e., in the order of 2h. This resulted in bad average performance (Figure 6.6(a)), also in terms of cumulative risk exposure (Figure 6.5). A typical learning run with AR is shown in Figure 6.7, showing very high peak performance on occasions compared to SARSA(λ). Due to the complex interaction of AR with this learning problem, we cannot conclude whether it can eventually perform equal or better than MRL-SARSA(λ). However, we can conclude that in practice, it is more difficult to apply it successfully. Although MRL-SARSA(λ) needed pessimistic initialization in order to significantly outperform SARSA(λ), it was relatively easy to improve its results and it never underperformed compared to SARSA(λ).

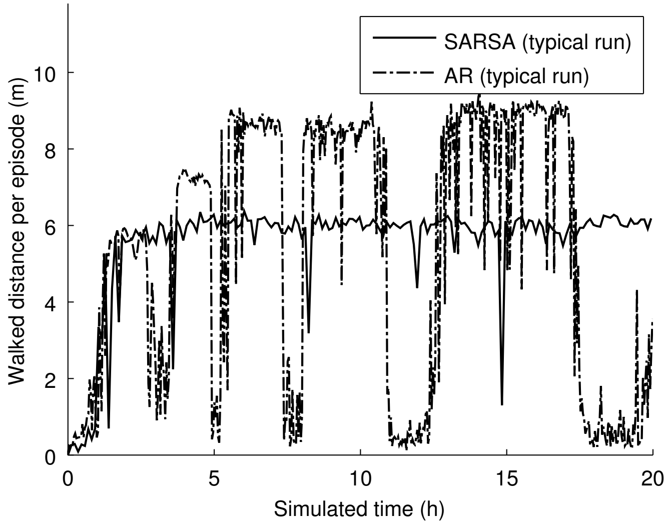


Figure 6.7: *Typical runs of Leo, comparing regular SARSA(λ) against SARSA(λ) with adaptive resolution (AR), illustrating the time varying performance of AR.*

6.7 Conclusion

In this chapter, we introduced a new model-free approach to learning from scratch with the aim to reduce the risk exposure that is accumulated during the learning process. The approach assumes that the reward function is composed of a set of partial rewards, of which the large, negative rewards give feedback on the risk of state transitions. Under this assumption, it is possible to learn the expected return of risk exposure separately from the expected return of performance related rewards. By using a coarser resolution in the function approximation of the risk related action-value function, the agent quickly and imprecisely learns to avoid risky behavior, while a finer representation is used to more accurately learn the action-value function of positive rewards. We implemented this idea using MRL-SARSA(λ). For two simulations of walking bipeds – the simplest walker model and a model of robot Leo – that learned to walk while avoiding the risky event of falling, this approach resulted in a higher behavioral performance against cumulative risk exposure. Using MRL-SARSA(λ), the simplest walker required 26% fewer falls and Leo required 47% less falls over SARSA(λ) to reach 63% of the final system performance. We compared our approach to three related, alternative methods: a coarse resolution method, a multi-resolution method and an adaptive resolution method. From these experiments, we can conclude that merely using a coarse resolution is likely to lead to an inferior policy, leading to increased cumulative risk exposure. The multi-resolution method, that combined a coarse and fine resolution, showed an improvement over merely using a coarse resolution. However, compared to MRL-SARSA(λ), it still suffered from increased risk exposure in the long run. The adaptive resolution method weighed a coarse and fine approximation, starting with the coarse approximation and assigning more weight to the fine approximation in more frequently visited states. This proved to be more effective than MRL-SARSA(λ) for the simplest walker, but could not be successfully applied to the simulation of Leo; it proved more difficult to set the key parameter value for AR than for MRL-SARSA(λ). In summary, we can conclude that our MRL approach is an effective, model-free, easy to apply method to reduce cumulative risk exposure for walking robots.

Chapter 7

Discussion, conclusions and future directions

7.1 Research goal

The market for service robots is expected to increase dramatically in the coming years. Because service robots need to operate in largely unstructured, highly diverse and renewing environments, it is difficult to provide them at production time with manually programmed controllers that enable them to perform a large variety of tasks in various environments. Having robots *learn* motor control tasks autonomously forms an attractive alternative to manual programming by experts. Reinforcement Learning (RL) has the potential to allow robots to learn motor control tasks autonomously from interaction with the environment in a largely unsupervised way, without the need for a model or an initial solution, and is therefore a promising paradigm. Despite the fact that RL has been successfully applied to a wide set of problems, the number of successful demonstrations of real robots using RL to learn motor control tasks in real-time on embedded hardware is only in the order of 10 worldwide. Currently, both theoretical and practical difficulties are impeding wide scale application of RL to real robots. Therefore, the research goal of this thesis was to identify and address difficulties in hardware design, software design and RL theory that currently prevent the application of RL to real, autonomous service robots. Below, we present the conclusions of this thesis in reply to the research questions posed in Chapter 1.

7.2 Discussion and conclusions

7.2.1 RL techniques

In Chapter 2, we presented the main RL techniques that have been used throughout this thesis in an answer to research question 1:

> *"What are suitable RL techniques for real-time, autonomous learning of low-level motor control tasks on a real robot without the need for prior knowledge on the task or its environment?"*

The Markov Decision Process (MDP) is the common framework within RL to describe learning problems. To solve MDPs, we chose Temporal Difference (TD) learning for the following reasons: it does not require a model of the system and its environment, nor does it need an initial solution from an expert; it has been successfully applied in simulation to learn robotic tasks; algorithms with low computational complexity are available, such as SARSA(λ) and Q(λ); and, under the right conditions, it converges to the globally optimal solution. To learn in continuous state spaces, we chose tile coding, a linear function approximation technique with low computational complexity – also in high dimensional state spaces – that showed important successes in the past. In addition, techniques have been discussed that reduce the computational requirements of RL in large state-action spaces, such as exploiting symmetry and using hashing for the storage of the function approximation's data.

We showed in simulation that time delay in the control loop – not included in the standard MDP framework – can have a strong negative influence on the convergence of TD learning. We proposed the new memoryless TD algorithm dSARSA(λ) that can perform better than regular SARSA(λ) while maintaining low computational complexity. If a system is accurately linearizable at the time scale of a single state transition, the method can also be applied when the control delay is not an integer multiple of the sampling period.

Throughout the thesis, the techniques introduced in Chapter 2 have proved to perform satisfactorily in learning high dimensional, realistic robotic tasks such as bipedal locomotion, requiring little task-specific knowledge and being able to run in real-time on embedded robot hardware. One particular boundary condition can be considered key in this success: the computational power of nowadays' computer hardware. Despite the fact that the techniques used are not very novel, they simply could not have been employed in real-time with acceptable control delay without modern computer hardware (and an efficient software implementation). Perhaps this partly explains why these techniques have thus far been unpopular in real robot applications.

7.2.2 Hardware and software requirements

In Chapter 3, we first addressed research question 2:

> *"What are the hardware and software requirements for a real robot in order to be suitable for these RL techniques?"*

We derived hardware and software requirements from the MDP framework in the context of a walking robot with the goal to create a suitable research platform for RL. The following hardware requirements have been derived:

1. The robot can walk over a period of days and is robust against falls and self-collisions.
2. The robot can observe state s, which holds all information relevant to the learning problem
3. The effect of action a in every state s is predictable.
4. The sampling period is constant.
5. The state transition probability density function T must be stationary within a time frame of tens of hours.
6. The robot's number of degrees of freedom, and thereby its state-action space, is limited such that learning succeeds within a reasonable time frame.

With real hardware, most of these requirements can only be met approximately. The following software requirements have been derived:

1. A realistic simulation of the robot and its environment can easily be created and modified.
2. Controller code works in simulation as well as on the real robot without additional modifications.
3. The software architecture facilitates the incorporation of a self-diagnostics module that monitors the system dynamics.
4. The system is real-time (a direct consequence of the requirement of a constant sampling period).
5. The delay between measurement s_k and control action a_k is minimal and measurable (this is a hardware requirement as well)
6. The robot's RL problem can be easily defined and modified without recompilation.

With prototype 'Leo', we presented a bipedal walking robot specifically designed according to the aforementioned requirements for online, autonomous Reinforcement Learning. Leo is small and light (approximately 50cm in height and 1.7kg) and has 7 servo motors: two in the ankles, knees and hips and one in its shoulder. The servo motors measure their position and temperature, and communicate over serial ports with an embedded computer (VIA Eden 1.2GHz CPU and 1GB RAM). Force sensors in the toes and heels detect foot contact. Sideways stability is enforced by a boom construction, which makes it effectively a 2D robot. Leo has foam bumpers in crucial places to protect it against falls in a wide range of configurations. Due to its boom construction which makes it run in circles and supplies power, its fall protection and its ability to stand up by itself, Leo can perform RL experiments without human assistance.

7.2.3 Identification of practical complications

Through experimentation, we addressed research question 3:

> *"What are the practical complications that arise from applying these RL techniques to a real robot?"*

Using a 'conventional', pre-programmed controller, we checked the robustness of the system by letting it walk for 8 hours, during which it made $43,000$ footsteps and fell 30 times before failing. Although this is an amount of effort comparable to that needed for a learning experiment (albeit with less falls), it was desirable to further improve the robustness. To this end, we replaced the position recording potentiometers in the actuators with contactless magnetic encoders and added torsionally flexible coupling elements to the joints to reduce gearbox damage. We made the actuation more predictable by compensating for the temperature of each motor. We verified the invariability of the system over a period of 8 hours by periodically building a model from measured state transitions, and validating this model with measurements recorded later. This showed that the system is not completely time invariant, which was mostly caused by a deteriorating position recording potentiometer in one of the actuators. We addressed this issue by replacing the potentiometers with magnetic encoders.

We measured the timing characteristics of our real-time control loop and found that the control delay between measuring the state and actuating the motors is significant. Since we found in Chapter 2 that control delay can negatively influence learning performance, we included control delay in simulations of 'Leo'.

To support the research on on-line RL on real robots, we created a generic motion control software framework with tight integration of a simulation environment. The developed simulation environment allows safe and realistic evaluation of controllers and is highly configurable via XML in terms of the robot's dynamics model, its environment and the learning controller. Controller code – newly created or based on the provided real-time implementations of temporal difference learning algorithms – can be shared one-on-one between the simulation and the real robot. Using a publish/subscribe architecture, state information is distributed to the controller and additional modules such as logging and visualization services. The system is real-time periodic when run under Linux with the Xenomai extension, with both minimal and measurable control delay.

