Sample-Efficient Reinforcement Learning for Walking Robots

B. Vennemann

Master of Science Thesis

TU Delft Delft University of Technology

Delft Robotics Institute

Sample-Efficient Reinforcement Learning for Walking Robots

Master of Science Thesis

For the degree of Master of Science in Mechanical Engineering at Delft University of Technology

B. Vennemann

September 9, 2013

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
BioMechanical Engineering (bme)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Sample-Efficient Reinforcement Learning for Walking Robots

by

B. Vennemann

in partial fulfillment of the requirements for the degree of

Master of Science Mechanical Engineering

Dated: September 9, 2013

Supervisor(s):

Dr. W. Caarls

Prof.dr. R. Babuška

Reader(s):

Prof.dr.ir. P.P. Jonker

Dr. H. Vallery

Abstract

By learning to walk, robots should be able to traverse many types of terrains. An important learning paradigm for robots is *Reinforcement Learning* (RL). Learning to walk through RL with real robots remains a difficult challenge however. To meet this challenge, a robot called LEO has been developed in the Delft BioRobotics Lab. LEO is a 2D bipedal robot built specifically to learn to walk through RL. Unfortunately, when learning with Sarsa(λ) the robot breaks down before it has learned a successful gait. A possible solution for this is to minimize the number of interactions with the environment (samples) needed to learn a satisfactory policy. A promising technique to reduce sample complexity in RL is to re-use samples instead of discarding them after one update. One of the contribution of this thesis is providing a theoretical comparison of sample re-use techniques in the form of a novel unified framework. With the help of the framework, *Experience Replay* (ER) is selected to be used for evaluation and analysis of sample re-use on walking robots. Empirical comparison of ER with Sarsa(λ) is done with three benchmark problems: simulations of the inverted pendulum, the simplest walker, and LEO. On initial experiments we observed slow and unpredictable learning with ER on the walking problems. We show that this is mainly caused by two issues. The first issue involves failing back-propagation due to optimism in the face of uncertainty. To deal with this, we develop a new algorithm called ER-σ which makes the attitude towards uncertainty a function of the state instead initialization of the value function. The second issue is concerning local maxima emerging in the value function due to self effecting states. For this, we propose a residual gradient variant of ER. We find that the new algorithms perform well on the walking problems. In particular, (residual) ER-σ gives very encouraging results when compared with Sarsa(λ) and vanilla-ER. From the results, we can see that the attitude towards uncertainty during replay is of particular importance for walking problems. We conclude that while ER is a promising technique, it gives no guarantee on good learning performance. We showed that by exploiting the available data and knowledge of the representation, the result of ER can significantly be increased.

Contents

List of Symbols

α	learning rate
a	action
a'	next action
A	displacement vector
$\mathcal{A}$	action space
γ	discount factor
δ	temporal difference error
d	distance measure
$\mathcal{D}$	sample database
ϵ	stochastic exploration modifier
e	eligibility trace
E	number of policy iteration steps
f	state transition function
F	approximation mapping
g	gravitational acceleration [m/s^2]
θ	parameter vector
I	moment of intertia [kgm^2]
J	error function
k	time step
K	number of updates
l	length
L	sample batch size
λ	(eligibility) trace decay factor
μ	slope angle
m	mass [kg]
$\hat{M}$	estimated model
N	number of tilings
π	policy
ρ	reward function
Q	action-value function
Q^π	action-value function under policy π
Q^*	optimal action-value function
$\hat{Q}$	approximated action-value function
$\mathcal{Q}$	set of all possible state-action values

r	reward
$\hat{r}$	predicted reward
R	return
σ	state safeness function
s	state
s'	next state
$\mathcal{S}$	state space
T	target value
T^π	Bellman evaluation operator
T^*	Bellman optimality operator
$\widehat{T}$	estimated Bellman operator
V	state-value function
ϕ	basis function vector
φ	actor parameter vector
w	parameter vector of W
W	visit function

Chapter 1

Introduction

In the past, robots have successfully been deployed in industrial environments. Manufacturing robots are widely used and are indispensable in many production lines. Service robots have not yet achieved this status but a large world wide growth in the number of these robots is expected (IFR Statistical Department, 2010; WHO, 2007). Service robots are semi- or fully autonomous robots which perform services to humans excluding manufacturing operations. Examples of service robots are rescue and security robots, cleaning, entertainment or medical robots. The main difference between manufacturing robots and service robots is that the former usually perform a limited variety of tasks in a well known, controlled environment, whereas service robots are ideally able to perform versatile manipulation in a large diversity of environments. This large variety of environments is one of the key challenges and the main reason that service robots have not entered our daily lives yet. When a robot is not familiar with the type of environment it is dealing with, tasks like locomotion and navigation become much more challenging. In addition to being unknown, the surroundings of service robots are likely to change. So in order to let these robots function autonomously, they have to *learn* how to deal with an environment. Letting the robot interact with its environment and learn by itself, would make the robot more versatile and autonomous than being programmed by experts manually.

1.1 Learning walking robots

When dealing with a very diverse and changing terrain, walking becomes an attractive alternative to wheels. As can be seen in nature, humans can traverse a much larger variety of terrains than e.g. cars. In the past, numerous pre-programmed walking robots have been developed (Knight et al., 2002). They are developed for very specific types of terrain however and are generally poor at handling unknown features. By learning to walk, robots should be able to traverse many types of terrains. An important learning paradigm for robots is *Reinforcement Learning* (RL) (Sutton and Barto, 1998; Kober and Peters, 2012).

1.1.1 Reinforcement learning

RL can solve complex problems without requiring any prior knowledge on how to solve the problem or by making any restricting assumptions about the environment the learner is in. This is done by giving rewards for the actions it takes. By remembering these rewards, the RL algorithm can devise a strategy which will maximize the total reward.

Because of its lack of assumptions about the task and the environment, RL has been successfully applied to a wide set of problems ranging from games to control and economics (Tesauro, 1995; Kaelbling et al., 1996; Moody and Saffell, 2001). Successful application of RL in robotic motor control has been done several times. For instance by Vijayakumar et al. (2003) and Peters and Schaal (2006). RL has been used on walking robots to synthesize and optimize gaits as well. Ogino et al. (2003), Morimoto et al. (2005) and Cherubini et al. (2009) are some of the most recent examples. However, the number of successful attempts on real robots and without the use of prior-knowledge remains very small (Kober and Peters, 2012). Learning to walk with real robots remains a difficult challenge because of the inherent high dimensionality of robots, time variant dynamics, hardware wear, limited computational and memory capabilities. So as a consequence, relatively little is known about the application of RL on real, high dimensional walking robots with on-board computing.

In Chapter 2 we will give a more formal and elaborate introduction to RL.

1.1.2 LEO

To meet these challenges, a robot called LEO has been developed in the Delft BioRobotics Lab (Schuitema et al., 2010). LEO is a 2D bipedal robot specifically built for learning to walk. 2D means that it cannot fall sideways, only backwards and forwards. This is realized by a boom connected to the hip of the robot, see Figure 1.1. Additionally, the boom lets the robot walk in circles and provides the robot with power. The choice for a bipedal walking robot is twofold. Firstly, it is a complex and challenging task. Secondly, there exists extensive experience on bipedal walking robots from previous work in the Delft BioRobotics Lab (Collins et al., 2005; Hobbelen et al., 2008).

LEO is equipped with seven motors (in the hips, knees, ankles and shoulder). Furthermore, all the RL computing is done on-board. After a fall it can stand-up autonomously using the motor in its shoulder. 3 of the 7 actuators are controlled by the learning algorithm. Namely, the two hip actuators and one of the knee actuators.

The hardware has been designed to fulfill the requirements needed to realize reinforcement learning on robots in real-time and on embedded hardware. This primarily means that the robot is robust to the trial and error nature of RL, its behavior is predictable, and all the on-board computing is quick enough to keep up with the dynamics of the environment. The requirements of predictability and computational time have been met to a large extent (Schuitema et al., 2010). Robustness of the hardware of LEO remains a big obstacle to successfully learning a gait however.

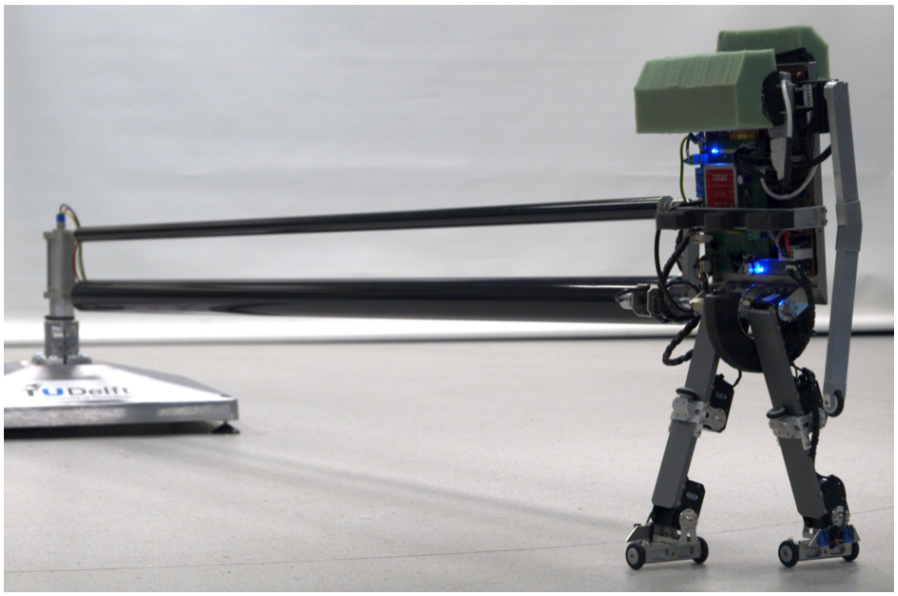


Figure 1.1: The robot LEO.

1.2 Problem statement

In recent attempts to let LEO learn to walk by Schuitema (2012), a Sarsa(λ) algorithm (see Section 2.6.3) with tile coding function approximation (see Section 2.7.2) was used. Schuitema succeeded to let LEO learn from scratch in a simulation but letting the real LEO learn to walk from scratch did not succeed however. While learning to walk typically takes several hours in simulation, the real robot starts to break down after 5 minutes. The hardware damage can be mainly attributed to applying different torques in a high frequency and falls occurring during the learning process. The need to replace all the broken parts makes learning from scratch on the real robot unfeasible with the current hardware and learning algorithm.

Schuitema (2012) *did* succeed in letting the real LEO learn by learning through demonstration. This was done by letting LEO observe a pre-programmed gait before learning by itself. By showing a pre-programmed gait we are supplying information on how to walk however. Consequently, this strategy amounts to using a lot of prior-knowledge. Since we are interested in walking in unknown environments, generally there is no such information available. Therefore, in order to have robots employable in unknown environments, we need to keep prior knowledge minimized.

Summarizing, the goal is to have LEO learn to walk without prior-knowledge. When learning from scratch however, the real LEO breaks down before it has learned a successful gait.

1.3 Research goal

In order to let the real LEO learn to walk , it needs to learn without breaking down and without implementing any prior-knowledge. For this, two possible solutions exist:

1. LEO does not break down before it learns to walk.

2. LEO learns to walk before it breaks down.

This thesis will focus on the second possible solution; letting LEO learn before it breaks down.

RL algorithms learn by interacting with the environment. Data of these interactions are collected in the form of *samples*. Usually, it takes numerous samples to learn a task successfully. However, each of these samples come with a cost: each interaction with the environment burdens the hardware of the robot. So in order to learn with a minimum amount of hardware damage, the number of samples it takes to learn needs to be minimized. When this is the case, the learning algorithm is said to have a low *sample complexity*.

Most classical RL algorithms only use samples once (Sutton and Barto, 1998). An important and promising way of reducing the number of samples needed is to re-use samples instead of discarding them after they have been used. Several of these methods have been developed showing promising results (Sutton, 1991; Lin, 1992; Ernst et al., 2005; Lagoudakis and Parr, 2003; Riedmiller, 2005; Lange et al., 2012). It is unclear however how these methods perform when compared with the current algorithm used on LEO, Sarsa(λ).

The goal of this thesis is to explore the properties and differences of sample re-use methods, identify how they scale to (high dimensional) walking robots and finally, evaluate whether they can reduce the sample complexity on LEO with respect to Sarsa(λ).

1.4 Approach

This thesis will start by introducing the more 'classical' RL-methods. In particular, the one used by Schuitema (2012) on LEO, Sarsa(λ). In addition, the most prominent sample re-use techniques will be discussed.

In order to investigate the properties of sample re-use techniques, this thesis presents a novel unified sample re-use framework. This framework allows us to easily identify and analyze the properties of the different techniques. With the analysis using the framework, a suitable sample re-use technique will be selected to be used throughout this thesis.

Evaluating the performance of the sample re-use technique is done on simulations of three benchmark problems: the inverted pendulum, the simplest walker and LEO. The choice for simulations instead of real-world setups is two-fold: real world problems often have extra confounding effects such as control delay, time invariant dynamics and noise. And secondly, learning on the real LEO has proven to be unpractical with Sarsa(λ) so performance can therefore not be compared on the real robot.

The problems include a non-walking problem (the inverted pendulum). Including a non-walking problem gives insight in whether results are caused by characteristics of a walking problem or whether they can be attributed to more general effects.

The simplest walker and LEO are both walking problems. By including two walking problems, it is made easier to draw conclusions on using sample re-use for walking problems. If only *one* walking problem would be included, it would be difficult determine whether results can be attributed to characteristics of walking or whether they are problem specific.

Evaluation of the algorithms is done by assessing how quickly the algorithm learns on the benchmark problems. This is done by measuring the performance of the agent during learning. Additionally, since falling is an important source of hardware damage, the number of falls it takes to learn will be evaluated on the walking problems.

Additionally, analysis of the results will be done on a 2D grid world. This is a non-continuous and low dimensional problem. This allows us to clearly visualize processes which are otherwise hard to find.

1.5 Contribution

The main contribution of this thesis is three-fold:

1. The derivation and discussion of a unified framework for samples re-use, clarifying the properties and differences of sample re-use techniques.