We performed several RL experiments on our prototype with increasing difficulty. The first RL experiment on the prototype consisted of a relatively simple learning task that involved only one leg (the other body parts were statically mounted) – the stairs step-up task – and showed that the robot was able to learn from scratch to place its foot on a plateau in approximately 15 minutes. The results obtained in simulation and in hardware did not differ significantly. This showed that the prototype's hardware and software were suitable for RL experiments of short duration. Subsequently, we defined the MDP of learning to walk – a task with 10 state dimensions and 3 action dimensions for our prototype – and presented simulation results. In simulation, the robot learned to walk from scratch in 3 hours or less. We showed that both control delay and floor height differences increased the learning time (up to a factor of two), as well as the number of falls occurring during the learning process. To reduce the latter, we adjusted the robot setup by leveling the floor. The simulation results also showed that the prototype would have to withstand thousands of falls before learning to walk from scratch. Since our prototype was not robust enough for that, we applied a method to speed up the initial learning period in which the robot falls frequently, while still being able to study the presented RL techniques of learning to walk on the current prototype. During an initial period of a few minutes, the prototype was controlled by a pre-programmed controller. For this demonstrated solution, the action-value function was estimated on-line, which then served as an initialization for the remainder of the learning process. After successfully testing this method in simulation, we executed the method on the prototype for 4.5 hours, in which it learned to perform at least as well as the demonstrated pre-programmed solution. To the best of our knowledge, this is the first demonstration of Temporal Difference learning in real-time, on embedded robot hardware involving a non-trivial task and a high-dimensional state-action space.

In summary, we can conclude that the main practical complications of on-line TD learning on a walking robot are the robustness and stationarity of the hardware with respect to the explorative nature of RL, control delay due to the computational complexity of the learning algorithm, and disturbances of various kinds, such as sensor disturbances and unmeasured irregularities in the environment. Despite these difficulties, it was shown that it is possible to employ autonomous TD learning in real-time and on a real robot to learn an non-trivial task.

While our prototype was able to meet (after various improvements) the majority of the posed hardware and software requirements, in the end, its transmission proved to be inadequate: gearboxes would break frequently. When taking into account that the motors would typically heat up to their maximum allowable temperature within 30 minutes, one could conclude that the combination of motor and gearbox was simply undersized. However, merely placing a heavier duty motor and gearbox in the current design would not necessarily solve the issues: the robot's mass would increase, leading to larger impacts on all robot parts. Though not trivial, a redesign of the prototype – in which other types of actuation can be considered as well, such as direct drive and harmonic drive mechanisms – is expected to increase the robustness of the hardware to an acceptable level.

It is undeniable that the trial-and-error nature of TD learning draws heavily on the hardware. The strength of TD learning – convergence to the global optimum without having to restrict the policy space to a task-specific class of solutions – is also its weakness: random actuation patterns and exotic system states put a heavy load on the robot's hardware. It remains a challenge how to reduce the likelihood of experiencing harmful actuation patterns and system states, without jeopardizing the convergence properties. The method presented in Chapter 6 sets a step in this direction.

7.2.4 Reducing practical complications

In Chapters 4, 5 and 6 we addressed research question 4:

> *"How can these practical complications be addressed?"*

From the experiments in Chapter 3, it became clear that disturbances, for example floor height variations, can have a negative impact on the learning performance. In Chapter 4, we researched the impact of large and infrequent disturbances – or outliers – on the learning process. Stochastic system behavior is part of the stochastic MDP framework and poses no problem for most learning algorithms, other than that it usually results in the need to average over more experience (i.e., using a lower learning rate) and thus longer learning times. However, the effect of large and infrequent disturbances is relatively unknown. Every real system will suffer from outliers to some degree. They can occur in sensor readings, timing or in unexpected interactions with the environment. We evaluated the effects of outliers on a simple simulation model of a walking robot, which learned to walk using SARSA(λ). We tested the effects of three types of outliers: an instantaneous push, a sensor reading outlier, and a sampling time irregularity.

Pushing the walker at random moments, on average once in approx. 6 footsteps, had a dramatic effect on the learning time and system performance. Rejecting the outliers by excluding the faulty state transitions from the learning process completely restored the performance of the walker. After an equal number of practicing hours, the 'ignorant' walker that included the outliers in the learning updates performed roughly half as well as the outlier rejecting walker. This contrasts with the possible expectation that learning under the influence of disturbances would produce a more robust policy; apparently, the size and frequency of the disturbances in this experiment did not allow for them to be treated as stochastic variations of the underlying MDP.

The introduction of random spike noise on the sensor reading of the hip angle, on average once every 50 measurements, had an undetectable effect on the learning agent. When their frequency was increased ten times (unrealistic), outlier rejection actually resulted in a *decrease* in learning speed. This can be explained by the fact that we excluded outliers in SARSA(λ) by clearing the eligibility traces, thus on average once in 5 samples, which slowed down learning. Doubling the sampling time randomly, on average every 50th sample, also had an undetectable effect on the learning agent. When increasing their frequency ten times (unrealistic), the effect became noticeable but was still surprisingly small. Again, rejecting outliers by clearing the eligibility traces led to a large drop in learning speed. The rejection process had a much more negative impact on the learning performance than the outliers themselves.

We can conclude that for the simple model used, large disturbances of the actual system state through unexpected interaction with the environment have by far the largest influence on the learning process, compared to timing and sensor outliers.

Chapter 2 and 3 showed that control delay can slow down the learning process and at worst make it diverge. In Chapter 5, we adopted a multiagent RL approach in which each actuator of the robot is controlled by an independent, separately learning agent. Because each agent has a smaller, one dimensional action space, consulting the policy requires much less computation, thereby reducing the control delay on the real robot. In addition, memory requirements are reduced. The difficulty introduced is the lack of coordination between the agents. However, due to the cooperative nature of the multiagent system, the system is still capable of finding (possibly suboptimal) solutions. We tested the approach on three robotic systems in simulation: a two-link manipulator, and bipedal walking robots Meta and Leo. While the learning performance proved to be similar to the single-agent case, the computational time needed to complete learning and the amount of memory needed to store the state-action space were significantly decreased; from an exponential problem in the number of actuators, it became a linear problem. For robot Leo, the shorter computation time needed for action selection resulted in a reduction of the control delay with 75% on the prototype. In addition, in one of the test setups, we showed that using Lenient Learning for the independent learners has the potential to significantly increase learning speed compared to single agent learning. The downside of the method is that in its current form, convergence is not guaranteed and some situations are known to lead to suboptimal solutions. Altogether, the multiagent approach is a promoising alternative to single-agent learning, and deserves further theoretical attention. It has perspective for future parallel implementations, such as on multi-core processors and robots with distributed computing, for example, in the form of actuators equipped with their own computing hardware.

The frequent hardware failures were the largest impediment in the experiments with prototype Leo (see Chapter 3). While hardware improvements form an important solution to increasing the robustness of robots using RL, we researched in Chapter 6 how a different formulation of the RL problem can contribute to reducing system damage. We introduced a new model-free approach to learning from scratch, based on Modular Reinforcement Learning (MRL), with the aim to reduce the risk exposure that is accumulated during the learning process. The approach assumes that the reward function is composed of a set of partial rewards, of which the large, negative rewards give feedback on the risk of state transitions. Under this assumption, it is possible to learn the expected return of risk exposure separately from the expected return of performance related rewards. By using a coarser resolution in the function approximation of the risk related action-value function, the agent quickly and imprecisely learns to avoid risky behavior, while a finer representation is used to more accurately learn the action-value function of positive rewards. We implemented this idea using MRL-SARSA(λ). For two simulations of walking bipeds – the simplest walker model and a model of robot Leo – that learned to walk while avoiding the risky event of falling, this approach resulted in a higher behavioral performance against cumulative risk exposure. Using MRL-SARSA(λ), the simplest walker required 26% less falls and Leo required 47% less falls over SARSA(λ) to reach 63% of the final system performance. We compared our approach to three related, alternative methods: a coarse resolution method, a multi-resolution method and an adaptive resolution method. From these experiments, we can conclude that merely using a coarse resolution is likely to lead to an inferior policy, leading to increased cumulative risk exposure. The multi-resolution method, that combined a coarse and fine resolution, showed an improvement over merely using a coarse resolution. However, compared to MRL-SARSA(λ), it still suffered from increased risk exposure in the long run. The adaptive resolution method weighed a coarse and fine approximation, starting with the coarse approximation and assigning more weight to the fine approximation in more frequently visited states. This proved to be more effective than MRL-SARSA(λ) for the simplest walker, but could not be successfully applied to the simulation of Leo; it proved more difficult to set the key parameter value for AR than for MRL-SARSA(λ). In summary, we can conclude that our MRL approach is an effective, model-free, easy to apply method to reduce cumulative risk exposure for walking robots.

7.3 Future directions

7.3.1 Autonomously learning from scratch

In this thesis, we chose the approach of learning a task completely from scratch, without the need for an initial solution from an expert or prior knowledge on the specific task at hand. While this approach has practical disadvantages – the most important one being the often long initial learning period in which behavior is mostly random and particularly straining for the hardware – the ultimate result of this approach is very powerful: machine learning techniques that do not have a dependency on the specific robotic system, nor on the task that it needs to perform. The techniques used throughout this thesis showed that this is possible to a large extent, but configuring the learning system – choosing the state-action space, function approximation resolution, learning parameters and rewards – still requires expert knowledge, especially when learning time and final system performance need to meet specific values. Below, we discuss research directions as to further reduce the need for expert knowledge.

While the state variables of the robot itself are known beforehand, state variables belonging to the specific task at hand, such as the location and orientation of objects that the robot needs to interact with, need to be manually specified with the RL techniques presented in this thesis. Automatically discovering these additional state variables forms a large challenge and involves automatically finding a state-action space in which both state transitions and rewards become predictable. State abstraction from raw sensor data, such as visual input from a camera sensor, is part of this challenge.

Once the state-action space of a learning task is known, the location and shape of the basis functions for approximating the action-value function need to be chosen. A large body of literature exists on methods that automatically choose these parameters, mostly based on heuristics, such as reducing the error or variance of the estimated function values by moving, splitting or aggregating basis functions. To the best of our knowledge, no method exists yet that can consistently produce satisfactory results for a wide range of learning problems. This is unfortunate, since choosing the size and location of the basis functions (the tile widths when using tile coding) formed a large – if not the largest – part of the effort spent on choosing learning parameters for the experiments in this thesis. Therefore, a break-through in this field would greatly speed up RL research.

Choosing the learning rate and exploration rate proved not to be difficult; approximately the same values were used for all experiments. Choosing the time discounting factor and eligibility trace discounting factor also proved not difficult, provided that the characteristic time scale of the learning problem was known. Finding methods to automatically discovering the characteristic time scale of a task can therefore help automatically choosing these learning parameters. Choosing the sampling period is related to choosing the action space and was more difficult. Although some initial guidelines were provided in this thesis for choosing sensible values, further research is needed to automate this process.