2. An analysis of issues rising for the selected sample re-use technique when applied on walking problems.

3. The derivation and discussion of novel algorithms dealing with these issues.

1.6 Thesis outline

The remainder of this thesis is structured as follows.

Chapter 2 gives a formal introduction to RL by giving the theoretical preliminaries for the remainder of this thesis. The focus will be on 'classical', model-free RL methods including Sarsa(λ).

Chapter 3 introduces the most prominent sample re-use techniques. Furthermore, a theoretical framework will be presented which allows greater insight to the theoretical differences of the methods. One of these sample re-use methods will be selected to be used for analysis.

Chapter 4 compares the performance of the sample re-use method with Sarsa(λ) on the three problems and discuss the results.

Chapter 5 analyzes issues arising from using the sample re-use methods with help of a grid world. Along with the analysis, solutions will be proposed for these issues.

Chapter 6 re-evaluates the performance of the new algorithms with respect to Sarsa(λ) on the same problems as chapter 4 and discusses the results.

Chapter 7 presents a summary and conclusions based on the previous chapters. And finally, future research directions will be proposed.

Chapter 2

Reinforcement Learning

This Chapter introduces the theoretical preliminaries of RL that are used in this thesis. The focus is on 'classical', model-free RL methods and Sarsa(λ), the technique which was used by Schuitema (2012) on the walking robot LEO. For a more general introduction to the field, the reader is referred to (Sutton and Barto, 1998) and (Bertsekas, 2007).

2.1 Introduction

Formally, RL consists of several elements with its own role in the learning process. The learner or *agent* can interact with its environment by taking actions. Each action results in a new state and a reward given by the *reward function*. RL comprises algorithms with the goal to learn a behavior that maximizes the future reward. The way RL algorithms learn can be compared to learning by trial-and-error: actions are tried out during a trial, rewards or punishments are received for these actions, and the data collected during this interaction is used to learn. During this process, the agent will learn maximize the cumulative reward by storing the expected future reward of each state-action pair. In RL, taking actions which will yield the highest expected reward is called *exploiting*. In addition to this, the agent might try new actions, this is called *exploring*. By doing so, a better strategy may be found in terms of the cumulative reward.

2.2 The Agent and the Environment

The *environment* is defined as the system in which the learning takes place, e.g. a robot with its surroundings or a simulation of it.

Figure 2.1 depicts the relation of the agent to the environment. Each time step k, an action a_k is taken by the agent resulting in a new state s_{k+1}. This action can e.g. be a voltage on a DC motors of a walking robot to change the position of the legs. The agent then receives a certain scalar reward r_{k+1}, e.g. a positive reward if the new position of the legs is good in terms of the end goal. The exact value of this reward is determined by the reward function, see Section 2.4.

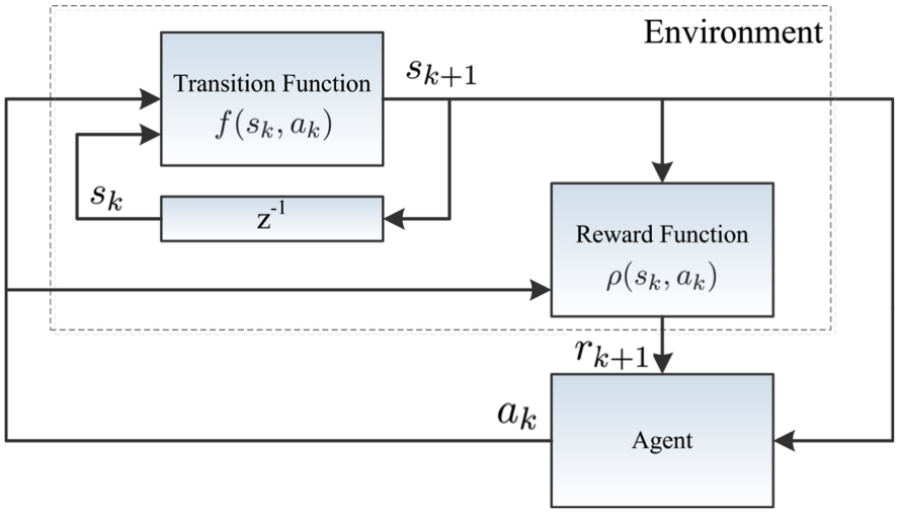


Figure 2.1: The agent and the environment. At a certain state s_k, the agent chooses an action a_k. Because the Markov property holds, the environment returns a new state s_{k+1} through the transition function and a reward r_{k+1} through the reward function. This information is used to improve the agent.

2.3 The Markov Property

In the deterministic case, the environment can be described by a mapping from state s_k and action a_k to new state s_{k+1} and reward r_{k+1}. In this form, the environment can be modeled by two transition functions: The state transition function $s_{k+1} = f(s_k, a_k)$, and the reward function $r_{k+1} = \rho(s_k, a_k)$, see Figure 2.1. Together, they form the environment as the mapping: $s_k, a_k \rightarrow s_{k+1}, r_{k+1}$. In the stochastic case, the mapping from state to new state are defined by probability distributions. The combination of the current state, current action, next state and reward is called a transition sample: $(s_k, a_k, s_{k+1}, r_{k+1})$.

In RL, the only required assumption about the environment is that the Markov Property holds: In either the deterministic or stochastic case, the transition is independent on states visited in the past and only dependent on the current state and the action. This means that the current state alone represents all relevant information. (Sutton and Barto, 1998). A decision process that satisfies this property is called a Markov Decision Process (MDP). Methods used in RL are under the assumption that the environment is an MDP. To fulfill the Markov Property on a robot, the state must typically not only contain the relevant positions of the bodies in space but also their velocities.

2.4 The Reward Function

The learning task is encoded in the reward function. This means that the reward function ultimately determines what the agent is to learn. For walking, a reward can for instance be given for every measure of distance a walking robot travels and punishments can be

given when the robot falls.

The ultimate goal of RL algorithms is to maximize the long-term reward. The long-term reward, also called *return*, at time step k can be defined as all the future rewards added together:

$$R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \ldots = \sum_{n=0}^{N} \gamma^n r_{k+n+1} \tag{2.1}$$

where γ is a discount factor to prevent the total long-term reward from going to infinity if $N = \infty$ (Sutton and Barto, 1998).

Because the agent learns through the reward function, synthesis of the reward function is an important aspect in the performance of RL algorithms. Very little is known however about synthesis of reward functions resulting in fast learning in the general case. In practice, researchers often resort to tuning the reward function until a satisfying performance rolls out. This process can take up a lot of time and in the case of LEO this is not different. For this reason, the reward function of LEO has been taken directly from (Schuitema, 2012) for this thesis.

2.5 The Value Function and the Policy

With discrete actions, the agent determines which action to take by evaluating each action by their respective expected return. In order to evaluate and update these expected returns, they need to be stored somehow. This is done in the *value function*. If a model of the environment is known, storing only the expected return for each state in the state space suffices, because for each action we know the next state the agent will end up in. In this case the value function is also called the state-value function and is denoted as $V(s)$. When this is not the case, an expected return for each action at each state (also called state-action pair) needs to be stored. It is then called the action-value function and is denoted as $Q(s, a)$. Since we are interested in keeping prior knowledge minimized, in this thesis only the model-less case will be considered. With Equation 2.1 in mind, the value functions are defined as:

$$V^\pi(s) = E^\pi \left\{ \sum_{n=0}^{N} \gamma^n r_{k+n+1} | s_k = s \right\} \tag{2.2}$$

and

$$Q^\pi(s, a) = E^\pi \left\{ \sum_{n=0}^{N} \gamma^n r_{k+n+1} | s_k = s, a_k = a \right\} \tag{2.3}$$

Where π is the policy, which can be seen as the control function determining the action the agent takes in a certain state ($\pi : s_k \rightarrow a_k$). $Q^\pi(s, a)$ is the expected total future reward of taking action a, being is state s while following the policy π.

Considering the state-action value function, the expected return of (s_k, a_k) is the current reward r_k plus the expected return of the future states and actions. By this definition, the expected return of the future states and actions is the expected return of

(s_{k+1}, a_{k+1}). Consequently, the expected return of (s_k, a_k) is: $Q^\pi(s_k, a_k) = E_\pi\{r_{k+1} + \gamma Q^\pi(s_{k+1}, a_{k+1}) | s_k = s, a_k = a\}$. This is called the *Bellman equation* for Q^π.

π can be determined in several ways, but the most common one is the ϵ-greedy policy. Following a greedy policy means taking the action with the highest Q-value. The ϵ-greedy policy also includes random actions with a probability of ϵ at every time step, hereby facilitating exploration. An optimal policy π^*, is a policy that maximized the expected return. Since the policy is derived from the value function it has a corresponding optimal action-value function satisfying the *Bellman optimality equation*: $Q^*(s_k, a_k) = E_\pi\{r_{k+1} + \gamma \max_{a_{k+1}} Q^*(s_{k+1}, a_{k+1}) | s_k = s, a_k = a\}$.

2.6 Temporal Difference Learning

Temporal Difference Learning(TD) comprises algorithms that learn policies without model of the environment. Consequently, the expected return will be stored in a state-action value function rather than a state value function. Because of the lack of a known transition function, learning can only be done from actual interactions with the environment.

Because samples can only be gathered from interactions, each episode in TD algorithms make a trajectory of samples through the state-space. An episode is one trial in which the agent gathers samples by following the policy. Usually, at the start of each episode the state is initialized following from some predefined distribution of initial conditions. The episode is usually terminated when the goal has been reached or when some terminal conditions have been met. Typically for walking, this is after a set amount of time steps or when the robot falls down.

In TD, learning takes place by using samples to make updates on the value function $Q(s, a)$. Learning ends when satisfactory behavior has been reached or when the value function has converged. Two major algorithms can be distinguished, namely Sarsa and Q-learning. They will be discussed in the next sections along with eligibility traces, an important technique for speeding up convergence yielding Sarsa(λ). The advantage of these techniques is that they are widely studied and come with some convergence guarantees (Sutton and Barto, 1998; Melo et al., 2008).

2.6.1 Sarsa

Sarsa (Rummery and Niranjan, 1994) is an on-policy TD algorithm. This means that it learns a Q-function for the policy the agent is following. This can make the policies of Sarsa more safe in comparison with off-policy TD and makes it in particularly suitable for non-stationary environments (Sutton and Barto, 1998; Otterlo and Wiering, 2012). The update rule is the following:

$$Q_{k+1}(s_k, a_k) = Q_k(s_k, a_k) + \alpha \left(r_{k+1} + \gamma Q_k(s_{k+1}, a_{k+1}) - Q_k(s_k, a_k)\right) \tag{2.4}$$

Here, α is the learning rate, which is a parameter in (0,1], and a_{k+1} is determined following the current policy. This rule can be seen as a gradient descent step of $Q(s_k, a_k)$

towards $r_{k+1} + \gamma Q(s_{k+1}, a_{k+1})$, with step size α. The Sarsa algorithm with ϵ-greedy action selection in pseudo-code is shown as Algorithm 1.

Algorithm 1: Sarsa

Input : discount factor γ, learning rate α, exploration rate ϵ

1 initialize Q arbitrarily;
2 **for** *each episode* **do**
3 $k \leftarrow 0$;
4 initialize s_k,a_k;
5 **for** *every timestep* k **do**
6 take action a_k and observe s_{k+1} and r_{k+1};
7 $a_{k+1} \leftarrow \begin{cases} \arg\max_a Q_k(s_k, a) & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$
8 $Q_{k+1}(s_k, a_k) = Q_k(s_k, a_k) + \alpha \left(r_k + \gamma Q_k(s_{k+1}, a_{k+1}) - Q_k(s_k, a_k)\right)$;
9 **end**
10 **end**

Output: Q

2.6.2 Q-learning

Q-learning (Watkins and Dayan, 1992) is one of the most popular RL methods (Otterlo and Wiering, 2012). In contrast to Sarsa, Q-learning is off-policy TD. This means that instead of learning a Q-function for the current policy, it aims at learning Q-values corresponding to the optimal policy π^*. This can be seen in the update rule:

$$Q_{k+1}(s_k, a_k) = Q_k(s_k, a_k) + \alpha(r_{k+1} + \gamma \max_a Q(s_{k+1}, a) - Q(s_k, a_k)) \tag{2.5}$$

The algorithm of Q-learning is the same as Sarsa with the exception of line 8, which is replaced by Equation 2.5.

2.6.3 Eligibility Traces

When receiving a reward with Sarsa and Q-learning, only the single state-action pair that immediately led to certain reward is updated, while in reality the whole past trajectory is responsible for reaching the point of reward (Sutton and Barto, 1998). The idea of eligibility traces is to include, to a certain degree, the past state-action pairs which were visited in the current trajectory in the update. Because the further the visited state-action pairs are in the past the less responsible for a given reward they are, they are discounted exponentially with a so called trace decay factor λ.

To implement eligibility traces, an extra variable called the eligibility trace $e_k(s) \in \mathbb{R}^+$ is associated with each state, which decays at each time step with a factor $\gamma\lambda$. The eligibility trace in the *replacing traces* variant (Singh and Sutton, 1996) is defined as:

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1} & \text{for } s \neq s_k \\ 1 & \text{for } s = s_k \end{cases} \tag{2.6}$$

Eligibility traces can be used with the Sarsa and Q-learning algorithms. These algorithms are then called Sarsa(λ) and Q(λ) respectively. Algorithm 2 shows the Sarsa(λ) algorithm with ϵ-greedy action selection in pseudo-code.

Algorithm 2: Sarsa(λ)

Input : discount factor γ, learning rate α, exploration rate ϵ

initialize Q arbitrarily;
for *each episode* **do**
  $k \leftarrow 0$;
  initialize s_k,a_k;
  $e \leftarrow \mathbf{0}$;
  for *every time step* k **do**
    take action a_k and observe s_{k+1} and r_{k+1};
    $a_{k+1} \leftarrow \begin{cases} \arg\max_a Q_k(s_{k+1}, a) & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$
    $e(s_k, a_k) = 1$;
    $\delta = r_{k+1} + \gamma Q_k(s_{k+1}, a_{k+1}) - Q_k(s_k, a_k)$;
    $Q_{k+1} = Q_k + \alpha\delta e$;
    $e \leftarrow \lambda\gamma e$;
  end
end

Output: Q

2.7 Function Approximation

The simplest form of storing expected values in a value function is in tabular form. However, if the state-action space of the system is very large it becomes inconvenient to store every state-action value in tabular. One reason is that the amount of memory needed to store the data might become too high. But moreover, learning speed reduces quickly for an increasing state-action space because learning with a tabular value function would typically involve visiting every relevant state-action combination in the table more than once. Additionally, when dealing with continuous states or actions, a tabular representation is not even possible.

Instead of using a table, the value function can be approximated by a function with vector $\theta = [\theta_1, \theta_2, \ldots, \theta_n]^T$ as parameters: $\widehat{Q}(s, a; \theta)$. The function approximator can be seen as a mapping F from the vector θ in $\mathbb{R}^n$ to the space of the action-value function containing the state-action values: $F : \mathbb{R}^n \rightarrow (\mathcal{S}, \mathcal{A} \rightarrow \mathbb{R})$. F and θ together form the approximated value function $\widehat{Q}(s, a) = [F(\theta)](s, a)$.

An important feature of function approximation is that is causes generalization. This means that adapting an element of θ results in a change of Q-values in a region of the state-action space. This can range form a global to a local scale; a single element of θ can effect Q-values at every point in state-action space or only a small region of the state-action space. How generalization occurs depends on the architecture of the function approximation itself. In addition to handling continuous spaces, function approximation can lead to faster learning, since a learning update on $\widehat{Q}(s_k, a_k)$ influences Q-values in a region around (s_k, a_k). Therefore, not all relevant state-action pairs have to be visited.

2.7.1 Approximate Sarsa

When the approximated value function is a smooth differentiable function of θ, TD-learning with function approximation typically involves taking a gradient descent step from $\widehat{Q}(s, a)$ towards $r + \gamma\widehat{Q}(s', a')$. To do this, the parameters are updated with the following rule:

$$\theta_{k+1} = \theta_k + \alpha \left(Q_{k+1}(s_k, a_k) - \widehat{Q}_k(s_k, a_k) \right) \nabla_{\theta_k} \widehat{Q}(s_k, a_k) \tag{2.7}$$

This rule can be easily extended to Sarsa(λ). For a full derivation of the approximate Sarsa rule, the reader is directed to Sutton and Barto's book *Reinforcement Learning: An Introduction* (1998).

2.7.2 Linear in parameters function approximation

Several constructions are commonly being used for mapping F in RL. A common approach is the use of function approximation which is linear in parameters. Usually, this type of function approximation is simply called *linear function approximation*. Linear techniques depend on two vectors of equal size, a parameter vector θ and a feature vector ϕ which contains a set of basis functions. The value function is then approximated as:

$$\widehat{Q}(s, a) = \theta^T \phi(s, a) \tag{2.8}$$

Linear techniques have several advantages, the theoretical analysis of the resulting algorithms is simpler, which has led to several convergence guarantees (Bertsekas and Tsitsiklis, 1996; Bertsekas, 2007; van Hasselt, 2012). Additionally they are easier and faster to compute than non-linear methods (Adam et al., 2012). A drawback is that the selection of good features is crucial with linear techniques (van Hasselt, 2012).

Tile Coding is the function approximator used by Schuitema (2012) in the learning algorithm of LEO. This method is also known as CMAC (Albus, 1975). It is a linear, parameterized method and it has similarities with simple discretization of the state-action space. In tile coding, the state space is divided in a number of overlapping grids called *tilings*. Each tiling contain *tiles* which take the form of multi-dimensional hypercubes. Figure 2.2 shows how this looks in 3 dimensions. Each tile has a basis function with a value of 1 if (s, a) is inside the hypercube and 0 if it is outside. Each

tiling is displaced from the origin with a certain offset to create an even distribution over the state-action space. A good choice of the displacement vector A_d^k for each tiling k in each dimension d is (Miller et al., 1990):

$$A_k^d = r_d(k-1)(1+2(d-1))/K \tag{2.9}$$

If N is the number of tilings, each state-action pair has exactly N tiles which it is full member off. Each element in θ corresponds to a value of one tile. When a state-action pair is updated, the elements of θ corresponding to the activated tiles are updated. Because neighboring state-action pairs share tiles with each other, a generalizing effect over the state-action space is created .

Tile coding is commonly used because of several advantages. Firstly, it is relatively easy to implement. Secondly, tile coding can often be achieved with relatively large hypercubes, resulting in relatively low computational and memory requirements for high dimensional systems (Schuitema, 2012). A downside is that tile coding cannot discover patterns and irregularities that span large parts of the state-action space as well as other, non-linear methods. This can result in that some sample information might go un-used (Gauci and Stanley, 2008). Another downside is that because the membership of the tilings is binary, the generalization is not done smoothly but rather cascaded. Smooth tile membership with e.g. radial basis functions is an easy extension of tile coding making the function approximation continuous. Schuitema (2012) has indicated that this does not increase learning performance on LEO however, while computational complexity rises.

2.7.3 Non-Linear and memory based function approximation

This section will briefly introduce some other well known function approximation schemes.

Local Linear regression In Local Linear Regression (LLR) (Atkeson et al., 1997) is a non-parameterized method in which the prediction is computed by taking the average of, or fitting a linear model through, the K nearest neighbors of the query.

These methods have shown to be an efficient function approximator but can be quite sensitive to the definition of the distance function which determines the K nearest samples that are included in the regression (Wettschereck et al., 1997; Kober and Peters, 2012).

Regression trees Largely introduced by Breiman (2001), tree based methods are non-parameterized learning techniques which use decision trees to map observations in a tree-like structure to determine the value of a query.

Tree based methods are a particularly promising supervised learning technique as shown by Caruana and Niculescu-mizil (2006). In this study, regression trees and in particular methods using ensembles of regression trees, performed remarkably well as three of the best supervised learning techniques according to prediction accuracy over a set number of trials.

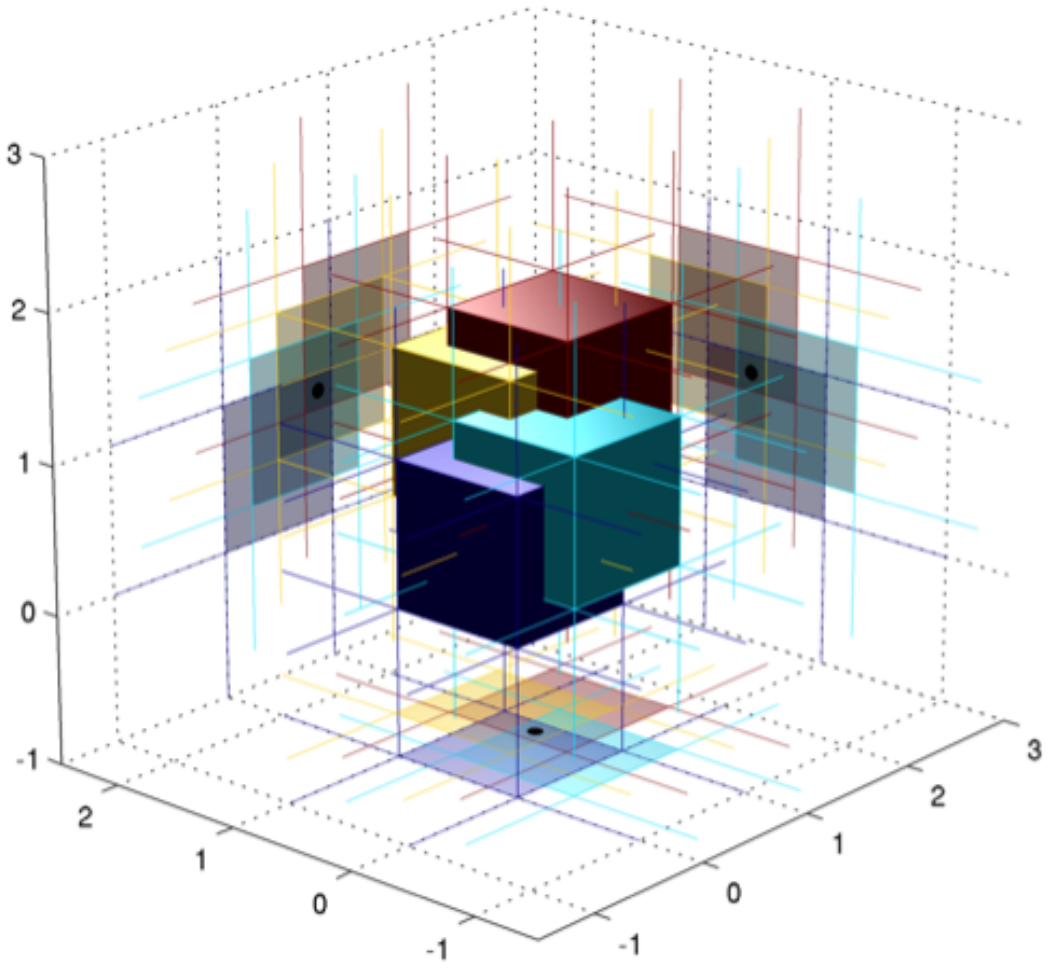


Figure 2.2: Visualization of activated tiles in 3 dimensions along with their projections on the 3 planes. In this example there are 4 tilings. Consequently, 4 cubes determine the value of the state-action combination.

Additionally, regression trees have shown some promising results when being combined with sample re-use (Ernst et al., 2005).

Artificial Neural Networks Artificial Neural Networks (ANNs) are quite commonly used in RL with Lin (1992) being among the first. Inspired by the biological neural network, ANNs are networks of mathematical models imitating the function of neurons. The main function of these networks is the ability to learn a complex relational function from a limited amount of labeled training data.

Perhaps the biggest advantages of using ANNs is their ability to handle high dimensional inputs and their excellent generalization properties which can potentially bring down sample complexity (Riedmiller, 2005; Coulom, 2002; Tesauro, 1995). In practice, because ANNs come with several drawbacks which can make them tricky to implement (Boyan and Moore, 1995), linear function approximation methods are often preferred to ANNs. Additionally, ill conditioning of the ANN can bring the performance down greatly. Consequently, tuning the number of layers, threshold functions and starting weights might be needed before a satisfactory result is reached.

As with regression trees, ANNs have shown some promising results when being combined with sample re-use (Riedmiller, 2005).

Chapter 3

Sample Re-use

An important technique if one wishes to achieve low sample complexity is sample re-use. It is therefore an important tool if we want to achieve sample efficient walking robots. In the past, numerous techniques have been developed capable re-using samples (Sutton, 1991; Lin, 1992; Ernst et al., 2005; Lagoudakis and Parr, 2003; Riedmiller, 2005; Lange et al., 2012), yielding promising results.

The theoretical differences of these techniques are not always clear however. Nor is it always clear under what circumstances and for what type of problems to use a certain method. In this chapter, the most important sample re-use techniques will be discussed and their theoretical differences will be investigated by presenting of a novel unified framework for sample re-use. With the properties found through the framework, a suitable sample re-use technique will be selected to be used throughout thesis.

3.1 Introduction

In the previous chapter we have seen learning methods which have the following learning paradigm: the agent interacts with the environment by taking an action in a certain state. This results in a new sample and with it, the value function is updated. After this, the sample is discarded, the agent takes a new action and the cycle starts over. In this thesis, we are interested in using the information gathered from interacting with the environment as efficiently as possible. To do this, samples can be stored for use later on, instead discarding them after a single update. It turns out that classical methods such as Sarsa, samples are often not really needed to gather more information about the environment, but are used to spread information through the state-action space (Lange et al., 2012). By storing samples, we can use their information as if they were new observations. We thus do not need new samples to back-propagate information, we only need new samples to gather information about the environment. This makes it is possible to improve the policy from various old samples, even if these samples were not part of a successful trajectory at the time.

Since we are interested in achieving a low sample complexity, the database of old samples is likely to be finite. For high dimensional systems, we even prefer an incomplete

dataset. Filling up the data set of samples might sound like a good idea, but for most problems in robotics, this would take too many samples. Because the given set of data is usually finite, the learner cannot be expected to always come up with an optimal policy. The objective of sample re-use techniques has therefore been changed from learning the optimal policy, to deriving the best possible policy for the given data (Lange et al., 2012). Consequently, sample efficient learning with sample re-use can be seen as finding the best possible policy for an incomplete dataset.

There exists several ways of re-using samples. In the following, the most prominent ones will be introduced. We will start by introducing *Experience Replay* (ER)(Lin, 1992; Adam et al., 2012). This method has the closest resemblance to the classical TD-methods of Sarsa and Q-learning. Secondly, *Batch RL* methods (Ernst et al., 2005; Riedmiller, 2005; Lagoudakis and Parr, 2003) will be addressed. This collection of techniques use various supervised learning techniques to fit a value function to the given data. And lastly, *model learning techniques* (Sutton, 1991; Hester and Stone, 2012). These methods use old samples to derive a model of the environment and learn from that model.

3.2 Experience Replay

ER re-uses old experience by simply applying the Sarsa or Q-learning update rule on old samples. ER was introduced by Lin (1992) and despite some considerable advantages, ER has been studied only sporadically ever since. Recently however, ER received some new attention yielding promising results (Kalyanakrishnan and Stone, 2007; Adam et al., 2012).

ER can be implemented in the following way: Every transition sample $(s_k, a_k, r_{k+1}, s_{k+1})$ is stored in a database. Once every L steps the old samples are re-used to improve the value function according to the underlying TD algorithm.

The selection of samples to be reused can be done in several ways: samples can be chosen separately at random or entire trajectories of samples can be replayed. In the former case, samples can be selected in either temporal or reversed temporal order. Another possibility is to use Prioritized Sweeping (Moore and Atkeson, 1993), a method also used in model learning techniques (see Section 3.4).

3.3 Batch RL

Formally, batch RL algorithms are off-line methods which first acquire and store a *batch* of experience and then solve the learning problem through supervised learning. Where ER can be seen as extension of Sarsa and Q-learning, batch RL does not necessarily rely on update rules such as Equation 2.7.