The reward function determines what is being learned. In addition, different reward functions can lead to (approximately) the same solution, while resulting in a greatly differing learning speed. In the envisioned environment of service robots, it is likely that rewards are generated both by the robot itself, such as penalties for energy usage and time, as well as by the environment in the form of the robot's supervisor (or trainer, or teacher), giving feedback on the robot's behavior. While some research is targeted towards engineering the reward function in detail with the goal to increase the learning speed, ultimately, one of the biggest advantages of RL is that it has the potential to learn from coarse feedback from its environment *without* requiring engineering or programming skills to specify the rewards in detail. Efforts into making RL more robust against variations in the rewards are therefore considered more valuable than 'optimizing' the reward function. From that point of view, a weakness in the current formulation is that the reward function has to be stationary and consistent; in other words, if the robot once received a reward for good behavior, it needs to receive the same reward *every time* it exercises that same behavior – as long as it is learning, even when its behavior is already optimal. This requirement could be relaxed by letting the robot estimate the reward function that its trainer is trying to communicate, interpreting feedback from the trainer as corrections only, i.e., once behavior is valued by the trainer, this value remains valid until additional feedback is provided. Interpreting feedback from humans – through natural or body language – forms another challenge in this respect.

7.3.2 Algorithmic improvements

The work in this thesis concentrated on Temporal Difference learning and used algorithms, such as SARSA(λ) and Q(λ), that have existed for almost two decades. It should be noted that new RL algorithms are continuously being developed. While in general, these developments do not address the research questions posed in this thesis, naturally, they are responsible for important progress in the RL research field, e.g., by algorithms with improved convergence properties, especially with respect to function approximation and off-policy learning. In particular, new actor-critic and policy gradient methods have become increasingly popular in recent years. Although we excluded these algorithms from our analysis due to their need for an initial solution upon which they improve, they form a welcome addition to methods designed to learn from scratch. Their advantages, such as the smooth way in which they improve the policy, are desirable in situations where an initial solution is available, e.g., through imitation learning, or when learning to improve a rough solution that was obtained by learning from scratch. In that sense, algorithmic improvements form a research trajectory parallel to the more practical one addressed in this thesis.

A research direction that has not yet received extensive attention, but might become increasingly relevant, is to develop algorithms that can benefit from parallel computation. The general trend in computer hardware development is an increase in the number of parallel computing elements (or cores). To be able to benefit from this increase in computing power, more research is needed to adapt current RL algorithms or to develop new ones.

The algorithms employed in this thesis have a particular weakness that is addressed in newer algorithms: they are not very sample efficient, i.e., the improvement in the action-value function estimate per sample (i.e., a state transition and accompanying reward) is limited. Several new algorithms, such as Least Squares Temporal Difference (LSTD) learning, are able to achieve a higher sample efficiency. A promising research direction that improves the sample efficiency – apart from developing new algorithms that directly learn the policy – is model learning. When models of the state transition probability density function and the reward function are learned using real experience, a background process (preferably executed in parallel, to make use of parallel computing hardware) can use these models to simulate experience and perform additional learning updates, which can increase the learning speed. Learning a state transition model has several other valuable applications. In Section 3.1.7, we used Local Linear Regression (LLR) to learn a state transition model and employed it to verify hardware stationarity. A state transition model can also be used to detect disturbances with the aim to exclude them from the learning process, as mentioned in Chapter 4. Finally, a state transition model can be used to predict future states, which can improve action selection under control delay. Therefore, model learning techniques are a particularly interesting future research direction to address the practical complications discussed in this thesis. Recently, new actor-critic algorithms have been introduced in which the actor, the critic, the state transition model, and a reference model are all approximated using LLR (Grondman et al., 2012).

While this thesis showed that under the right conditions, RL can solve tasks in high dimensional state-action spaces, ultimately, it becomes inefficient to learn every complex task from scratch. Many tasks share common skills, such as navigating to a new location, or moving certain robot extremities to a desired location in space. Several approaches exist in literature that extend the RL framework to one in which the action space of a learning task consists of commanding other, simpler tasks. One of those approaches is Hierarchical Reinforcement Learning (HRL). In addition, methods exist to automatically detect such task compositions and hierarchies automatically from experience. In (Van Vliet et al., 2011), we adopted a hierarchical learning approach to the stairs step-up task presented in Section 3.3 and presented results obtained on prototype Leo.

Despite the large body of RL research, RL on real robots is still in its infancy, and the learning capabilities that have thus far been realized cannot compete with those of humans. But watching a humanoid robot learn its first footsteps without human intervention is truly exciting. Its steady progress with the occasional falls looks exemplary for the long but promising research path that lies ahead.

Appendix A

Leveling the floor

Floor height differences, when left unmodeled, violate the Markov property and can slow down the learning process (see Section 3.3.3). Therefore, the 4m by 4m floor that Leo walks on was leveled. The height of the floor with respect to a laser level was measured at 441 equidistant grid points, 20cm apart. To correct for the systematic error of the laser level device, several points were measured twice, making a total of 496 measurements. The floor was then leveled with 5 layers of 3mm foam sheet, topped with a laminate floor. The illustrations in Figure A.1-A.6 describe the process step by step.

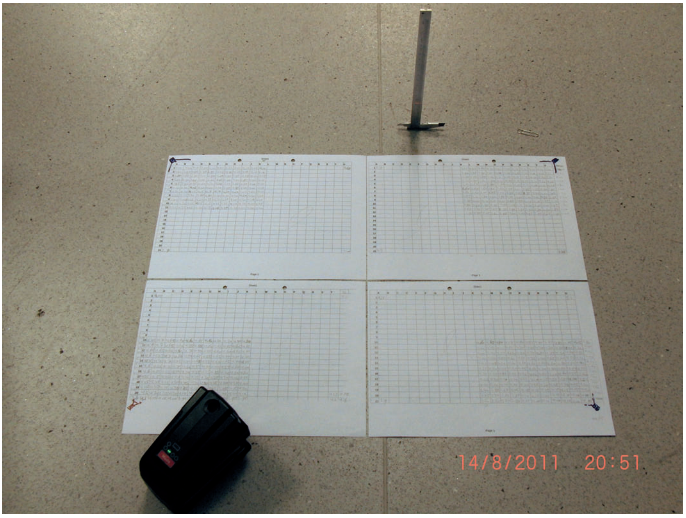
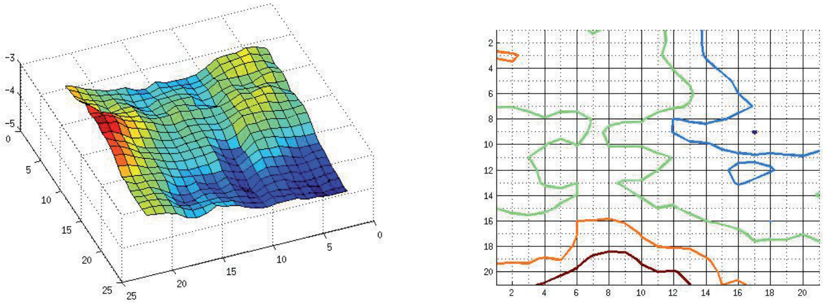


Figure A.1: *Using a laser level and a ruler (caliper), the relative height of the floor was measured at* 441 *equidistant points on a 4m by 4m area. From each corner,* 121 *points close to the laser level were measured.*



(a) Height map (b) Contour plot with isolines at 3mm steps

Figure A.2: *Measurement result. The data is corrected for the laser measurement error, which was determined from redundant measurement data, i.e., points that were measured multiple times from different corners. The laser error is 0mm at 4m at its leftmost corner and +2mm at 4m at its rightmost corner. The difference between the lowest and highest measured point is 15mm.*

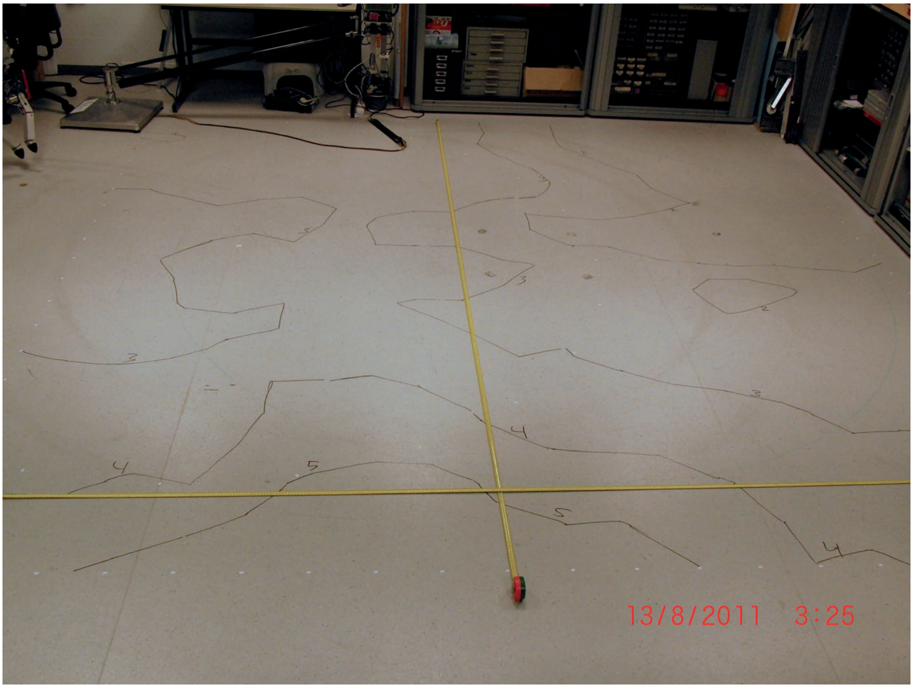
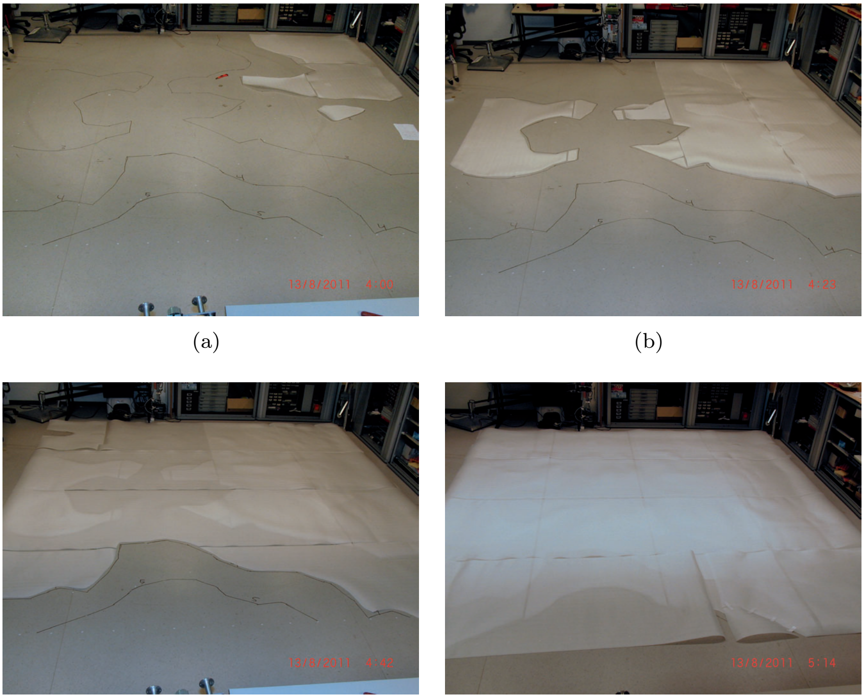


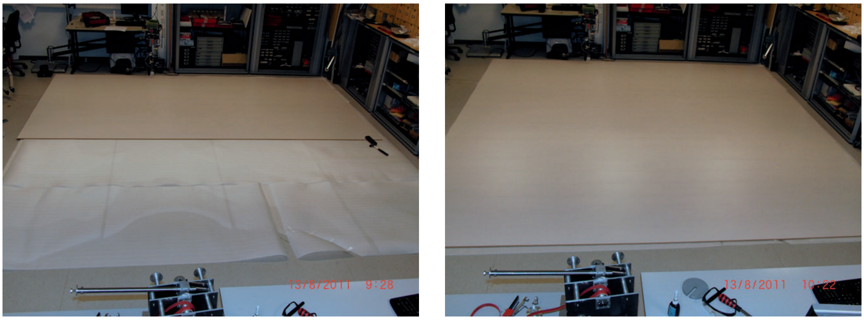
Figure A.3: *The contour plot from Figure A.2(b) was copied to the floor.*



(a) (b)

(c) (d)

Figure A.4: *Layers of 3mm thick foam were placed on the floor to level it.*



(a) (b)

Figure A.5: *Laminate was placed on top of the foam layers.*

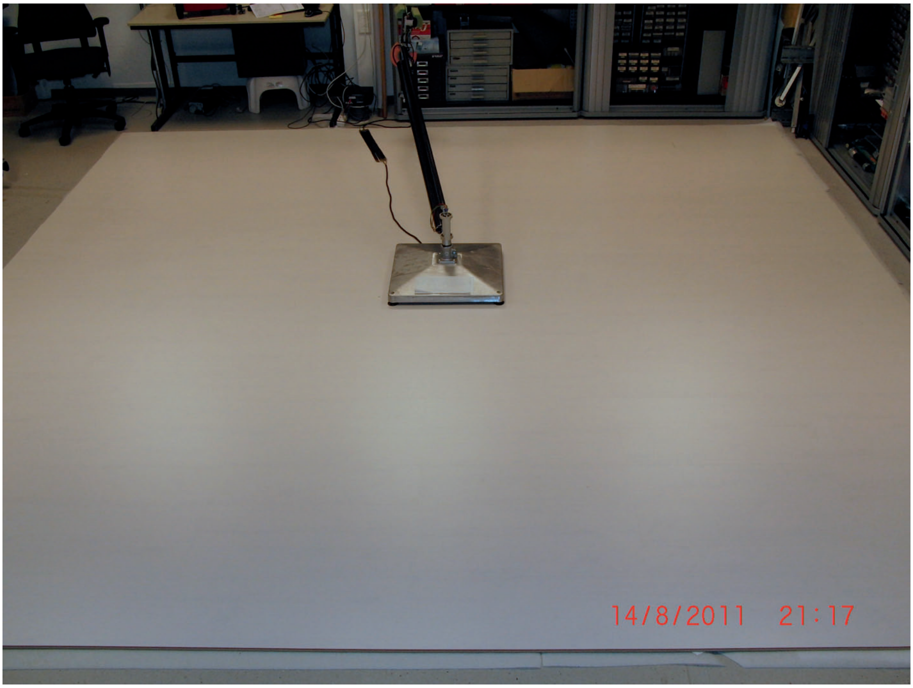


Figure A.6: *Leo on the new level floor. After leveling, the difference between the lowest and highest measured point (randomly sampled) was 3mm.*

Appendix B

Additional results

In Section 3.3.1, an experiment was conducted in which robot Leo learned to perform a stairs step-up task. This experiment was repeated 12 times on the prototype. Due to a software flaw in the experiment length computation, however, the experiments are of unequal duration. Figure 3.13 therefore only shows the average learning performance up to approximately 13 minutes of experimentation time – the duration of the shortest experiment. To illustrate the convergence behavior of individual runs – and to illustrate the spread between individual learning experiments in general – this appendix presents the learning curves of all stairs step-up learning experiments performed on robot Leo. They are presented in Figure B.1.

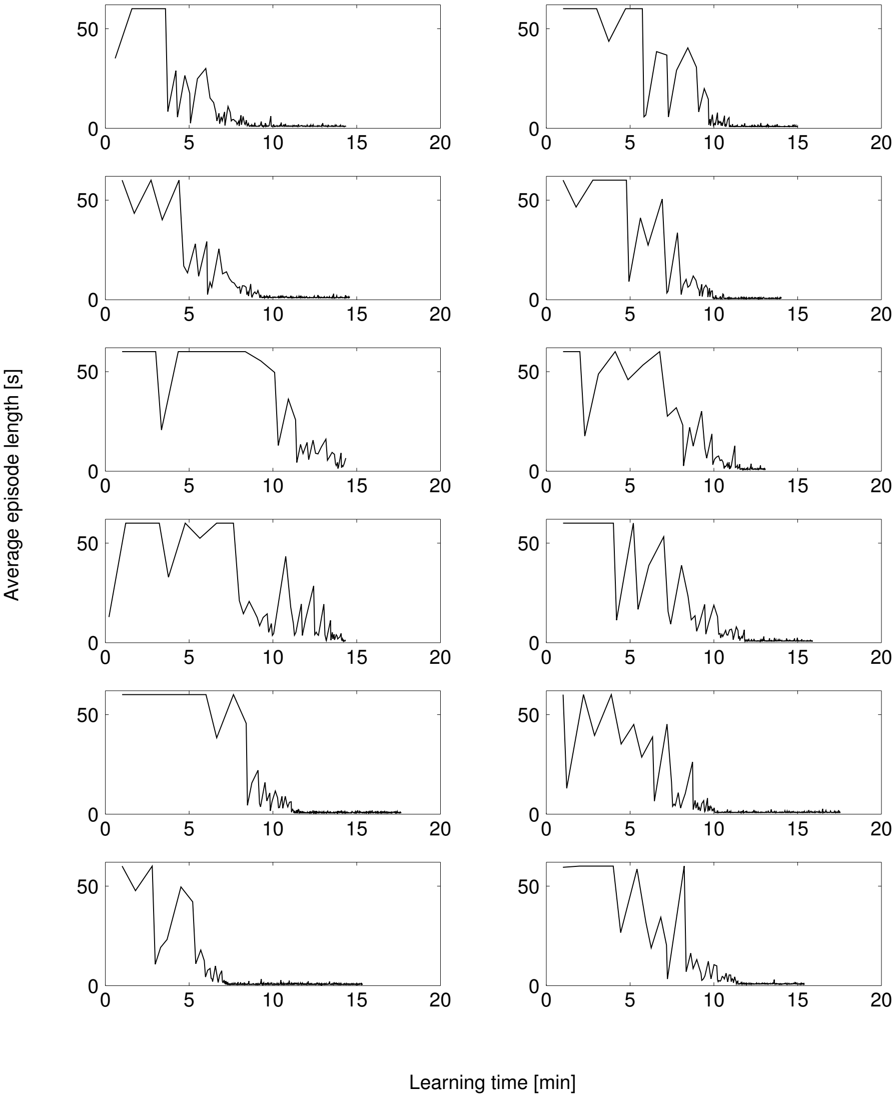


Figure B.1: *Robot Leo learning the stairs step-up task (real robot results), showing the learning performance of individual experiments. The episode length (lower is better) is plotted against learning time. Due to a flaw in the experiment length computation (the regular motor cooling pauses were accidentally included when computing the experiment length), individual experiments are of unequal length.*

Bibliography

Albus, J.S. (1971). “A theory of cerebellar function”. In: *Mathematical Biosciences* 10.1-2, pp. 25–61. (Cit. on pp. 16, 18).

— (1981). *Brains, behavior and robotics.* McGraw-Hill, Inc. New York, USA. (Cit. on pp. 16, 18).

An, P.C.E. (1991). “An improved multi-dimensional CMAC neural network: receptive field function and placement”. PhD thesis. University of New Hampshire. (Cit. on p. 20).

Anderson, C.W., Young, P.M., Buehner, M.R., Knight, J.N., Bush, K.A., and Hittle, D.C. (2007). “Robust Reinforcement Learning Control Using Integral Quadratic Constraints for Recurrent Neural Networks”. In: *IEEE Transactions on Neural Networks* 18.4, pp. 993–1002. (Cit. on p. 88).

Appleby, A. (2008). *SMHasher MurmurHash.* URL: `http://code.google.com/p/smhasher/`. (Cit. on p. 22).

Argall, B.D., Chernova, S., Veloso, M., and Browning, B. (2009). “A survey of robot learning from demonstration”. In: *Robotics and Autonomous Systems* 57.5, pp. 469 –483. (Cit. on p. 2).

Atkeson, C.G., Moore, A.W., and Schaal, S. (1997). “Locally weighted learning”. In: *Artificial intelligence review* 11.1, pp. 11–73. (Cit. on p. 91).

Baillie, J.C. (2004). “URBI: Towards a universal robotic body interface”. In: *Proceedings of the 4th International Conference on Humanoids Robotics.* URL: `http://www.gostai.com/`. (Cit. on p. 52).

Baird, L. (1995). “Residual Algorithms: Reinforcement Learning with Function Approximation”. In: *Proceedings of the Twelfth International Converence on Machine Learning.* Morgan Kaufmann, pp. 30–37. (Cit. on p. 16).

Benbrahim, H. (1996). “Biped dynamic walking using reinforcement learning”. PhD thesis. University of New Hampshire. (Cit. on p. 3).

Berg, I. (2010). *muParser – a fast math parser library.* URL: `http://muparser.sourceforge.net/`. (Cit. on p. 56).

Bernstein, A. and Shimkin, N. (2010). “Adaptive-resolution reinforcement learning with polynomial exploration in deterministic domains”. In: *Machine learning*, pp. 1–39. (Cit. on pp. 20, 129).

Bertsekas, D.P. (2007). *Dynamic Programming and Optimal Control, Vol. II.* 3rd. Athena Scientific. (Cit. on pp. 2, 9).

— (1987). *Dynamic programming: deterministic and stochastic models.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. (Cit. on p. 87).

Bhat, S., Isbell, C.L., and Mateas, M. (2006). “On the difficulty of modular reinforcement learning for real-world partial programming”. In: *Proceedings of the National Conference on Artificial Intelligence.* Vol. 21. 1, p. 318. (Cit. on p. 130).

Bhatnagar, S., Sutton, R., Ghavamzadeh, M., and Lee, M. (2008). “Incremental natural actor-critic algorithms”. In: *Advances in Neural Information Processing Systems* 20, pp. 105–112. (Cit. on p. 13).

Bhatnagar, S., Sutton, R.S., Ghavamzadeh, M., and Lee, M. (2009). “Natural actor-critic algorithms”. In: *Automatica* 45.11, pp. 2471–2482. (Cit. on p. 13).

Boutilier, C. (1996). “Planning, learning and coordination in multiagent decision processes”. In: *Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, pp. 195–210. (Cit. on p. 104).