There are two major batch RL algorithms. Namely, fitted Q-iteration by Ernst et al. (2005) and LSPI by Lagoudakis and Parr (2003). In the following they will be discussed briefly.

3.3.1 Fitted Q-iteration

Fitted Q-iteration (FQI) calculates a target value T for every sample and uses a supervised learning technique to fit a value function on to these values. In fact, any supervised learning technique can be used to derive the value function. This is done by iterating over the following two steps:

1. For each sample $(s_k, a_k, r_{k+1}, s_{k+1})$ calculate a new target value $T^{i+1}_{s_k,a_k}$ according to:
$$T^{i+1}_{s_k,a_k} = r_{k+1} + \gamma \max_a \widehat{Q}_i(s_{k+1}, a) \tag{3.1}$$
and store (s_k, a_k, T^{i+1}) with all other (s, a, T)-combinations.

2. Use a supervised learning algorithm to train a value function on all $(s, a, T^{i+1}_{s,a})$ combinations found so far, with input the state-action combinations and output the target values. This results in the new approximated function $\widehat{Q}^{i+1}(s, a)$.

Originally step 2 was performed with regression trees. Two another notable examples of FQI are when step two is performed with ANNs, which has yielded Neural Fitted Q-iteration (Riedmiller, 2005) and with kernel regressors, yielding KADP (Ormoneit and Sen, 2002).

3.3.2 LSPI

Similar to FQI, LSPI builds a set of target values and fits a value function to it. With LSPI, this is done in one step using least squared regression. Since least squared regression is only possible with linear functions with respect to parameters (Busoniu et al., 2012), this method can only be used with linear function approximators such as tile coding.

The major difference of LSPI to other approximate RL methods is that LSPI determines an approximation of the state-action value function in one step and analytically, so no iterations are needed until a satisfactory convergence is reached. Therefore, the method has fewer parameters to tune such as a learning rate α which can cause oscillations, overshoot and divergence (Lagoudakis and Parr, 2003).

3.4 Model Learning

Samples can also be used to learn a model of the environment in the form of an estimated model $\hat{M} : (s, a) \rightarrow (\hat{r}, \hat{s}')$. This model can be used to update the policy or value function along with newly acquired experience with dynamic programming techniques, this is called *planning* (Sutton, 1991). Dynamic programming is the collection model-based RL algorithms (Sutton and Barto, 1998). With a model, we can predict what the effect of a certain action would be in terms of the next state and the returned reward. Consequently, no actual interaction with the environment is needed. Thus, with a learned model, the value function can be improved without needing to gather more samples.

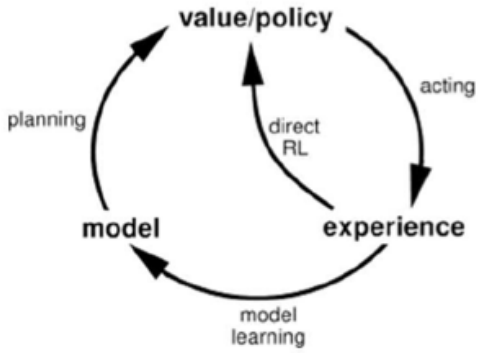


Figure 3.1: Schematic representation of Dyna. The policy choses an action at a certain state after which the environment returns a reward and the new state. This experience is used for direct RL (i.e. update the value function with for instance Sarsa) and to learn a model of the environment. This model can then be used improve the value function (planning).

Learning a model of the environment can be done by supervised learning techniques like ANNs or LLR (Hester and Stone, 2012). In order to update the policy or value-function without troublesome model errors, the model has to be learned sufficiently well. This can take numerous samples, well distributed over the state-action space.

The most widely and well-known model learning techniques is known as Dyna (Sutton, 1991). This algorithm uses experience both for direct RL to update the value function and for model learning. Figure 3.1 shows the Dyna architecture.

There are several known issues with model learning. Firstly, in reality the transitions might be very stochastic due to noise or disturbances increasing the number of samples needed to learn an accurate model. Secondly, the behavior of the system might be time variant. Thirdly, the model might contain discontinuities or non-Markov effects which makes representing an accurate model more difficult (Toussaint and Vijayakumar, 2005). And lastly, the computational cost can increase severely with respect to non-model learning techniques (Sutton, 1991).

3.5 A Unified Framework

In the previous section, the most prominent methods of re-using samples in the RL literature were introduced. The goal of all these methods is to seek an optimal value function with the information available. The theoretical differences between them is not entirely clear however. For instance, both ER and batch RL use previous samples without the use of a model. What then, is their the fundamental difference in value function improvement and what are the consequences of this? Furthermore, can one argue that patterns found in learned model can be found in the approximated value function as well? Additionally, it is not clear whether or not certain aspects of the above methods might be combined. A unified framework will give more insight on the general

way how to re-use samples and might give rise to new ways of doing so.

In this section, a unified framework of sample re-use will be presented in the form of a general algorithm. Within this algorithm, the methods introduced in the previous section are a special case. First, the differentiating aspects will be discussed. Next, the framework will be presented in the form of algorithms and a property table.

Several other frameworks exist in the literature. This framework is unique in the sense that is focuses on sample re-use. Other frameworks focus only on a specific sub-domain of RL such as multi-agent RL (Hu and Wellman, 1998) or are so general that they do not give any insights in how sample are actually used such as Generalized Policy Iteration by Sutton and Barto (1998). Geist and Pietquin (2013) group several RL methods in three ways of updating the parameter vector θ but do not specifically consider the use of old samples either.

3.5.1 Value function update

A first distinguishing aspect, is the way the samples are being used to improve the Q-function. They have in common however that they are all centered around the Bellman operator. It is their specific use of the Bellman operator which differs. This section will introduce the Bellman operator, derive a general notation for parameter updates and define two general ways of re-using samples to update the value function.

Thanks to the Markov property, RL algorithms can rely on the Bellman operator to derive a value function. The Bellman operator uses information from samples to make an estimate of the expected value corresponding to the state-action combination of that specific sample. When more information in the form of transitions arises, these estimates can get more accurate. So when more is information becomes available, estimates following from Bellman operators applied on old samples might become useful again. For the action-value function case, we can define the Bellman evaluation operator T^π as follows:

$$[T^\pi Q](s_k, a_k) = E\{\rho(s_{k+1}, a_k) + \gamma Q(s_{k+1}, a_{k+1})\} \tag{3.2}$$

Here, $T^\pi Q^\pi$ means taking the Bellman operator while following the current policy. Consequently, since the value function itself consists of expected values, it should satisfy: $Q^\pi = T^\pi Q^\pi$. The optimal action value function Q^* satisfies the Bellman optimality operator T^*:

$$[T^* Q](s_k, a_k) = E\{\rho(s_{k+1}, a_k) + \gamma \max_{a_{k+1} \in \mathcal{A}} Q(s_{k+1}, a_{k+1})\} \tag{3.3}$$

In exploration driven RL, the goal is to find the optimal action-value function Q^* from samples observed by interacting with the environment. Therefore, instead of exact computations of the Bellman operator, an estimated Bellman operator $\widehat{T}$ can be computed with a transition sample:

$$[\widehat{T}^* Q](s_k, a_k) = r_{k+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{k+1}, a) \tag{3.4}$$

Every transition sample leads to its own estimate of the expected value for a certain state-action combination. Since we are bootstrapping i.e. updating an estimate of the

value function based on other estimates (Sutton and Barto, 1998), $\widehat{T}^*Q(s_k, a_k)$ is merely an estimate. However, when more samples become available, the value of $Q(s', a')$ might become more accurate. Because of this, for a certain transition sample $\widehat{T}^*$ might move closer to T^* as learning progresses.

When using function approximation, there is no exact representation of the value function Q, but an approximation of it $\widehat{Q}$. So Equation 3.4 becomes:

$$[\widehat{T}^*\widehat{Q}](s_k, a_k) = r_{k+1} + \gamma \max_{a \in \mathcal{A}} \widehat{Q}(s_{k+1}, a) \tag{3.5}$$

With parametric value function representation, we can represent the value function as a mapping F from a parameter vector θ in $\mathbb{R}^n$ to the state-action space $\mathcal{Q}$. The approximated value function can then be expressed as:

$$\widehat{Q}(s, a) = [F(\theta)](s, a) \tag{3.6}$$

Substituting Equation 3.6 into 3.5 we get:

$$[\widehat{T}^*F(\theta)](s_k, a_k) = r + \gamma \max_{a \in \mathcal{A}} [F(\theta)](s_{k+1}, a) \tag{3.7}$$

In order to improve the value function, we must minimize the error between the expected values following from the current value function $\widehat{Q}(s, a)$, and the expected values from the transition samples $\widehat{T}^*\widehat{Q}(s, a)$ (or $\widehat{T}^\pi\widehat{Q}(s, a)$). To do this, we can define a cost function J expressing the error between the expected values from the transition samples and the current $\widehat{Q}$. That is:

$$\widehat{J}_{\mathcal{D}}(\theta) = \sum_{d=1}^{p} \left(\widehat{T}_d\widehat{Q} - \widehat{Q}(s_d, a_d)\right)^2 = \sum_{d=1}^{p} \left([\widehat{T}_d F(\theta)] - [F(\theta)](s_d, a_d)\right)^2 \tag{3.8}$$

Here, d represents the d-th transition sample (s_d, a_d, s'_d, r_d) in a set of p available transition samples $\mathcal{D} = \{(s_d, a_d, s'_d, r_d) | d = 1, \ldots, p\}$, and $\widehat{T}_d$ is the Bellman operator applied to that specific sample. By minimizing $\widehat{J}$ we can find a suitable new parameter vector θ_{k+1} for a given set of samples $\mathcal{D}$:

$$\theta \leftarrow \operatorname*{argmin}_{\theta} \widehat{J}_{\mathcal{D}}(\theta) \tag{3.9}$$

With ER and Dyna, $J_{\mathcal{D}}$ is minimized in the following way. A gradient descent step on the error of a single sample ($J_{\mathcal{D}} = J_d$) is taken with the current parameter vector θ_i, yielding a new parameter vector θ_{i+1}. With this, a new gradient descent step is taken with another sample to yield θ_{i+2}. This means that these methods minimize the error following from only one sample at the time. What follows is that the parameters are updated by iterating the rule:

$$\theta_{k+1} = \theta_k - \frac{1}{2}\alpha\nabla_{\theta_k}([\widehat{T}_d F(\theta_k)] - [F(\theta_k)](s_d, a_d))^2 \tag{3.10}$$

over separate samples in $\mathcal{D}$, where T can be either T^π or T^* for respectively Sarsa and Q-learning updates. When no samples are being re-used, the error is minimized over the current sample only i.e. $\mathcal{D} = \{(s_k, a_k, s_{k+1}, r_{k+1})\}$. What follows in this case is:

$$\theta_{k+1} = \theta_k - \frac{1}{2}\alpha\nabla_{\theta_k}([\widehat{T}_k F(\theta_k)] - [F(\theta_k)](s_k, a_k))^2 \tag{3.11}$$

This equation results in the well known TD update rules (Sutton and Barto, 1998). When sample re-use *is* being used in combination with TD methods, learning takes place by iterating equation 3.10 with samples arbitrarily taken from the sample set $\mathcal{D}$.

Since we are using a gradient descent step on $J_{\mathcal{D}} = J_d$, there is no regard for the error following from other samples in $\mathcal{D}$. As a result, for some function approximators an update in a certain region of the state-action space might have unpredictable or unwanted consequences in other regions of the state-action space (Gordon, 1995; Lange et al., 2012). This is known to cause slow and unpredictable learning with certain function approximators (Ernst et al., 2005; Riedmiller, 2005; Lange et al., 2012). The extend of this effect depends on the extend of what Gordon (1995) calls the 'exaggeration' of the function approximator. Mainly global function approximators suffer from a high amount of exaggeration (Gabel and Riedmiller, 2005). In particular, ANNs in combination with single sample updates have shown to require a lot of samples before convergence (Riedmiller, 2005).

By updating θ considering the cost function all samples simultaneously, different updates will not interfere with each other. Instead of improving single Q-values of state-action pairs, the value function can be updated so that it 'fits' to the target values of all samples at once. This is the characteristic way batch RL algorithm improve the value function. Now, instead of minimizing the error of each sample after another, batch RL minimizes the error of *all* samples simultaneously. In other words, $\theta \leftarrow \underset{\theta}{\operatorname{argmin}}\, \widehat{J}_{\mathcal{D}}(\theta)$ is executed with the entire set of samples in one *projection*. In other words, batch RL methods project the target values of all samples onto hypothesis space of the function approximator.

Batch RL can use any supervised learning technique to update the value function. Consequently, the value function can be a black box and does not need to be parametric (Ernst et al., 2005). This contrary to gradient descent methods which calculate gradients with respect to parameters. The disadvantage of parametric function approximation is that one has to select the approximation architecture a priori. This often requires using prior-knowledge or going through a difficult process involving guessing and tuning of parameters until good results are achieved.

Summarizing, two distinctive value function update methods can be distinguished:

1. Single sample updates, minimizing the cost function following from one sample, $\widehat{J}_d = \left(\widehat{T}_d F(\theta) - [F(\theta)](s_d, a_d)\right)^2$, iteratively with gradient descent on the parameters.

2. Projection updates, minimizing the cost function following from multiple samples: $\widehat{J}_{\mathcal{D}} = \sum_{d=1}^{p} \left(\widehat{T}_d F(\theta) - [\widehat{T}_d F(\theta)](s_d, a_d)\right)^2$, or in the case of a non-parametric representation: $\widehat{J}_{\mathcal{D}} = \sum_{d=1}^{p} \left(\widehat{T}_d \widehat{Q} - \widehat{Q}(s_d, a_d)\right)^2$. This is done by projecting the target values onto the hypothesis space of function approximation.