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). “The Real-Time Motion Control Core of the Orocos Project”. In: *IEEE International Conference on Robotics and Automation*, pp. 2766–2771. URL: `http://www.orocos.org/`. (Cit. on p. 51).

Buşoniu, L., De Schutter, B., and Babuška, R. (Aug. 2006). “Decentralized reinforcement learning control of a robotic manipulator”. In: *International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pp. 1–6. (Cit. on pp. 102, 109).

Buşoniu, L., Ernst, D., De Schutter, B., and Babuška, R. (2007). “Fuzzy Approximation for Convergent Model-Based Reinforcement Learning”. In: *IEEE International Conference on Fuzzy Systems (FUZZ-IEEE).* (Cit. on pp. 16, 18, 31, 109).

Cherubini, A., Giannone, F., Iocchi, L., Lombardo, M., and Oriolo, G. (2009). “Policy gradient learning for a humanoid soccer robot”. In: *Robotics and Autonomous Systems* 57.8. Humanoid Soccer Robots, pp. 808 –818. (Cit. on p. 3).

Chow, C.S. and Tsitsiklis, J.N. (1991). “An optimal one-way multigrid algorithm for discrete-time stochastic control”. In: *IEEE Transactions on Automatic Control* 36.8, pp. 898–914. (Cit. on p. 20).

Claus, C. and Boutilier, C. (1998). “The dynamics of reinforcement learning in cooperative multiagent systems”. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence* 746, p. 752. (Cit. on pp. 102, 104, 106).

Collett, T.H.J., MacDonald, B.A., and Gerkey, B.P. (Dec. 2005). “Player 2.0: Toward a Practical Robot Programming Framework”. In: *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005).* Sydney, Australia. URL: `http://playerstage.sourceforge.net/`. (Cit. on p. 51).

Collins, S., Ruina, A., Tedrake, R., and Wisse, M. (2005). "Efficient bipedal robots based on passive-dynamic walkers". In: *Science* 307.5712, p. 1082. (Cit. on pp. 35, 36).

Collins, S.H., Wisse, M., and Ruina, A. (2001). "A two legged kneed passive dynamic walking robot". In: *Int. J. of Robotics Research* 20.7, pp. 607–615. (Cit. on p. 115).

Crites, R. and Barto, A. (1996). "Improving Elevator Performance Using Reinforcement Learning". In: *Advances in Neural Information Processing Systems 8*. MIT Press, pp. 1017–1023. (Cit. on p. 2).

Crites, R.H. and Barto, A.G. (1998). "Elevator Group Control Using Multiple Reinforcement Learning Agents". In: *Machine Learning* 33 (2), pp. 235–262. (Cit. on p. 102).

Dasgupta, S. (Feb. 2010). "Strange effects in high dimension: technical perpective". In: *Communications of the ACM* 53 (2), pp. 96–96. (Cit. on p. 20).

Denniss, W. (2004). *XODE - XML ODE data interchange format*. URL: `http://tanksoftware.com/xode/`. (Cit. on pp. 52, 58).

Diankov, R. and Kuffner, J. (July 2008). *OpenRAVE: A Planning Architecture for Autonomous Robotics*. Tech. rep. CMU-RI-TR-08-34. Robotics Institute. URL: `http://openrave.programmingvision.com`. (Cit. on p. 52).

Dietterich, T.G. (2000). "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". In: *Journal of Artificial Intelligence Research* 13, pp. 227–303. (Cit. on p. 55).

Doya, K. (2001). "Robust Reinforcement Learning". In: *Advances in Neural Information Processing Systems*, pp. 1061–1067. (Cit. on p. 88).

Garcia, M., Chatterjee, A., Ruina, A., and Coleman, M. (1998). "The simplest walking model: Stability, complexity, and scaling". In: *ASME Journal of Biomechanical Engineering* 120, pp. 281–288. (Cit. on p. 92).

Garca, J. and Fernndez, F. (2011). "Safe Reinforcement Learning in High-Risk Tasks through Policy Improvement". In: *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*. Paris, France. (Cit. on p. 129).

Geibel, P. (2001). "Reinforcement learning with bounded risk". In: *Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 162–169. (Cit. on p. 127).

Gerum, P. (2004). *Xenomai - Implementing a RTOS framework on GNU/Linux*. URL: `http://xenomai.org`. (Cit. on p. 49).

Gomes, E.R. and Kowalczyk, R. (2009). "Dynamic analysis of multiagent Q-learning with ε-greedy exploration". In: *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 369–376. (Cit. on p. 105).

Gordon, G.J. (2001). "Reinforcement learning with function approximation converges to a region". In: *Advances in Neural Information Processing Systems* 13, pp. 1040–1046. (Cit. on p. 16).

— (1995). "Stable Function Approximation in Dynamic Programming". In: *Proceedings of the Twelfth International Conference on Machine Learning, Tahoe*

City, California, July 9-12, 1995. Morgan Kaufmann, pp. 261–268. (Cit. on p. 16).

Grondman, I., Vaandrager, M., Buşoniu, L., Babuška, R., and Schuitema, E. (June 2012). "Efficient Model Learning Methods for Actor-Critic Control". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 42.3, pp. 591 –602. (Cit. on p. 157).

Grzes, M. and Kudenko, D. (2010). "Reward Shaping and Mixed Resolution Function Approximation". In: *Developments in Intelligent Agent Technologies and Multi-Agent Systems: Concepts and Applications*, p. 95. (Cit. on p. 129).

Hans, A., Schneegaß, D., Schäfer, A.M., and Udluft, S. (2008). "Safe exploration for reinforcement learning". In: *Proceedings of the European Symposium on Artificial Neural Networks*, pp. 413–418. (Cit. on pp. 129, 130).

Hennes, D., Tuyls, K., and Rauterberg, M. (2008). "Formalizing multi-state learning dynamics". In: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology.* Vol. 2, pp. 266–272. (Cit. on p. 108).

— (2009). "State-coupled replicator dynamics". In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems.* Vol. 2, pp. 789–796. (Cit. on p. 108).

Hobbelen, D., De Boer, T., and Wisse, M. (Sept. 2008). "System overview of bipedal robots Flame and TUlip: Tailor-made for Limit Cycle Walking". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* Nice, France, pp. 2486 –2491. (Cit. on pp. 39, 79).

Hobbelen, D.G.E. (2008). "Limit Cycle Walking". PhD thesis. Delft University of Technology. (Cit. on pp. 35, 39, 62, 76, 115).

Horiuchi, T., Fujino, A., Katai, O., and Sawaragi, T. (Sept. 1996). "Fuzzy interpolation-based Q-learning with continuous states and actions". In: *Fuzzy Systems, 1996., Proceedings of the Fifth IEEE International Conference on.* Vol. 1, pp. 594–600. (Cit. on p. 18).

IFR Statistical Department (2010). *World Robotics 2010 – Service Robots.* Tech. rep. VDMA Robotics + Automation association. (Cit. on p. 1).

Ito, K., Takayama, A., and Kobayashi, T. (2009). "Hardware design of autonomous snake-like robot for reinforcement learning based on environment". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2622–2627. (Cit. on p. 3).

Jaakkola, T., Jordan, M.I., and Singh, S.P. (1994). "Convergence of Stochastic Iterative Dynamic Programming Algorithms". In: *Neural Computation* 6, pp. 1185–1201. (Cit. on p. 87).

Jouffe, L. (1998). "Fuzzy inference system learning by reinforcement learning". In: *IEEE Transactions on Systems, Man and Cybernetics* 28.3, pp. 338–355. (Cit. on p. 18).

Kalmár, Z., Szepesvári, C., and Lörincz, A. (1998). "Module-based reinforcement learning: Experiments with a real robot". In: *Machine Learning* 31.1, pp. 55–85. (Cit. on p. 3).

Kamio, S. and Iba, H. (2005). “Adaptation technique for integrating genetic programming and reinforcement learning for real robots”. In: *IEEE Transactions on Evolutionary Computation* 9.3, pp. 318–333. (Cit. on p. 3).

Karssen, J.G.D. and Wisse, M. (2012). “Running Robot Phides”. In preparation. (Cit. on p. 63).

Katsikopoulos, K.V. and Engelbrecht, S.E. (2003). “Markov decision processes with delays and asynchronous cost collection”. In: *IEEE Transactions on Automatic Control* 48.4, pp. 568–574. (Cit. on pp. 23–25).

Kober, J. and Peters, J. (2009). “Learning motor primitives for robotics”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2112–2118. (Cit. on p. 3).

— (2012). “Reinforcement Learning in Robotics: A Survey”. In: *Reinforcement Learning.* Vol. 12. Adaptation, Learning, and Optimization. Springer Berlin Heidelberg, pp. 579–610. (Cit. on p. 13).

Koenig, N. and Howard, A. (Sept. 2004). “Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator”. In: *Proceedings of IROS.* Sendai, Japan. (Cit. on pp. 52, 58).

Kohl, N. and Stone, P. (2004). “Policy gradient reinforcement learning for fast quadrupedal locomotion”. In: *IEEE International Conference on Robotics and Automation (ICRA).* (Cit. on pp. 2, 3, 129).

Kretchmar, R.M. and Anderson, C.W. (1997). “Comparison of CMACs and radial basis functions for local function approximators in reinforcement learning”. In: *International Conference on Neural Networks.* Vol. 2, pp. 834–837. (Cit. on p. 16).

Kretchmar, R.M., Young, P.M., Anderson, C.W., Hittle, D.C., Anderson, M.L., and Delnero, C.C. (2001). “Robust Reinforcement Learning Control with Static and Dynamic Stability”. In: *Intl. Journal of Robust and Nonlinear Control* 2001, pp. 1469–1500. (Cit. on p. 88).

Lagoudakis, M.G. and Parr, R. (2003). “Least-squares policy iteration”. In: *The Journal of Machine Learning Research* 4, pp. 1107–1149. (Cit. on p. 129).

Lane, S.H., Handelman, D.A., and Gelfand, J.J. (Apr. 1992). “Theory and development of higher-order CMAC neural networks”. In: *IEEE Control Systems Magazine* 12.2, pp. 23 –30. (Cit. on p. 20).

Laud, A.D. (2004). “Theory and application of reward shaping in reinforcement learning”. PhD thesis. University of Illinois. (Cit. on p. 12).

Lauer, M. and Riedmiller, M.A. (2000). “An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems”. In: *Proceedings of the Seventeenth International Conference on Machine Learning.* ICML ’00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 535–542. (Cit. on pp. 104, 106–108).

Laurent, G.J., Matignon, L., and Fort-Piat, N.L. (2011). “The world of independent learners is not markovian”. In: *International Journal of Knowledge-based and Intelligent Engineering Systems* 15.1, pp. 55–64. (Cit. on pp. 102, 104).

Lima, J.L., Gonçalves, J.A., Costa, P.G., and Moreira, A.P. (2009). "Humanoid Realistic Simulator - The Servomotor Joint Modeling". In: *ICINCO-RA*, pp. 396–400. (Cit. on p. 59).

Lin, L.J. (1993). "Reinforcement learning for robots using neural networks". PhD thesis. Pittsburgh, PA, USA. (Cit. on p. 129).

Loch, J. and Singh, S. (1998). "Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes". In: *Proceedings of the fifteenth International Conference on Machine Learning*, pp. 141–150. (Cit. on p. 25).

Marthi, B. (2007). "Automatic shaping and decomposition of reward functions". In: *Proceedings of the 24th international conference on Machine learning.* ACM, pp. 601–608. (Cit. on pp. 12, 129).

Mataric, M.J. (1994). "Reward Functions for Accelerated Learning". In: *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, pp. 181–189. (Cit. on p. 12).

Matignon, L., Laurent, G.J., Le Fort-Piat, N., and Chapuis, Y.A. (2010). "Designing decentralized controllers for distributed-air-jet mems-based micromanipulators by reinforcement learning". In: *Journal of intelligent & robotic systems* 59.2, pp. 145–166. (Cit. on p. 102).

McGeer, T. (1990). "Passive Dynamic Walking". In: *The International Journal of Robotics Research* 9.2, p. 62. (Cit. on p. 115).

Metta, G., Fitzpatrick, P., and Natale, L. (2006). "YARP: Yet Another Robot Platform". In: *International Journal of Advanced Robotic Systems* 3.1, pp. 43–48. URL: `http://eris.liralab.it/yarp/`. (Cit. on p. 51).

Michel, O. (2004). "WebotsTM: Professional Mobile Robot Simulation". In: *International Journal of Advanced Robotic Systems* 1.1, pp. 39–42. (Cit. on p. 52).

Mihatsch, O. and Neuneier, R. (2002). "Risk-Sensitive Reinforcement Learning". In: *Machine Learning* 49 (2), pp. 267–290. (Cit. on p. 128).

Miller, W.T., An, E., Glanz, F., and Carter, M. (1990). "The design of cmac neural networks for control". In: *Adaptive and Learning Systems* 1, pp. 140–145. (Cit. on pp. 19, 20).

Moody, J. and Saffell, M. (July 2001). "Learning to trade via direct reinforcement". In: *IEEE Transactions on Neural Networks* 12.4, pp. 875 –889. (Cit. on p. 2).

Moore, A.W. and Atkeson, C.G. (1993). "Prioritized sweeping: Reinforcement learning with less data and less time". In: *Machine Learning* 13.1, pp. 103–130. (Cit. on p. 129).

— (1995). "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces". In: *Machine Learning* 21.3, pp. 199–233. (Cit. on p. 20).

Morimoto, J. and Doya, K. (2001). "Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning". In: *Robotics and Autonomous Systems* 36.1, pp. 37–51. (Cit. on pp. 3, 13).

Morimoto, J., Nakanishi, J., Endo, G., Cheng, G., Atkeson, C.G., and Zeglin, G. (Apr. 2005). “Poincaré-Map-Based Reinforcement Learning For Biped Walking”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2381 –2386. (Cit. on p. 3).

Munos, R. and Moore, A. (2002). “Variable resolution discretization in optimal control”. In: *Machine Learning* 49.2, pp. 291–323. (Cit. on pp. 20, 129).

Nakanishi, J., Morimoto, J., Endo, G., Cheng, G., Schaal, S., and Kawato, M. (2004). “Learning from demonstration and adaptation of biped locomotion”. In: *Robotics and Autonomous Systems* 47.2-3, pp. 79–91. (Cit. on p. 13).

Nash Jr, J.F. (1951). “Non-Cooperative Games”. In: *Annals of Mathematics* 54.2. (Cit. on p. 106).

Nemec, B., Zorko, M., and Zlajpah, L. (2010). “Learning of a ball-in-a-cup playing robot”. In: *Proceedings of the IEEE 19th International Workshop on Robotics in Alpe-Adria-Danube Region (RAAD).* (Cit. on p. 3).

Neumann, G. (2005). “The Reinforcement Learning Toolbox, Reinforcement Learning for Optimal Control Tasks”. MA thesis. Institut für Grundlagen der Informationsverarbeitung, Technischen Universit”at Graz. URL: `http://www.igi.tugraz.at/ril-toolbox/general/overview.html`. (Cit. on p. 52).

Ng, A., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2004). “Autonomous inverted helicopter flight via reinforcement learning”. In: *International Symposium on Experimental Robotics.* Singapore. (Cit. on p. 2).

Ng, A.Y., Harada, D., and Russell, S.J. (1999). “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. In: *Proceedings of the Sixteenth International Conference on Machine Learning.* Morgan Kaufmann Publishers Inc., pp. 278–287. (Cit. on pp. 12, 129).

Nissen, S. (2003). “Implementation of a Fast Artificial Neural Network Library (FANN)”. MA thesis. Department of Computer Science, University of Copenhagen. URL: `http://leenissen.dk/fann/`. (Cit. on p. 52).

Ogino, M., Katoh, Y., Aono, M., Asada, M., and Hosoda, K. (2004). “Reinforcement learning of humanoid rhythmic walking parameters based on visual information”. In: *Advanced Robotics* 18.7, pp. 677–697. (Cit. on p. 3).

Ormoneit, D. and Sen, Ś. (2002). “Kernel-based reinforcement learning”. In: *Machine learning* 49.2, pp. 161–178. (Cit. on p. 129).

Panait, L. and Luke, S. (2005). “Cooperative Multi-Agent Learning: The State of the Art”. In: *Autonomous Agents and Multi-Agent Systems* 11.3, pp. 387–434. (Cit. on p. 106).

Panait, L., Sullivan, K., and Luke, S. (2006). “Lenience towards Teammates Helps in Cooperative Multiagent Learning”. In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multi Agent Systems–AAMAS-2006, ACM Press, New York.* (Cit. on pp. 102, 106, 108).

Panait, Liviu, Tuyls, Karl, and Luke, Sean (2008). “Theoretical Advantages of Lenient Learners: An Evolutionary Game Theoretic Perspective”. In: *Journal of Machine Learning Research* 9.Mar, pp. 423–457. (Cit. on p. 106).

Peters, J. and Schaal, S. (2008). “Natural actor-critic”. In: *Neurocomputing* 71.7-9, pp. 1180–1190. (Cit. on pp. 3, 13).

— (2006). “Policy gradient methods for robotics”. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2219–2225. (Cit. on p. 3).

Peters, J., Vijayakumar, S., and Schaal, S. (2003). “Reinforcement learning for humanoid robotics”. In: *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots.* (Cit. on pp. 3, 13, 129).

Petters, S., Thomas, D., and Von Stryk, O. (Oct. 2007). “RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots”. In: *Proc. Workshop on Measures and Procedures for the Evaluation of Robot Archtectures and Middleware of the 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems.* San Diego, CA, USA. URL: `http://robocup.informatik.tu-darmstadt.de/humanoid/research/architecture.en.php`. (Cit. on p. 52).

Precup, D., Sutton, R.S., and Dasgupta, S. (2001). “Off-policy temporal-difference learning with function approximation”. In: *Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 417–424. (Cit. on p. 16).

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A.Y. (2009). “ROS: an open-source Robot Operating System”. In: *ICRA Workshop on Open Source Software.* URL: `http://www.ros.org/wiki/`. (Cit. on p. 51).

Rencher, A.C. and Schaalje, G.B. (2000). *Linear models in statistics.* Wiley New York. (Cit. on p. 45).

Riedmiller, M., Gabel, T., Hafner, R., and Lange, S. (2009). “Reinforcement learning for robot soccer”. In: *Autonomous Robots* 27.1, pp. 55–73. (Cit. on p. 3).

Russell, S.J. and Zimdars, A. (2003). “Q-Decomposition for Reinforcement Learning Agents”. In: *Proceedings of the Twentieth International Conference on Machine Learning*, pp. 656–663. (Cit. on pp. 130, 132, 133).

Salatian, A.W., Yi, K.Y., and Zheng, Y.F. (1997). “Reinforcement Learning for a Biped Robot to Climb Sloping Surfaces”. In: *Journal of Robotic Systems* 14.4, pp. 283–296. (Cit. on p. 3).

Salkham, A., Cunningham, R., Garg, A., and Cahill, V. (Dec. 2008). “A Collaborative Reinforcement Learning Approach to Urban Traffic Control Optimization”. In: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT).* Vol. 2, pp. 560 –566. (Cit. on p. 2).

Samejima, K., Doya, K., and Kawato, M. (2003). “Inter-module credit assignment in modular reinforcement learning”. In: *Neural Networks* 16.7, pp. 985 –994. (Cit. on p. 130).

Schaal, S., IJspeert, A., and Billard, A. (2003). “Computational approaches to motor learning by imitation”. In: *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences* 358.1431, pp. 537–547. (Cit. on p. 2).

Schaal, S., Peters, J., Nakanishi, J., and IJspeert, A. (2005). “Learning movement primitives”. In: *Robotics Research*, pp. 561–572. (Cit. on p. 2).

Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., and Schmidhuber, J. (2010). “PyBrain”. In: *Journal of Machine Learning Research* 11. URL: `http://pybrain.org/`. (Cit. on p. 52).

Schuitema, E., Buşoniu, L., Babuška, R., and Jonker, P. (2010a). “Control Delay in Reinforcement Learning for Real-Time Dynamic Systems: a Memoryless Approach”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. (Cit. on pp. 22, 47).

Schuitema, E., Wisse, M., Ramakers, T., and Jonker, P. (2010b). “The Design of LEO: a 2D Bipedal Walking Robot for Online Autonomous Reinforcement Learning”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. (Cit. on p. 35).

Schuitema, E., Caarls, W., Wisse, M., Jonker, P., and Babuška, R. (Oct. 2010c). “The Effects of Large Disturbances on On-Line Reinforcement Learning for a Walking Robot”. In: *Proceedings of the 22nd Benelux Conference on Artificial Intelligence (BNAIC)*. Luxembourg. (Cit. on p. 87).

Schuitema, E., Hobbelen, D.G.E., Jonker, P.P., Wisse, M., and Karssen, J.G.D. (2005). “Using a controller based on reinforcement learning for a passive dynamic walking robot”. In: *5th IEEE-RAS International Conference on Humanoid Robots*, pp. 232–237. (Cit. on pp. 5, 6, 20, 37, 39, 115, 117).

Singh, S., Jaakkola, T., Littman, M.L., and Szepesvári, C. (2000). “Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms”. In: *Machine Learning* 38 (3), pp. 287–308. (Cit. on p. 27).

Singh, S.P., Jaakkola, T., and Jordan, M.I. (1994). “Learning without state-estimation in partially observable Markovian decision processes”. In: *Proceedings of the eleventh International Conference on Machine Learning*, pp. 284–292. (Cit. on p. 25).

Singh, S.P., Barto, A.G., Grupen, R., and Connolly, C. (1994). “Robust reinforcement learning in motion planning”. In: *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, pp. 655–662. (Cit. on p. 88).