3.5.2 Value Iteration vs. Policy Iteration

An additional, although similar distinction can be made regarding the policy update. Namely, whether the improvement of the value function can be regarded as value iteration or policy iteration. With value iteration, updates on $Q(s_k, a_k)$ also directly influence the policy $\pi(s_k)$. Policy iteration methods separate updating of the value function and deriving the policy in separate steps. This means that $\widehat{T}^{\pi_k}$ is used to calculate Q_{k+1}, then a new policy π_{k+1} is derived from Q_{k+1}, after which it is used in $\widehat{T}^{\pi_{k+1}}$ to derive Q_{k+2}. In other words, policy iteration fixes the policy until a new $\widehat{Q}$ has been derived while with value iteration, any update on Q simultaneously changes π . ER and Dyna are value iteration since these method do not explicitly store a policy. With these methods the policy is always directly derived from the current value function. Batch RL is policy iteration because the Bellman operator is computed for all samples by fixing the policy.

This distinction can have consequences for sample efficiency. With model-based RL, it has been shown that value iteration often converges faster than policy iteration because it is usually inefficient to sweep over all the state-action combination before improving the policy (Sutton and Barto, 1998). Similarly, it might be inefficient to sweep over all the samples before improving the policy.

3.5.3 Sample Database

A second major distinction between sample re-use methods concerns the set of available transition samples. $\mathcal{D}$ can be formed in several ways. As we have seen, not re-using samples means that $\mathcal{D}$ only contains the current sample: $\mathcal{D} = \{(s_k, a_k, s_{s+1}, r_{k+1})\}$. When every previous transition sample is stored $\mathcal{D} = \{(s_d, a_d, s_{d+1}, r_{d+1}) | d = 1, \ldots, k\}$.

Many algorithm do not store all previous samples in $\mathcal{D}$ however. For instance, Lin (1992) only used the 100 most recent samples with ER. Using only recent samples for updates has some advantages. Recent samples are more likely to be on policy and according to (Sutton and Barto, 1998), off-policy bootstrapping combined with function approximation can lead to divergence. In non-stationary environments it may even be necessary to only use recent experiences since old sample might not give accurate information anymore. Kalyanakrishnan and Stone (2007) showed that in their experiments, best performance was always achieved by using *all* the training samples for both ER and FQI however.

Additionally, due to computational complexity of minimizing the squared error (see next section) of large databases, it might be beneficial to remove excess samples from the database. Especially in later training phases and with cyclic tasks like walking, a majority of the gathered samples might already be contained in the same or similar form. Improving the value function might be more effective per update if a pruned database with evenly distributed samples is maintained. This thesis does not go into this aspect however.

In addition to storing previous transition samples, supervised learning techniques can be used to train an approximate model $\hat{M}$ on the available data to generate samples. This is done by first constructing a training set from the database of transition samples and us-

ing a supervised learning algorithm to find a function $\hat{M} : \mathcal{S}, \mathcal{A} \rightarrow \mathcal{S}, \mathcal{R}$. Any supervised learning technique can be used to construct this function. With this model, an arbitrarily set $\mathcal{M}$ of n approximated samples can be generated : $\mathcal{M} = \{(s_i, a_i, \hat{s}'_i, \hat{r}_i) | f = 1, \ldots, n\}$. $\mathcal{D} \cup \mathcal{M}$ then represents the total set of samples available for updates.

3.5.4 Computational Complexity

In this section, the computational complexity of the sample re-use techniques will be discussed. In this discussion, the following case will be considered. The agent has gathered a number of n samples over some pre-defined amount of time. We are interested in the order of computational complexity of the sample re-use techniques if they were to learn a value function from these samples.

Firstly, on-line learning algorithms such as Sarsa and Q-learning only make one update for every experience. Therefore the number of updates made while encountering n samples is $\mathcal{O}(n)$. Considering ER, there is full freedom of choosing the amount of sample replays. If this number is called K, and when combined with direct learning, the complexity is $\mathcal{O}(n + K)$.

To learn a value function with FQI, E times a value function needs to be trained using i iterations. i is the number of iterations needed to train a certain supervised learning technique to n samples. And E represents the number of policy iteration sweeps, i.e. the number of times that the Bellman operator is evaluated for all samples. This yields a computational complexity of $\mathcal{O}(nEi)$. Since the policy iteration step is done analytically, LSPI doesn't need i iterations to train a value function. However, complexity of least squares regression is cubed to the number of basis functions f. This yields a complexity of $\mathcal{O}(nEf^3)$. The number of policy iteration sweeps E is the optimization horizon of the algorithm (Ernst et al., 2005): E iterations corresponds to an E-step optimization. In other words, a value function is derived for expected future rewards E steps into the future.

Model learning techniques consist of three separate processes: on-line learning, model learning and planning. As we have seen, on-line learning is in the order of $\mathcal{O}(n)$. If we are using the same supervised learning technique as with FQI to learn our model, i iterations are needed. The learning of $\hat{M}$ is thus of order $\mathcal{O}(ni)$. In the literature, planning is often done a set amount of K times. This yields $\mathcal{O}(K)$. The total sample complexity is $\mathcal{O}(n) + \mathcal{O}(ni) + \mathcal{O}(K)$. The maximum is dominant yielding $\mathcal{O}(ni + K)$.

From the above, several conclusions can be drawn. Firstly, because parameter K can be chosen arbitrarily, ER and model learning techniques allow greater control over the computational complexity. Because FQI is policy iteration, we are bound to $\mathcal{O}(nEi)$ if we wish to have an E-step optimization. With single sample updates we can achieve this without having to sweep over all samples e.g. with smart sample selection such as prioritized sweeping. Furthermore, for high dimensional problems, the number of basis functions f is likely to be high. Since the complexity is cubed to f, LSPI quickly becomes infeasible for high dimensional systems.

3.5.5 General Algorithm

With the above distinctions, we can construct a framework in the form a general algorithm of RL with sample re-use. Algorithm 3 presents this algorithm. At every time-step k an actions a_k is chosen. After making an action, the new state and reward are observed

Algorithm 3: Sample Re-use RL

Input : RL algorithm *Learn*, database update algorithm *updateD*, number of updates K, batch size L,discount factor γ,learning rate α, exploration rate ϵ

```
initialize Q̂ arbitrarily;
D ← ∅;
for every timestep k do
    take action a_k using policy derived from Q̂;
    observe s_{k+1} and r_{k+1};
    update sample database: D ← updateD(D, (s_k, a_k, s_{k+1}, r_{k+1}));
    if mod(k,L)=0 then
        update Q-function : Q̂ ← Learn(Q̂, D, K, γ, α, ε);
    end
end
```

Output: $\widehat{Q}$

yielding the current sample $(s_k, a_k, s_{k+1}, r_{k+1})$. Together with the sample database $\mathcal{D}$, the sample is used to derive a new sample database with the underlying database algorithm *updateD*. As we have seen before, for methods without sample re-use, this consists of the simple operation : $\mathcal{D} \leftarrow \{(s_k, a_k, s_{s+1}, r_{k+1})\}$. For methods using all previous samples but no model learning, this consists of the operation: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_k, a_k, s_{s+1}, r_{k+1})\}$. Algorithm 4 present the database update for model learning. First, the artificial samples following from the previous model are deleted from the database. Then a new model $\hat{M}$ is trained on the database. In practice, this can also be done by on-line updating the old $\hat{M}$ towards the new sample. Then, n times an artificial sample is generated from the model and added to $\mathcal{D}$. In the literature, n is usually a number around 10.

After updating the database, every L-th time step $\mathcal{D}$ is used to derive a new approximation of the value function. L does not necessarily have to be a pre-defined number, updating can also be event based, for instance at the end of each episode. Furthermore, note that for purely on-line algorithms, $L = 1$.

Algorithms 5 and 6 present the two distinctive learning methods as outlined in Sections 3.5.1. Algorithm 5 represents methods using single sample updates. Since single sample updates use gradient descent on parameters, the input of this algorithm is a parametric value function. K times, a sample is taken arbitrarily from $\mathcal{D}$ and is used to calculate $\widehat{T}_d F(\theta)[s, a]$. The sample can e.g. be taken in the order of the trajectories of the episodes or sampled from a uniform distribution over the database. Dyna only uses the current sample along with artificial samples following from $\hat{M}$.

Algorithm 6 shows projection updates of $\widehat{Q}$. With projection updates the value

Algorithm 4: updateD - model learning

Input : $\mathcal{D}, (s_k, a_k, s_{k+1}, r_{k+1})$

remove samples from $\mathcal{D}$ following from previous model;
train model $\hat{M}$ from $\mathcal{D}$;
choose number of artificial samples n;
for $i = 1$ *to* n **do**
    Initialize s_m and a_m;
    Predict $\hat{r}$,$\hat{s}'_m$ using $\hat{M}$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_m, a_m, \hat{s}'_m, \hat{r})\}$;
end

Output: $\mathcal{D}$

Algorithm 5: Learn-single samples

Input : $\widehat{Q}(s, a; \theta)$, $\mathcal{D}$, number of updates K, discount factor γ, learning rate α

for $i = 1$ *to* K **do**
    arbitrarily retrieve sample (s_d, a_d, s'_d, r_d) from $\mathcal{D}$;
    determine a_{d+1} from θ;
    $\theta_{i+1} = \theta_i - \frac{1}{2}\alpha\nabla_{\theta_i}([\widehat{T}_d F(\theta_i)] - [F(\theta_i)](s_d, a_d))^2$;
end

Output: $\widehat{Q}(s, a; \theta)$

function can have any structure, not just parametric. In this type of update, K times, $\widehat{T}^*\widehat{Q}(s, a)$ is computed for all samples. Any supervised learning method can then be used to minimize the error between their the target values values in the current approximated value function $\widehat{Q}(s, a)$.

Algorithm 6: Learn-projection

Input : $\widehat{Q}(s, a)$, $\mathcal{D}$, number of updates K, discount factor γ, learning rate α

for $i = 1$ *to* K **do**
    compute $\widehat{T}^*\widehat{Q}$ of all samples in $\mathcal{D}$;
    use a supervised learning technique to minimize error between $\widehat{Q}$:
    $\widehat{Q} \leftarrow \underset{\widehat{Q}}{\operatorname{argmin}} \sum_{d=1}^{p} \left(\widehat{T}_d\widehat{Q} - \widehat{Q}(s_d, a_d)\right)^2$;
end

Output: $\widehat{Q}$

In Table 3.1 several prominent RL algorithms are shown along with their characteristics inside the framework. These characteristics include the type of update, what specific Bellman operator is used, the form of the database, the cost function minimizer, whether it is policy iteration (pi) or value iteration (vi), the computational complexity

and the representation of the value function.

As can be noted in the table, different ways of minimizing the cost function $\widehat{J}_{\mathcal{D}}$ with projection updates, e.g. kernel regression with KADP (Ormoneit and Sen, 2002) and least squares regression with LSPI (Lagoudakis and Parr, 2003), have yielded different algorithms. In this framework, all these algorithms fit under the FQI iteration architecture however.

Furthermore, it can be noted that all projection updates are policy iteration and all single sample updates are value iteration. While it is true that all methods using projection updates are indeed policy iteration, methods using single sample updates do not necessarily have to be value iteration: while making updates, the policy can be fixed until a satisfactory value function has been derived, after which the policy can be updated.

Table 3.1: Algorithm properties.

	Update	Bellman operator	Cost function minimizer	sample set used for updates	pi or vi	Computational complexity	Representation
Sarsa	samples	T^π	gradient descent	$\{(s_k, a_k, s_{k+1}, r_{k+1})\}$	vi	$\mathcal{O}(n)$	Parametric
Q-learning	samples	T^*	gradient descent	$\{(s_k, a_k, s_{k+1}, r_{k+1})\}$	vi	$\mathcal{O}(n)$	Parametric
ER-sarsa	samples	T^π	gradient descent	$\{(s_d, a_d, s_{d+1}, r_{d+1})\|d = 1, \ldots, k\}$	vi	$\mathcal{O}(n+K)$	Parametric
ER-Q learning	samples	T^*	gradient descent	$\{(s_d, a_d, s_{d+1}, r_{d+1})\|d = 1, \ldots, k\}$	vi	$\mathcal{O}(n+K)$	Parametric
KADP	projection	T^*	kernel regressors	$\{(s_d, a_d, s_{d+1}, r_{d+1})\|d = 1, \ldots, k\}$	pi	$\mathcal{O}(nEi)$	Non-parametric
FQI	projection	T^*	any	$\{(s_d, a_d, s_{d+1}, r_{d+1})\|d = 1, \ldots, k\}$	pi	$\mathcal{O}(nEi)$	Any
NFQ	projection	T^*	gradient descent	$\{(s_d, a_d, s_{d+1}, r_{d+1})\|d = 1, \ldots, k\}$	pi	$\mathcal{O}(nEi)$	ANNs
LSPI	projection	T^*	least squares	$\{(s_d, a_d, s_{d+1}, r_{d+1})\|d = 1, \ldots, k\}$	pi	$\mathcal{O}(nEf^3)$	Parametric, Linear
Dyna	samples	T^π/T^*	gradient descent	$\{(s_k, a_k, s_{k+1}, r_{k+1})\} \cup \mathcal{M}$	vi	$\mathcal{O}(ni+K)$	Parametric

3.5.6 Framework Limitations

At this point, it is important to note that not all RL methods which re-use samples fit in this framework. This framework considers only sample re-use methods using a action-value function to store expected values. In the RL literature, there are policy search techniques which re-use samples to compute policy gradients such as PILCO (Deisonroth and Ramussen, 2011), a model learning technique or R^3 (Hachiya et al., 2009), which re-uses samples to determine the policy gradient. These methods use no explicit value function and update the policy directly. Consequently, they are not centered around the Bellman operator and therefore, the way they use their samples fundamentally different.

With model learning techniques, different ways of action selection have yielded different algorithms such as Dyna-2 and the E^3 and R-MAX algorithms (SIlver et al., 2008; Kearns and Singh, 1998; Brafman and Tennenholtz, 2001). Since this framework covers general ways of re-using samples, it leaves out any distinctions made by action selection however.

Furthermore, this framework can easily be extended to actor-critic algorithms using sample re-use for critic updates. This has for instance been done in (Cheng et al. 2011). Experience replay is possible for actor updates as well, this involves calculating a direction in which the actor is modified on an average in state s (Wawrzynski, 2009). This makes learning is not purely centered around applying the Bellman operator and this method is therefore not considered in this framework.