Smart, W.D. and Kaelbling, L.P. (2000). “Practical Reinforcement Learning in Continuous Spaces”. In: *Proceedings of the Seventeenth International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc., pp. 903–910. (Cit. on p. 3).

Smith, R. (2011). *Open Dynamics Engine*. `http://ode.org/`. (Cit. on pp. 52, 56).

Smith, R. et al. (2006). “Open Dynamics Engine”. In: *Computer Software. From http://www.ode.org*. (Cit. on p. 115).

Sondik, E.J. (1978). “The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs”. In: *Operations Research*, pp. 282–304. (Cit. on p. 11).

Sprague, N. and Ballard, D. (2003). “Multiple-Goal Reinforcement Learning with Modular Sarsa(0)”. In: *International Joint Conference on Artificial Intelligence.* Acapulco. (Cit. on pp. 130, 132).

Stone, P. and Sutton, R.S. (2001). “Scaling reinforcement learning toward RoboCup soccer”. In: *Proceedings of the Eighteenth International Conference on Machine Learning*, 537544. (Cit. on pp. 2, 16).

Stone, P., Sutton, R.S., and Kuhlmann, G. (2005). “Reinforcement learning for robocup soccer keepaway”. In: *Adaptive Behavior* 13.3, p. 165. (Cit. on p. 20).

Stone, P. and Veloso, M. (2000). “Multiagent Systems: A Survey from a Machine Learning Perspective”. In: *Autonomous Robots* 8 (3), pp. 345–383. (Cit. on p. 102).

Sutton, R.S. (July 1991). “Dyna, an integrated architecture for learning, planning, and reacting”. In: *SIGART Bull.* 2 (4), pp. 160–163. (Cit. on p. 129).

— (1996). “Generalization in reinforcement learning: Successful examples using sparse coarse coding”. In: *Advances in Neural Information Processing Systems* 8, pp. 1038–1044. (Cit. on pp. 16, 20).

Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction.* MIT Press. (Cit. on pp. 2, 9, 15–17, 21, 24, 32, 55, 129, 132).

Sutton, R.S., Szepesvári, C., and Maei, H.R. (2009). “A convergent O (n) algorithm for off-policy temporal-difference learning with linear function approximation”. In: *Advances in Neural Information Processing Systems* 21. (Cit. on p. 16).

Sutton, R.S., Szepesvári, C., Geramifard, A., and Bowling, M. (2008). “Dyna-style planning with linear function approximation and prioritized sweeping”. In: *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence.* Vol. 23. (Cit. on p. 129).

Sutton, R.S., Modayil, J., Delp, M., Degris, T., Pilarski, P.M., White, A., and Precup, D. (2011). “Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction”. In: *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems.* (Cit. on pp. 3, 6).

Tedrake, R., Zhang, T.W., and Seung, H.S. (2004). “Stochastic policy gradient reinforcement learning on a simple 3D biped”. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* (Cit. on pp. 3, 4, 13, 129).

Tesauro, G. (1995). “Temporal difference learning and TD-Gammon”. In: *Communications of the ACM* 38.3, pp. 58–68. (Cit. on p. 2).

Troost, S., Schuitema, E., and Jonker, P.P. (2008). “Using cooperative multi-agent Q-learning to achieve action space decomposition within single robots”. In: *18th European Conference on Artificial Intelligence, workshop ERLARS.* Patras, Greece. (Cit. on pp. 37, 39, 101).

Tsitsiklis, J.N. (1994). “Asynchronous Stochastic Approximation and Q-Learning”. In: *Machine Learning*, pp. 185–202. (Cit. on p. 87).

Uchibe, E., Asada, M., and Hosoda, K. (1996). “Behavior coordination for a mobile robot using modular reinforcement learning”. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3, pp. 1329–1336. (Cit. on p. 130).

Van Diepen, M. (2011). “Avoiding failure states during Reinforcement Learning”. MSc thesis. Delft University of Technology. (Cit. on p. 127).

Van Vliet, B., Caarls, W., Schuitema, E., and Jonker, P. (Nov. 2011). “Accelerating reinforcement learning on a robot by using subgoals in a hierarchical framework”. In: *Proceedings of the 23rd Benelux Conference on Artificial Intelligence (BNAIC)*. Ghent, Belgium. (Cit. on p. 157).

Vaughan, R.T. (2000). *Stage: A Multiple Robot Simulator*. Tech. rep. IRIS-00-394. Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California. (Cit. on p. 52).

Vrancx, P., Tuyls, K., and Westra, R. (2008). “Switching dynamics of multiagent learning”. In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*. Vol. 1, pp. 307–313. (Cit. on p. 108).

Vukobratovic, M. and Borovac, B. (2004). “Zero-moment point – thirty five years of its life”. In: *International Journal of Humanoid Robotics* 1.1, pp. 157–173. (Cit. on p. 76).

Vukobratovic, M., Frank, A.A., and Juricic, D. (1970). “On the stability of biped locomotion”. In: *IEEE Transactions on Biomedical Engineering* 1, pp. 25–36. (Cit. on p. 76).

Walsh, T.J., Nouri, A., Li, L., and Littman, M.L. (2009). “Learning and planning in environments with delayed feedback”. In: *Autonomous Agents and Multi-Agent Systems* 18.1, pp. 83–105. (Cit. on pp. 23–25, 29, 33, 47).

Whiteson, S. (2010). “Adaptive Tile Coding”. In: *Adaptive Representations for Reinforcement Learning*, pp. 65–76. (Cit. on pp. 20, 129).

WHO (2007). “Investing in the health workforce enables stronger health systems”. In: *Fact sheet 2007, Belgrade,Copenhagen*. (Cit. on p. 1).

Yamanishi, K., Takeuchi, J.I., Williams, G., and Milne, P. (2000). “On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms”. In: *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Boston, Massachusetts, United States: ACM, pp. 320–324. (Cit. on p. 91).

Summary

The assistance of service robots in households, health care and other labour intensive environments is expected to become increasingly important in the near future. Whereas factory robots are widely employed in production facilities worldwide, service robots are still a novelty and are currently only capable of simple tasks such as vacuum cleaning and lawn mowing. The main difference lies in the environment: where factories are structured and predictable, domestic environments are typically unique and continually changing. For service robots to be successful, they need to be versatile and able to perform emerging tasks in various environments. Since this variety of tasks and environments cannot be completely foreseen and tested at the robot's production time, manually programming such robots becomes complicated. Having robots *learn* task solutions autonomously through interaction with the real world forms an attractive alternative.

Reinforcement Learning (RL) is a generic machine learning paradigm that has been applied to a wide range of problems. Through interaction with the environment, an RL system is able to autonomously learn task solutions by continuously improving its behavior using only coarse feedback: desired behavior is reinforced by positive rewards; undesired behavior is punished by negative rewards. Because of its generic formulation and ability to learn from real experience, RL is a promising candidate for adding learning capabilities to service robots.

Unfortunately, relatively little is known on how to successfully apply RL to *real* robots. The goal of this thesis is to identify and address the difficulties in hardware design, software design and RL theory that currently prevent the application of RL to real, autonomous service robots. To this end, RL techniques are selected from the literature that are considered suitable for our particular purpose, based on their theoretical properties and on existing simulation results. The requirements that these RL techniques pose on the hardware and software are first identified and then realized in a prototype: bipedal walking robot Leo. The central learning task in this thesis is the task of learning to walk for a bipedal robot, which is both challenging and interesting to solve with RL.

This thesis starts by motivating the choice for Temporal Difference (TD) learn-

ing, followed by additional techniques, such as tile coding function approximation, that enable it to be applied specifically to robotic tasks. In addition, it is shown that control delay, i.e., the time delay between sensing and acting, is likely to occur on real robots and can have a strong negative effect on the learning performance. A new TD learning algorithm is introduced – dSARSA(λ) – that can perform better than its 'vanilla' counterpart SARSA(λ) by taking into account the control delay in the learning updates without adding computational complexity.

After deriving the hardware and software requirements that follow from the selected RL techniques, the design of bipedal walking robot Leo is presented. This prototype serves as a RL research platform throughout the thesis for a number of experiments and is accompanied by a realistic simulation environment. Experiments conducted in simulation and on the real robot show that it is possible to employ TD learning to learn non-trivial tasks such as walking (in 5 hours on average) and placing a foot on a step of stairs (in 15 minutes on average), in real-time, on embedded hardware, despite the high dimensional state-action spaces of these tasks. While the prototype met most of the derived requirements, its robustness fell short in facing the trial-and-error nature of TD learning – it required repair every 30 minutes on average – which prevented repeating the experiments on the real robot up to statistical significance. The experiments revealed a number of practical complications, which are addressed in the remainder of the thesis.

The effects on the learning process of large and infrequent disturbances, such as floor irregularities and unexpected interactions with the environment such as a push, are relatively unknown. This thesis presents simulation results on a simple walking model that show that large and infrequent disturbances of the actual system state, e.g., by means of pushing the robot, have a dramatic effect on the learning time and system performance. Outliers in the sampling period of the system and in sensor readings, on the other hand, have a negligible effect on the learning process. A method is presented that restores the learning performance by excluding the outliers from the learning process.

To reduce the control delay caused by the learning controller, this thesis proposes to use Independent Learners (IL) for the individual actuators of a robot, creating an architecture in which individual, independent learning systems each control a single actuator. This decentralized approach reduces the computation time of the learning system's decision making process and thus the control delay.

In order to reduce the hardware strain of TD learning, this thesis presents a solution based on Modular Reinforcement Learning (MRL) in which the robot coarsely but quickly learns about the risk of its behavior. This reduces the robot's cumulative risk exposure while learning the actual task solution.

In summary, this thesis shows that TD learning has potential to let real robots autonomously learn non-trivial tasks. The main complication is that TD learning requires very robust hardware due to its trial-and-error nature. Furthermore, it is sensitive to control delay and infrequent, unexpected interactions with the environment. Solutions in the form of IL and MRL are presented that have the potential to reduce control delay and the risk of hardware strain.

Samenvatting

Reinforcement Learning op autonome mensachtige robots

De verwachting bestaat dat in de nabije toekomst de assistentie van dienstverlenende robots in huishoudens, gezondheidszorg en andere arbeidsintensieve omgevingen steeds belangrijker wordt. Hoewel fabrieksrobots reeds op grote schaal worden ingezet in productiefaciliteiten over de hele wereld, zijn dienstverlenende robots betrekkelijk nieuw en vooralsnog alleen in staat om simpele taken uit te voeren zoals stofzuigen en grasmaaien. Het belangrijkste verschil ligt in de omgeving: waar fabrieken gestructureerd en voorspelbaar zijn, is een huishoudelijke omgeving typisch uniek en aan verandering onderhevig. Om succesvol te kunnen zijn moeten dienstverlenende robots veelzijdig zijn en in staat zijn om nieuwe, opkomende taken te vervullen in verscheidene omgevingen. Deze veelzijdigheid aan taken en omgevingen is niet vooraf te voorzien en te testen op het moment dat de robot wordt geproduceerd, waardoor het handmatig programmeren van zulke robots problematisch is. Robots die zelf *leren* taken te vervullen door interactie met de echte wereld vormen een aantrekkelijk alternatief.