3.6 Discussion

In this section, the framework presented in this chapter will be discussed. Firstly, the implications of the type of update and of the composition of sample database will be discussed. Next, some novel sample re-use methods emerging from the unified framework will be proposed. Then, some hints will be given regarding what algorithm to choose for a specific problem. And finally, a sample re-use method to be used for analysis throughout the rest of this thesis will be selected.

Using either separate sample or projection updates can have several consequences. The biggest practical differences between single sample and projection updates are concerning stability issues and computational complexity. Stability issues are mainly caused by unwanted consequences on other parts of the state-action space when making updates on the error following from single samples (Gordon, 1995). This can results in unpredictable and slow learning depending on the function approximator. This effect can be prevented with projection updates (Lange et al., 2012). As a rule of thumb, global function approximators are generally most affected by these issues (Gordon, 1995) and might therefore require projection updates to yield a satisfactory performance.

A projection update is inherently policy iteration, therefore we are bound to making sweeps over the entire sample database. This can be very inefficient since not all samples in the database might result in useful updates. Additionally, it takes an entire sweep to back-propagate information one step back in temporal order. As we have seen, with single sample updates we can select our samples arbitrarily and with each update, the

policy changes simultaneously. Consequently, a single update on the value function using a single sample can back-propagate information one step back in temporal order. Furthermore, arbitrary sample selection allows us to avoid making computations with unhelpful samples. From the literature, there are indeed indications that single sample updates are generally more computationally efficient. Kalyanakrishnan and Stone (2007) reported that for similar performance, FQI needed to make far more updates per episode than ER. Adam et al. (2012) stated that ER, exploits computational efficiency of the Sarsa or Q-learning algorithm.

We can note several things regarding the sample database as well. Due to restrictions of computational power or a time variant environment not all previous samples might be helpful. It is difficult to determine however, the exact sample database needed to get satisfactory performance so making an educated selection will be difficult.

Considering the use of model, because both $\hat{M}$ and $\widehat{Q}$ result from the same database one could argue that patterns found in $\hat{M}$ can be found in $\widehat{Q}$ as well. However, without model learning, these patterns might only emerge when $\widehat{Q}$ has converged for the given data. With model learning, we do not have wait for such convergence since the model is trained to static outputs. Additionally, a major difference between model learning and 'pure' sample re-use, is that model learning has two levels of approximation and thus two levels of generalization: an approximate model is used to approximate a value function. This can result in powerful generalization properties but at the same time high sensitivity to approximation errors.

From this framework, several novel methods emerge. Note that it is difficult to predict the value of these new methods beforehand. Future experiments should therefore turn out their usefulness. Firstly, when using a database filled with all samples and the model, $\{(s_d, a_d, s_{d+1}, r_{d+1}) | d = 1, \ldots, k\} \cup \mathcal{M}$, old samples might be used for replay as well, yielding a combination of ER and model learning. One could imagine for instance ER with interpolation between samples. Additionally, when using model learning, the value function can be updated with projections. This might solve stability issues depending on the representation of the value function. This would for instance allow us to use ANNs for the value function in combination with model learning. Furthermore, a policy iteration variant of ER can be used. From experiments conducted for this thesis it turned out that this can solve jumping between policies.

One can wonder what algorithm to choose for a specific problem. There is no easy answer to this but certain hints can be given. First off, a choice between T^π and T^* can be made. For certain problems it might be useful to learn a safe policy. It has been shown that using T^π can yield a safer policy than learning the optimal value function directly through T^* (Sutton and Barto, 1998). Table 3.1 shows that all methods using projection updates use an update with T^*. Gradient descend handles stochastic policies more natural: with a small learning rate, stochasticity has an averaging effect, gradient descent in this context is also called *stochastic gradient descent* (Baird et al., 1999). Therefore, when using on policy learning, single samples methods may be more suitable.

Among batch RL methods, the performance is purely determined by the supervised learning technique used to train the value function. It is therefore difficult to predict the performance of batch RL algorithms in general. We have seen that when using global

function approximation such as ANNs, projection updates can solve some stability issues (Ernst et al., 2005; Riedmiller, 2005) but might increase computational complexity. Additionally, if tuning parameters is problematic, non-parametric function approximation might be considered. Therefore, when the benefits of a global or non-parametric function approximation scheme outweigh the costs of possible extra computations, batch RL can be a valuable option.

A choice between learning a model and using old samples is not an easy one as well. It depends on the problem whether an accurate model can be learned without needed an excessive amount of samples. It is more easy to determine when *not* to use model learning however. It is known that the biggest downside of model learning is model errors(Sutton et al., 2008). Real world problems often have non-Markov effect and discontinuities which are difficult to represent. These can be a major source of approximation errors. This can be particular destructive the performance of model learning techniques and this effect should therefore be kept in mind when deciding on model learning.

To select a suitable sample re-use technique for the rest of this thesis, several properties are preferred. Firstly, it should be feasible in term of computational complexity to execute experiments with the algorithm. Secondly, methods which are compatible with the tile coding are preferred. This is because considerable work tuning the tile coding parameters on both the simplest walker and LEO has already been done by Schuitema (2012). And thirdly, we prefer methods which are transparent or allow easy analysis.

In the remainder of this thesis, ER with tile coding shall be used to evaluate and analyze the performance of sample re-use on walking problems. Kalyanakrishnan and Stone (2007) showed that tile coding combined with FQI has shown to perform less well than with ER in Keepaway soccer (Stone et al., 2005). Furthermore, ER is advantageous in respect of computational complexity since it allows great control over the computational requirements. Additionally, because the black box nature of batch RL makes analysis difficult (Fonteneau et al., 2012) single samples updates seems more suitable. Tile coding is not a global function approximator. We can therefore assume that updates with single sample can be made without risking unwanted effects in other part of the value function. And finally, we shall not be using model learning. Work has been done on model learning on LEO in the Delft BioRobotics Lab, which has not yet succeeded. It will therefore not be attempted in this thesis.

Chapter 4

Empirical Analysis of Experience Replay

Now that a suitable sample re-use method is chosen (Experience Replay (ER)) we can compare its performance to that of Sarsa(λ). Numerous studies have shown that ER has a superior performance to classical methods such as Sarsa (Lin, 1992; Smart and Kaelbling, 2000; Dung et al., 2008; Adam et al., 2012). To the best of the author's knowledge, no studies show the performance of ER in comparison with Sarsa(λ) however. In this thesis, this will be done with three experiments: simulations of the inverted pendulum, the simplest walker model (Garcia et al., 1998) and LEO.

In the first section of this chapter a specific replay method will be selected. Then, the experimental setups will be discussed in detail. Next, the results will be presented followed by a discussion.

4.1 Replay technique

As mentioned in Section 3.2 there exist several ways of presenting the data to the underlying solving algorithm. First of all, samples can be drawn from the database independently and randomly. Secondly, samples can be drawn from the database in form of trajectories. In this technique, trajectories are drawn randomly from the database, and for the chosen trajectory, the samples will be drawn in their temporal order. And lastly, the samples can be drawn in trajectories with samples drawn in their *reversed* temporal order.

Lin (1992) suggested that replaying in reversed temporal order is an effective method because a reward can be back-propagated through the entire trajectory with a single trajectory replay. If we would replay the trajectories in temporal order, multiple trajectory replays would be needed to back-propagate a reward. To the best of my knowledge it has never been shown that reverse trajectory replay is actually more effective per sample replayed however. To test this, the following experiment was done on the inverted pendulum (Section 4.2.1). First, a database of samples was collected with the Sarsa(λ) algorithm until 200 episodes have been completed. After this, ER with random samples,

trajectories and reversed trajectories was used on the database to learn a policy. After a set amount of replays, the performance of these policies were evaluated.

Full details and the results of this experiment can be seen in Appendix A. As can be seen in the results, trajectories in temporal order clearly performed worse than reversed trajectories and separate samples. Additionally, one can observe that reversed trajectories was slightly better than separate sample. Therefore, in the remainder of this thesis, ER with reversed trajectories will be used.

Each sample replay, the next action a' has to be determined in order to make an update. In (Lin, 1992) and (Adam et al., 2012), either the current policy or greedy action selection was used. The former method is called ER-Sarsa (on-policy) and latter, ER-Q-learning (off-policy). In this thesis we will be using ER-Sarsa. This is mainly because we are comparing with a on-policy algorithm. We are not so much interested in the effect of using an off-policy algorithm on LEO. The resulting ER algorithm to be used for comparison is listed as Algorithms 7 and 8 in Appendix C.

Another, more primitive way of replaying samples is with the same policy as when they were encountered. In this case, a transition can be stored in the form of: $(s_k, a_k, s_{k+1}, a_{k+1}, r_{k+1})$ with which updates can be made. This is a naive approach however, since a_{k+1} might no longer be the optimal action at state s_{k+1} at the time of replay. This especially holds for samples from early stages in the learning process.

4.2 Experimental Setups

In this section, the experimental setups which will be used to evaluate the performance of ER and Sarsa(λ) will be discussed. In this thesis, three setups were used. Simulations of the inverted pendulum, the simplest walker and LEO.

Apart from having a higher dimensionality, the walking problems have some other significant differences from the pendulum. Firstly, the system is initialized in an unstable position. By taking steps it moves from one unstable position to another. Failing to do this, will immediately lead to a failed episode. In fact, the robot either succeeds by making a step or fails by falling down. The second major difference is: with walking, success and failure are very close to each other, there is often little room for error. And thirdly, on a walking problem rewards will generally be event based, i.e. rewards will be administered as so-called box-rewards instead of through a continuous reward function. For walking, these are typically positive rewards for making steps and punishments for when the learner falls.

4.2.1 Inverted Pendulum

The inverted pendulum (Figure 4.1) was chosen to include a non-walking, low-dimensional and commonly used problem. It consists of a pendulum which is initialized pointing down. The learning task is to move the pendulum from the initial position to pointing up. A torque can be applied on the joint but it is not strong enough to directly rotate the pendulum to its up position. The agent has to learn to swing back and forth to gather enough momentum in order to swing the pendulum up.

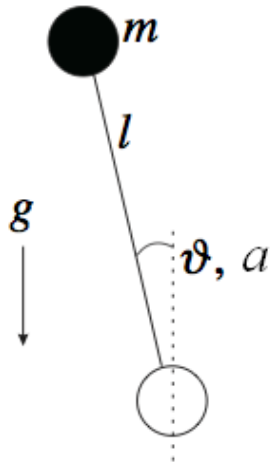


Figure 4.1: The pendulum

Table 4.1: Inverted pendulum model parameters

Model parameter	Symbol	Value	Units
Pendulum mass	m	1.113	kg
Pendulum inertia	J	$3.773 \cdot 10^{-3}$	kgm^2
Gravity	g	9.81	m/s^2
Pendulum length	l	$8.5 \cdot 10^{-2}$	m

The pendulum has been modeled in Open Dynamics Engine (ODE) in a Matlab environment developed in the Delft BioRobotics Lab (MatODE). The model parameters are listed in Table 4.1.

The state consists of the angle and angular rate of the joint: $s = [\vartheta \dot{\vartheta}]^T$. The torque is discretized into 7 actions and the reward function is the following:

$$r = -5\vartheta^2 - 0.1\dot{\vartheta}^2 - a^2 \tag{4.1}$$

With this reward function, states away from $s = [00]^T$ are punished exponentially. In addition, we can see that actions are punished. This is done to promote efficiency.

In the conducted experiments, tile coding function approximation was used with 16 tilings with a tile width of 0.33 rad and 0.33 rad/s respectively. The sampling period is 0.01 seconds, the learning rate $\alpha = 0.2$, the discount factor $\gamma = 0.98$ and exploration rate $\epsilon = 0.1$. For Sarsa(λ), the trace decay factor $\lambda = 0.92$. After every episode, 10 times the number of trajectories in $\mathcal{D}$ are replayed. An episode lasts 15 seconds or when the goal is reached. Every episode, the cumulative reward is recorded.

4.2.2 Simplest Walker

The simplest walker model (Garcia et al., 1998) consists of two rigid legs with a mass located at the hip. The feet have an infinitesimal mass and the legs are massless. The walker consists of a stance leg, which makes contact with the floor and a swing leg which exhibits a pendulum-like behavior from the hip (Figure 4.2). In order to walk without knees, the walker must be allowed to touch the ground without friction at mid-stance. When a step is made, the swing leg becomes the stance and vice versa. This way, the symmetry of walking is exploited. The state of the system consists of 4 dimensions. Namely, the angle and the angular rate of the stance leg with the ground and the angle and the angular rate of the swing leg relative to the stance leg:

$$s = \begin{bmatrix} \phi_{\text{st}} \\ \phi_{\text{h}} \\ \dot{\phi}_{\text{st}} \\ \dot{\phi}_{\text{h}} \end{bmatrix} \tag{4.2}$$

where subscripts 'st' stands for *stance* and 'h' for *hip*. The dynamics of the system can be described by a hybrid system with the following equations of motion:

$$\begin{bmatrix} \ddot{\phi}_{\text{st}} \\ \ddot{\phi}_{\text{h}} \end{bmatrix} = \begin{bmatrix} \sin(\phi_{\text{st}} - \mu) \\ \sin(\phi_{\text{h}})(\dot{\phi}_{\text{st}}^2 - \cos(\phi_{\text{st}} - \mu)) + \sin(\phi_{\text{st}} - \mu) \end{bmatrix} \tag{4.3}$$

where μ is the inclination of the floor, which is 0.004rad. The landing of the foot of the swing-leg (heelstrike) can be modeled by the following map:

$$\begin{bmatrix} \phi_{\text{st}}^+ \\ \phi_{\text{h}}^+ \\ \dot{\phi}_{\text{st}}^+ \\ \dot{\phi}_{\text{h}}^+ \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 \\ 0 & 0 & \cos(2\phi_{\text{st}}^-) & 0 \\ 0 & 0 & \cos(2\phi_{\text{st}}^-)(1 - \cos(2\phi_{\text{st}}^-)) & 0 \end{bmatrix} \begin{bmatrix} \phi_{\text{st}}^- \\ \phi_{\text{h}}^- \\ \dot{\phi}_{\text{st}}^- \\ \dot{\phi}_{\text{h}}^- \end{bmatrix} \tag{4.4}$$

which is applied once each time $\phi_{\text{st}} = 2\phi_{\text{h}}$ and $\phi_{\text{st}} < 0$. The superscript + indicates state values after the map, superscript − are state values before the map. By inspecting Equation 4.4, we can see that some energy is dissipated in the form of a slight decrease of the angular velocities of the legs at each step.