Reinforcement Learning (RL) is een generiek paradigma voor machinaal leren dat reeds voor een breed scala aan problemen is ingezet. Door interactie met de omgeving kan een RL systeem autonoom leren om taken op te lossen. Dit gebeurt door continu zijn gedrag te verbeteren op basis van enkel grofmazige terugkoppeling: gewenst gedrag wordt versterkt door positieve beloningen, ongewenst gedrag wordt bestraft door negatieve beloningen. Door zijn generieke formulering en vermogen om van echte ervaringen te leren is RL een veelbelovende kandidaat om lerende vaardigheden aan dienstverlenende robots toe te voegen.

Helaas is er relatief weinig bekend over hoe RL succesvol kan worden toegepast op *echte* robots. Het doel van dit proefschrift is om de moeilijkheden te identificeren en aan te pakken op het gebied van hardware-ontwerp, software-ontwerp en RL theorie, welke op dit moment het toepassen van RL op echte, autonome dienstverlenende robots verhinderen. Hiertoe zullen RL technieken worden geselecteerd uit de literatuur die geschikt worden geacht voor ons specifieke doel, op basis van hun theoretische eigenschappen en bestaande simulatieresultaten. De eisen die deze RL technieken stellen aan de hardware en software worden eerst geïdentificeerd en vervolgens gerealiseerd in een prototype: de tweevoetige, lopende robot Leo. De centrale leertaak in dit proefschrift is de taak van het leren lopen voor een tweevoetige robot, een taak die zowel uitdagend als interessant is om met RL op te lossen.

Deze thesis begint met een motivatie van de keuze voor Temporal Difference (TD) leren, gevolgd door een uitleg van additionele technieken zoals functie-approximatie door tile coding, die het specifiek toepasbaar maken voor robotische taken. Daarnaast wordt aangetoond dat vertraging in de regellus, oftewel de tijdvertraging tussen het meten van de toestand en de actuatie, waarschijnlijk optreedt bij echte robots en dat het een sterk negatieve invloed kan hebben op de prestaties van het leren. Een nieuw TD leeralgoritme wordt geïntroduceerd – dSARSA(λ) – dat beter kan presteren dan zijn klassieke tegenhanger SARSA(λ) door de vertraging in de regellus mee te nemen in de leerstappen zonder computationele complexiteit toe te voegen.

Nadat de eisen aan hardware en software zijn afgeleid die voortvloeien uit de geselecteerde RL technieken, wordt het ontwerp van de tweevoetige, lopende robot Leo gepresenteerd. Dit prototype dient als RL onderzoeksplatform voor een aantal experimenten in de thesis en wordt vergezeld van een realistische simulatie-omgeving. Experimenten uitgevoerd in simulatie en op de echte robot laten zien dat het mogelijk is om TD leren in te zetten om niet-triviale taken in te leren, zoals lopen (in ongeveer 5 uur) en het plaatsen van een voet op een traptrede (in ongeveer 15 minuten), real-time, op embedded hardware, ondanks de hoogdimensionale toestand-actie-ruimte van deze taken. Hoewel het prototype aan de meeste afgeleide eisen voldeed, schoot de robuustheid tekort bij de confrontatie met het trial-and-error karakter van TD leren – een reparatie was gemiddeld elke 30 minuten nodig – wat verhinderde dat de experimenten op de echte robot herhaald konden worden tot aan statistische significantie. De experimenten legden een aantal praktische complicaties bloot, die in het resterende deel van de thesis worden aangepakt.

De effecten op het leerproces van grote en infrequente verstoringen, zoals ongelijkheden in de vloer en onverwachte interacties met de omgeving zoals een duw, zijn relatief onbekend. Deze thesis presenteert simulatieresultaten van een simpel loopmodel dat laat zien dat grote en infrequente verstoringen van de daadwerkelijke systeemtoestand, bijvoorbeeld door de robot te duwen, een dramatisch effect hebben op de leersnelheid en systeemprestaties. Aan de andere kant hebben uitschieters in de bemonsteringsperiode van het systeem en in de sensorwaardes een verwaarloosbaar effect op het leerproces. Een methode wordt gepresenteerd die de leerprestaties herstelt door de uitschieters uit te sluiten van het leerproces.

Om het deel van de vertraging in de regellus dat veroorzaakt wordt door de leerregelaar te verkleinen, stelt deze thesis voor om Independent Learners (IL) te gebruiken voor de afzonderlijke actuatoren van een robot, om zo een architectuur te creëren waarin individuele, onafhankelijke leersystemen elk een enkele actuator aansturen. Deze gedecentraliseerde aanpak verkleint de rekentijd benodigd voor het beslissingsproces van het leersysteem en daarmee ook de vertraging in de regellus.

Om de belasting van de hardware door TD leren te verminderen, stelt deze thesis een oplossing voor gebaseerd op Modular Reinforcement Learning (MRL) waarmee de robot grof maar snel leert over het risico van zijn gedrag. Dit vermindert de cumulatieve blootstelling aan risico voor de robot tijdens het leren van de daadwerkelijke taakoplossing.

Samenvattend laat deze thesis zien dat TD leren potentie heeft om echte robots autonoom niet-triviale taken te laten leren. De belangrijkste complicatie is dat TD leren erg robuuste hardware nodig heeft door zijn trial-and-error karakter. Verder is het gevoelig voor vertraging in de regellus en voor infrequente, onverwachte interacties met de omgeving. Oplossingen in de vorm van IL en MRL worden gepresenteerd die de potentie hebben om de vertraging in de regellus en het risico op belasting van de hardware te verminderen.

Dankwoord

Dit proefschrift betekent het einde van een roerige tijd, met flinke contrasten door de onderzoeksjaren heen. Na een aanloop van ambitieuze plannen en initiatieven zorgde het werken met experimentele robothardware voor langdurige perioden van tegenslag. Hoewel ik lang niet heb kunnen afmaken wat ik oorspronkelijk wilde, geeft het eindresultaat toch enorme voldoening. Na veel ploeteren, nadenken, repareren en verbeteren heb ik mijn doel kunnen bereiken: een lerende robot.

Ik heb daarbij uiteraard van veel kanten hulp gehad, in de eerste plaats van mijn promotoren en copromotor. Pieter, je hebt een grote invloed gehad op mijn jaren aan de TU en ik heb een hoop aan je te danken. Als afstudeerbegeleider en vervolgens als promotor heb je me altijd in alle vrijheid onderzoek laten doen en groot vertrouwen getoond in wat ik deed. Ook deelden we dezelfde langetermijndoelen en konden we open en inspirerend overleggen over de te varen koers. Mijn fantastische stagetijd in Japan had je snel voor me geregeld en ook onze robotdemonstratie in de Ridderzaal op het Binnenhof zal ik nooit vergeten. Robert, jij hebt mijn onderzoek en proefschrift naar een hoger niveau getild. Dankzij jou heb ik mijn onderzoek in een breder (theoretisch) perspectief kunnen plaatsen. Door je zeer scherpe oog voor detail heb je veel onvolkomenheden uit mijn proefschrift gehaald waar ik inmiddels (soms al jaren) overheen las. Martijn, je passie om een goed lopend robotlab op te starten en te laten groeien en bloeien heeft zijn vruchten afgeworpen. In het DBL heb ik ongestoord onderzoek kunnen doen tussen de mooiste machines, met alles binnen handbereik: apparatuur, onderdelen, enthousiaste mensen, eten, bier.. ik heb me er erg op mijn gemak gevoeld. Ook heb je mijn wetenschappelijke schrijfstijl enorm vooruit geholpen. Bedankt voor je dagelijkse begeleiding! Ook wil ik Frans bedanken voor zijn brede kijk en inspirerende inbreng in mijn onderzoek tijdens de opstartperiode. Als laatste wil ik Lucian, Gabriel en Wouter bedanken voor hun hulp en samenwerking in het onderzoek. Wouter, ik vond het erg fijn om met je te kunnen sparren en samen mooie code te schrijven!

Voor de ontwikkeling van de robot ben ik veel dank verschuldigd aan Guillaume, die een belangrijk deel van het ontwerp heeft gemaakt, aan Jan van Frankenhuyzen, die naast adviseren ook samen met mij heeft staan zweten boven de robot toen deze elke 10 minuten gerepareerd moest worden, en aan Ad, John, Dries en Guus die hebben geholpen met het maken van de mechanische en elektrische hardware.

Natuurlijk wil ik de mensen in het lab en in de vakgroep bedanken voor de samenwerking, leuke tijden, interessante discussies en mooie avonden. Daan, Tomas, Daniël, Freerk, Steve, Tim, Gijs, Eelko, Cor, Oytun, Berk, Maja, Xin, Wietse, Boris, Sander, Anne, Thom en de rest: ik heb veel van jullie geleerd en veel met jullie kunnen lachen. Guillaume, behalve je hulp aan de robot zal ik natuurlijk ook onze woensdagavonden nooit vergeten! Ook heb ik met veel plezier samengewerkt met de afstudeerders Sebastiaan, Jan-Willem, Maarten, Thijs, Bart en Merel. Ik vond het begeleiden van jullie een van de leukste aspecten van mijn promotie en wil jullie bedanken voor jullie inzet en resultaten, die een significante plek hebben gekregen in dit proefschrift.

Verder wil ik mijn collega's bij S[&]T en TASS bedanken voor hun begrip en de geboden ruimte, waardoor ik het afgelopen jaar de laatste loodjes van mijn promotie heb kunnen combineren met een leuke fulltime baan die me nieuwe energie heeft gegeven.

Tot slot wil ik vrienden en familie bedanken – in het bijzonder pa, ma, zus en natuurlijk mijn lieve Marlous – voor het geduld dat ze de afgelopen jaren hebben opgebracht, als ik tijden niks van me liet horen of weer eens met een half oor zat te luisteren terwijl ik in gedachten robotproblemen aan het oplossen was. Het is nu eindelijk echt af!

Erik

Curriculum Vitae

January 30, 1981
Born in Puttershoek, The Netherlands.

1992-1998
Secondary school (‘gymnasium’) at Sint-Montfortcollege, Rotterdam, The Netherlands.

1998-2006
M.Sc. Applied Physics, Delft University of Technology (cum laude). The final project was conducted in the Quantitative Imaging Group at Applied Physics on the topic of Reinforcement Learning for wheeled and walking robots. In 2005, an internship was done at NEC, Kawasaki, Japan, implementing Simultaneous Localization and Mapping on the PaPeRo wheeled personal robot.

2006-2012
Ph.D. at the Delft Biorobotics Lab of the faculty of Mechanical, Maritime and Materials Engineering, department of BioMechanical Engineering, Delft University of Technology. The thesis project with title “Reinforcement Learning on autonomous humanoid robots” involved the application of Reinforcement Learning to robotic tasks in simulation and on real hardware, supported by the development of a dedicated bipedal walking robot.