The simplest walker is a passive walker. This means that under certain initial conditions, and given that the slope of the surface is inclined, the walker can have a stable walking mode without needing any actuation. These initial conditions are very strict however (Schwab and Wisse, 2001). Fortunately, actuation on the hip joint can be used to increase the domain of attraction and increase walking speed. Because the feet and legs are massless, in practice the actuation is an angular acceleration added to $\ddot{\phi}_{\text{h}}$. The agent can choose its actions ranging from -1.2 to 1.2 rad/s^2 in 15 uniformly spaced steps.

At the beginning of each episode, the walker is initialized with random variations of

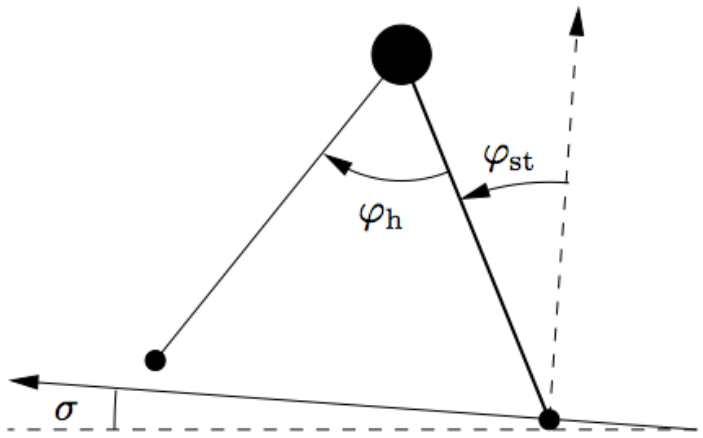


Figure 4.2: The simplest walker model. Note that ϕ_{st} is an absolute angle and ϕ_h a relative one.

Table 4.2: Tile widths used in the tile coding function approximation of Q(s,a) for the simplest walker

	ϕ	$\dot{\phi}$
Stance leg	0.10 rad	0.11 rad/s
Hip joint	0.10 rad	0.21 rad/s

the following state:

$$s = \begin{bmatrix} 0.1534 \text{ rad} \\ 0.3068 \text{ rad} \\ -0.1561 \text{ rad/s} \\ 0.0073 \text{ rad/s} \end{bmatrix} \tag{4.5}$$

The distribution of states is defined so that the walker always has enough energy to start walking.

Each episode last 100 seconds or until the walker falls down; falling is defined when the hip mass touches the ground. At every footstep, the agent receives a reward of 50 for every meter footstep length. At every fall, the agent receives a reward of -50. Additionally, every time-step is slightly punished with -1 to encourage walking speed.

Tile coding with 16 tilings is used to approximate the value function with tile widths as listed in Table 4.2. The sampling time is 0.2 seconds, the time discount factor $\gamma = 0.99$, the exploration rate $\epsilon = 0.05$ and the trace decay factor $\lambda = 0.92$, these value are taken from (Schuitema, 2012). The learning α is 0.2. This value was manually tuned. Schuitema originally used a learning rate of 0.4 for the simplest walker, a learning rate of 0.2 yielded better performance with ER however.

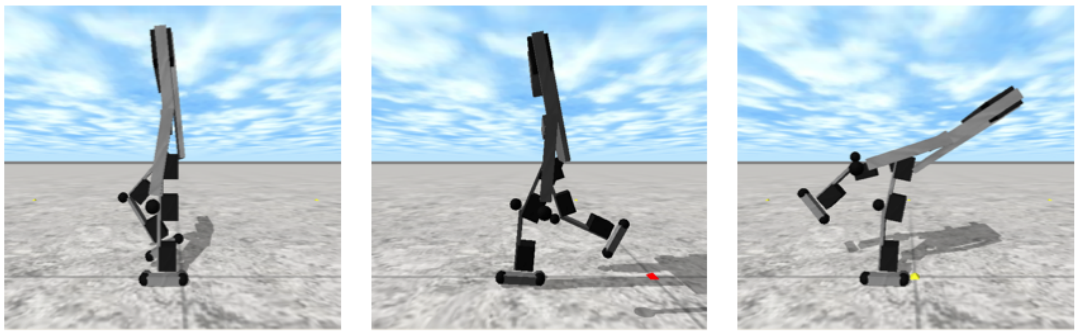
After every step or fall, 100 trajectories are replayed. Increasing this amount showed no significant performance increase while computational time rose linear. After every 100 seconds of learning time, a series of 10 evaluation runs without learning and exploration of 100 seconds was performed. Each evaluation run, the walking distance is recorded. After 10 runs, the average of the distance is taken. This is done because the initial conditions influence the walking speed of the simplest walker. Taking the average over multiple evaluation runs will therefore give a more accurate indication of the performance of the learner. Additionally, the number of falls were recorded during the learning process.

4.2.3 LEO

LEO is a real, high dimensional walking robot. In this thesis, a simulation of LEO shall be used. The reason for this is that learning with Sarsa(λ) on the real robot is very impractical due to hardware wear. This makes comparison of ER with Sarsa(λ) on the actual robot impractical as well.

LEO is modeled in the rigid body dynamics simulator Open Dynamics Engine (ODE) by Schuitema (2012). In the simulation the boom is modeled by an extra mass at the hip. Figure 4.3 shows the simulated environment.

To reduce the state-action space to a more feasible dimension, the ankle actuators are controlled so that the feet remain parallel with the ground. The resulting state space spans 10 dimensions. Namely the angle and angular rate of the torso, the hip joints of



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

Figure 4.3: Simulation of LEO in several situations occurring in the learning process. The initial configuration (a) is varied slightly each episode. (b) shows a still of a successful gait. In (c), the robot is doomed to fall.

both legs and knee joint of both legs:

$$s = \begin{bmatrix} \phi_{\text{torso}} \\ \dot{\phi}_{\text{torso}} \\ \phi_{\text{st.hip}} \\ \dot{\phi}_{\text{st.hip}} \\ \phi_{\text{sw.hip}} \\ \dot{\phi}_{\text{sw.hip}} \\ \phi_{\text{st.knee}} \\ \dot{\phi}_{\text{st.knee}} \\ \phi_{\text{sw.knee}} \\ \dot{\phi}_{\text{sw.knee}} \end{bmatrix} \tag{4.6}$$

The learn-able actuators are situated at both hip joints and in the knee of the swing leg. They are are discretized into 7 actions, yielding $7^3 = 373$ different actions to choose from.

A walking episode lasts for 25 seconds or when the robot is doomed to fall. The robot is doomed to fall when the torso angle is too large, $|\phi_{\text{torso}}| > 1.0\text{rad}$ (see Figure 4.3) or when the stance leg angle becomes too large, $|\phi_{\text{torso}} + \phi_{\text{st.hip}}| > 1.13\text{rad}$ (Schuitema, 2012). At the beginning of each episode, the robot starts with small random variations

Table 4.3: Tile widths used in the tile coding function approximation of Q(s,a) for LEO's learning task

	ϕ	$\dot{\phi}$
Torso	0.14 rad	5 rad/s
Stance hip	0.28 rad	10 rad/s
Swing hip	0.28 rad	10 rad/s
Stance knee	0.28 rad	10 rad/s
Swing knee	0.28 rad	10 rad/s

of the following state:

$$\begin{bmatrix} \phi_{\text{torso}} \\ \dot{\phi}_{\text{torso}} \\ \phi_{\text{st.hip}} \\ \dot{\phi}_{\text{st.hip}} \\ \phi_{\text{sw.hip}} \\ \dot{\phi}_{\text{sw.hip}} \\ \phi_{\text{st.knee}} \\ \dot{\phi}_{\text{st.knee}} \\ \phi_{\text{sw.knee}} \\ \dot{\phi}_{\text{sw.knee}} \end{bmatrix} = \begin{bmatrix} 0.10 \text{ rad} \\ 0 \text{ rad/s} \\ 0.10 \text{ rad} \\ 0 \text{ rad/s} \\ 0.82 \text{ rad} \\ 0 \text{ rad/s} \\ 0 \text{ rad} \\ 0 \text{ rad/s} \\ -1.27 \text{ rad} \\ 0 \text{ rad/s} \end{bmatrix} \tag{4.7}$$

The sample time is 1/30s, the discount factor $\gamma = 0.996$ and an exploration rate $\epsilon = 0.05$ and with Sarsa(λ), the eligibility trace decay factor $\lambda = 0.859$. Tile coding with binary basis functions are used with 16 tilings with tile widths as listed in Table 4.3. These values are directly taken from (Schuitema, 2012) who spend considerable time tuning these parameters.

Also taken from (Schuitema, 2012), is the reward function. The agent receives a reward of 300 m^{-1} of positive displacement of the swing foot and -300 m^{-1} for negative displacement. When the robot falls, a reward of -125 is given. Time and energy usage are punished with -1 every time step and -3 J^{-1} of electrical work. Note that this function was extensively tuned to work well with Sarsa(λ). This took a considerable amount of work and due to time limitation we did not attempted to alter it.

On the experiments on LEO, it turned out that using ER while exploring with Sarsa(λ) in between replays, yielded a better sample database in terms of percentage of the state space visited, and thus better results than when using ER alone. Therefore, Sarsa(λ) is used to gather samples in between replays.

After every 100 episodes, 1500 trajectories were replayed. After each 200 seconds of simulated learning time a test run of 25 seconds was performed without exploration. During this test run, the walked distance was stored. Additionally, the cumulative number of falls was stored during learning.

4.3 Results and discussion

Figure 4.4 presents the results of Sarsa(λ) and ER on the inverted pendulum, the simplest walker and LEO. The Sarsa(λ) algorithm can be found in Section 2.6.3, the used ER algorithm is listed as Algorithms 7 and 8 in Appendix C. The graph of the inverted pendulum shows the total accumulated reward against the number of episodes. The graphs of the simplest walker and LEO show the traveled distance over 100 and 25 seconds respectively, against learning time. Each graph shows the average of 20 runs with a 95% confidence interval of the average.

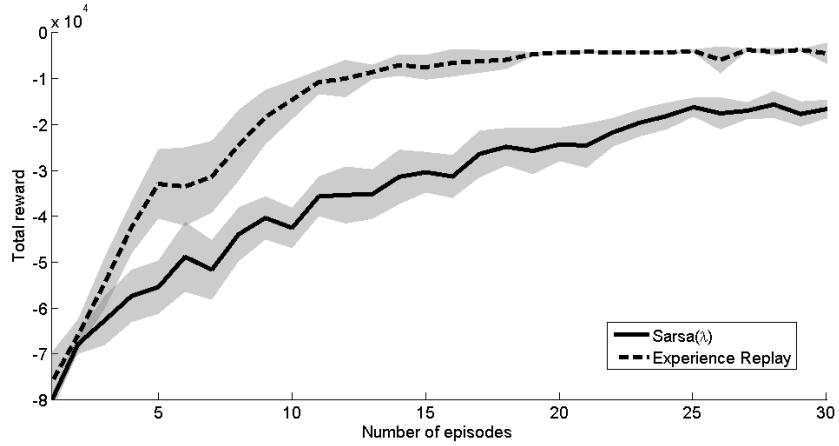
We can observe from the results that learning with ER is clearly faster on the inverted pendulum. However, for the simplest walker and LEO, the performance of ER is worse than that of Sarsa(λ). Typical runs of Sarsa(λ) and ER on the simplest walker and LEO are shown in Figure 4.5. While performance is initially good in most learning runs, learning with ER is generally slower and very unpredictable compared to Sarsa(λ). Most runs exhibit typical drops in performance on both the simplest walker and LEO shown in Figure 4.5. Additionally, some runs with ER on LEO failed to find a successful policy altogether within reasonable time. Seemingly, ER distorts learning in some way and this effect is most profound on walking tasks.

In Figure 4.6, the typical behavior of the simplest walker resulting from both algorithms is shown. In the figure, the angle of the stance leg and the hip joint are plotted against time. Additionally, the height of the foot of the swing leg is plotted. For interpretation of this figure, it is important to note that a successful step is very similar to an unsuccessful one on the simplest walker. This is illustrated in Figure 4.7. In fact, whether or not the walker makes a step can be an infinitesimal difference.

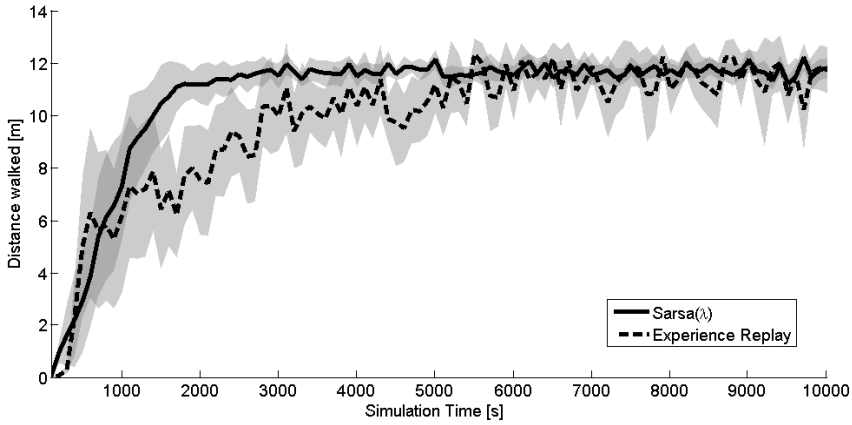
In the figure we can see that only a slight change in how high the foot is lifted from the ground, can result in a fall. The height of the foot gives an indication of how 'safe' a given trajectory is. By lifting the foot higher prior to each step we are further away from the infinitesimally thin devision line between success and failure. We can see that the behavior following from ER, hardly lifts is foot from the ground. This is an unsafe strategy: there is little margin for error and at some point the walker will fail to lift its foot high enough and it will fall. Indeed, many of these type of falls were observed and the walker seemed to only slowly to learn from them. This behavior was still visible even after a relative high number of successful episodes. In contrast, Sarsa(λ), learned a more safe policy. As can be seen in Figure 4.6, the walker clearly lifts it foot much higher before heel-strike when compared with the behavior following from ER.

With LEO, unsafe trajectories are more difficult to identify due to the higher complexity. However, some additional issues were be observed. Sometimes during learning, the simulated robot tended to stand still or balance without any intention of walking forward. The agent only slowly seemed to unlearn this behavior. Again, these problems did not occur with Sarsa(λ).

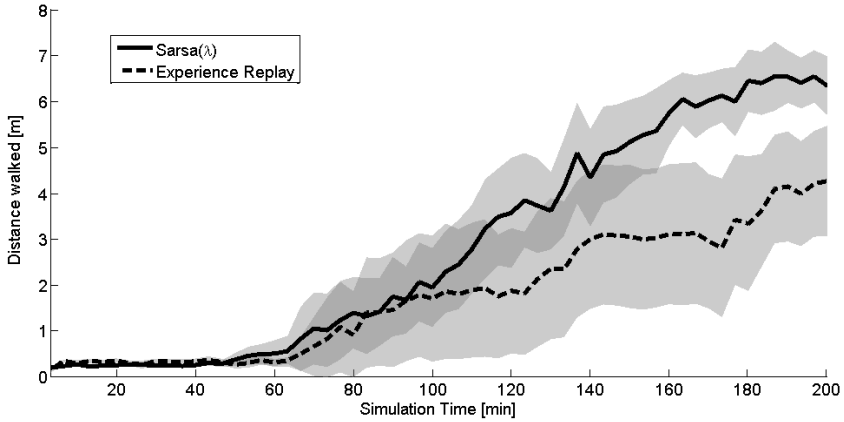
From these observations it can be concluded that on the walking problems, ER fails to derive a successful policy even if the given data contains successful trajectories. Failure of ER and batch RL to produce optimal policies even if the data contains (near-)optimal trajectories been reported several times before in the literature (Fonteneau



(a) The inverted pendulum

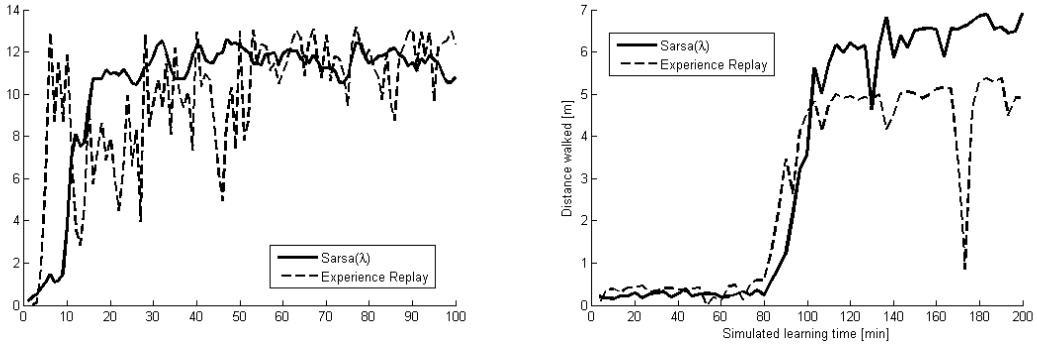


(b) The simplest walker



(c) LEO

Figure 4.4: Performance of Sarsa(λ) and ER on the compass walker. The solid and dotted line represent the mean of 20 runs, the shaded area represent the 95% confidence interval of the average.



(a) The simplest walker (b) The simplest walker

Figure 4.5: Typical runs of Sarsa(λ) and ER on the simplest walker and LEO, illustrating unpredictable learning with ER.

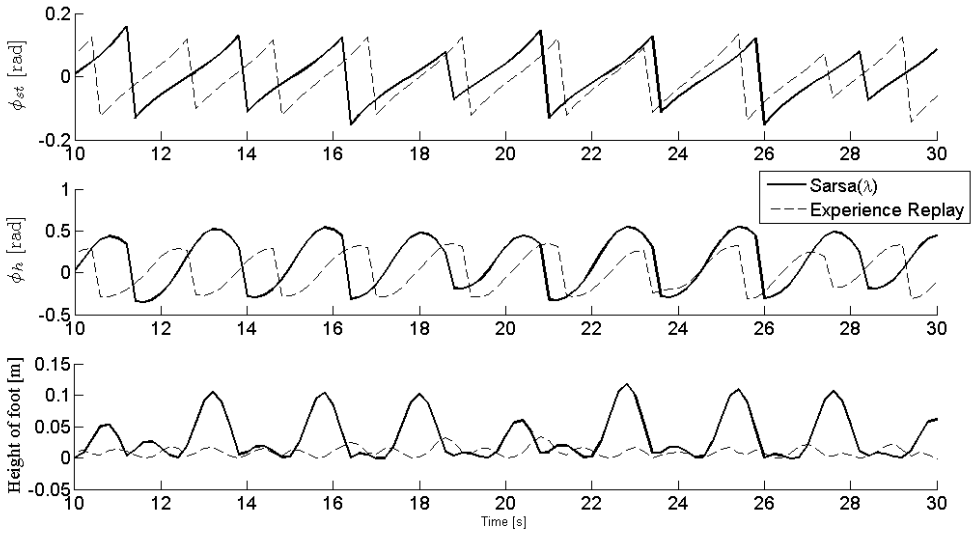


Figure 4.6: Typical learned evolution of the angle of the stance leg, the hip joint and of the height of the foot of the simplest walker. One can observe that the policy following from ER hardly lifts its swing foot from the ground, indicating the unsafeness of the behavior.

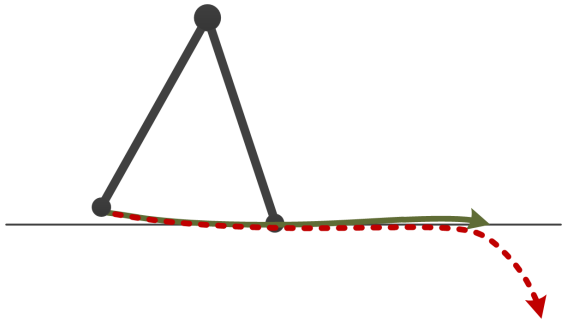


Figure 4.7: Drawing of the simplest walker with typical trajectories of the foot. The solid green line represents a successful step, the dotted red line is a fall indicating the similarity of success and failure. If the foot is not lifted enough from the ground, the foot will not land on the ground resulting in a fall.

et al., 2012; Kalyanakrishnan and Stone, 2007, 2011). Fonteneau et al. (2012) reported observing unsafe policies as well. This time, in a puddle world problem and with FQI. Furthermore, batch RL in combination with tile coding has no guarantee to converge and has been shown to sometimes diverge, especially with an incomplete data set (Timmer and Riedmiller, 2007; Kalyanakrishnan and Stone, 2007). Lin (1992) examined the issue of so-called over-training,which occurs when the same experiences are used far too many times, saying that a value function can become too specific to that experience, which usually harms generalization.

While these problems have been reported clearly, little knowledge on the working principles behind them exists. In the following chapter, the issues causing the problems outlined here will be addressed.

Chapter 5

Issues with ER on walking problems

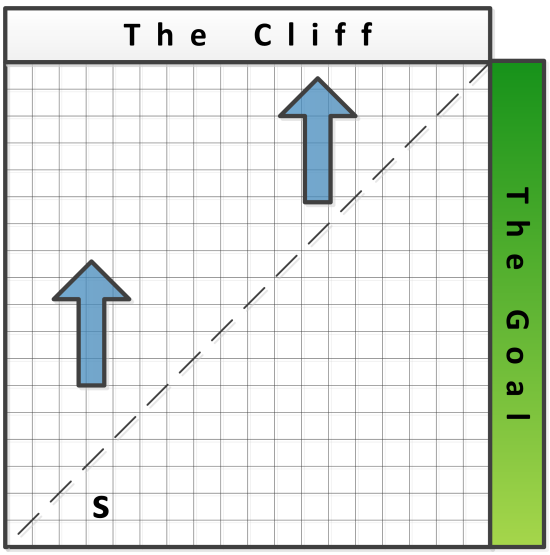
In the previous section, we observed unsafe behavior of the learner on the simplest walker and reported balancing or still standing behavior of the learner on the simulation of LEO. This chapter will analyze the issues causing this behavior and propose solutions to solve them.

Due to the high amount of samples replayed and the high dimension of the state-action space, analysis of the issues when learning with ER is difficult on the simplest walker and LEO themselves. Additionally, because of the complexity of these problems, they might possess a multitude of confounding effects making it difficult to draw conclusions. Therefore, we will illustrate and analyze these issues with a simple grid world inheriting some characteristic of walking. The analysis will be done by visualizing typical Q-functions arising when using ER on this grid world. Visualizing the Q-function can give valuable insight in the learning process, and because the state dimension of a grid world grid world is 2 ($x-$ and $y-$coordinate), this can be done easily.

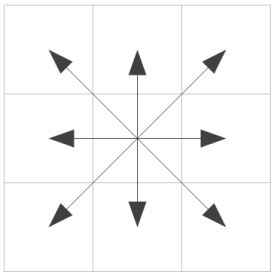
The first section in this chapter will introduce the grid world and present typical value functions of ER and Sarsa(λ). Section 5.2 will discuss the first of two issues, regarding replaying with uncertain Q-values. Section 5.3 will discuss the second issue, concerning emerging local maxima. For both these issues a solution will be proposed and typical value functions with the new algorithms will be presented. Finally, Section 5.4 will show typical value function when both solutions are combined.

5.1 Grid world

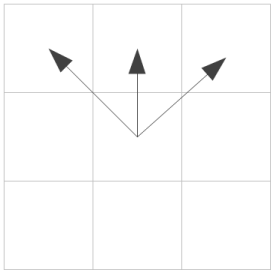
Figure 5.1 presents the grid world which will be used throughout this chapter. It is a windy grid world with a cliff at one size and a goal state at another. The agent is initialized at S and the goal is to navigate safely and as quickly as possible to the highest reward possible. The wind only applies at and above the striped line and causes the agent to be blown towards the cliff. Out of the wind, the agent can make king's moves. In the windy area, any action results in standing still or going one tile towards the cliff, see Figures 5.1b and 5.1c. Consequently, once the agent is above the striped line, it is doomed to fall from the cliff. The reward for reaching the goal gets higher for reaching



(a) The windy gridworld.



(b) King's moves



(c) Windy moves

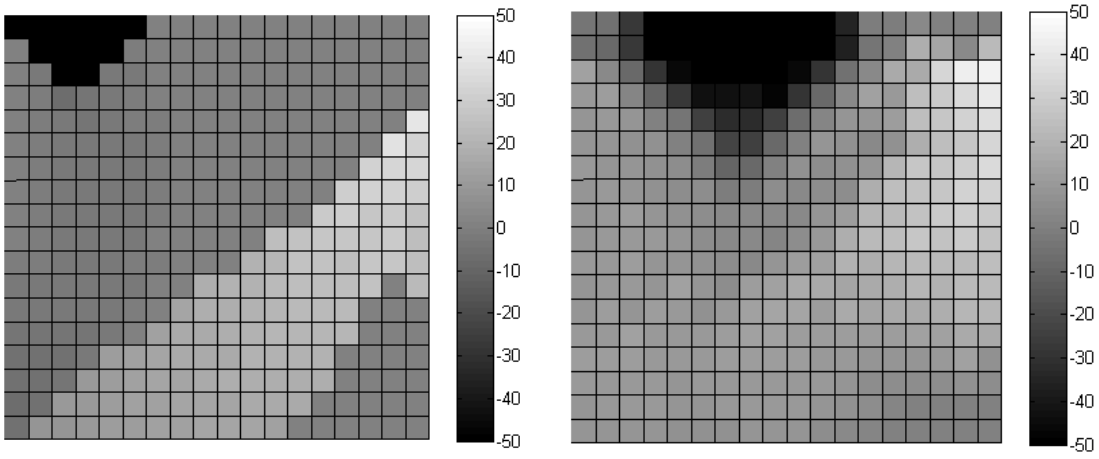
Figure 5.1: (a) shows a drawing of the windy grid world which allows analysis of issues arising when applying ER to walking problems. It has some characteristics of a walking problem such as box-rewards and closeness of success and failure. (b) shows the moves the agent can make out of the wind, (c) shows the possible moves in the windy area.

the goal closer to the cliff. A reward is given of 50 at the most northern state, down linearly to 0 at the most southern state. To promote finding the quickest route, the agent receives a punishment of -1 each time step. The features described above give the grid world some of the characteristics of a walking problem: it contains box-rewards, success and failure are close to each other (especially in the top right corner) and certain regions in the state-space are doomed to lead to failure.

In the experiments on the grid world, ER is used after every episode. Each replay, 10 times the current amount of trajectories are replayed. The discount factor $\gamma = 0.98$, the exploration rate $\epsilon = 0.05$ and the learning rate $\alpha = 0.25$, these are values commonly used in RL literature. Two cases will be considered: *with* and *without* function approximation. Tile coding will be used with 16 tilings and tile widths of two times the width of a grid.

Figure 5.2 shows typical value functions of separate learning runs *with* and *without* using function approximation. The value functions are captured shortly after the goal state has been discovered. Typically at this stage, the agent has already explored a large portion of doomed to fall area.

In both cases (Figures 5.2a and 5.2b),we can observe that the punishment of falling is not back-propagated through the doomed to fall area. In fact, only a small portion of the windy area shows to have a negative expected return. This is remarkable since all trajectories entering this area end up falling in the cliff. Without function approximation, most of the doomed to fall region is still at the initialized value of 0. When using ER



(a) no function approximation

(b) tile coding function approximation

Figure 5.2: Typical projected value functions of the grid world using ER *with* (a) and *without* function approximation after the goal has been discovered. Illustrates the lack of back propagation, and generalization into the windy area with function approximation.

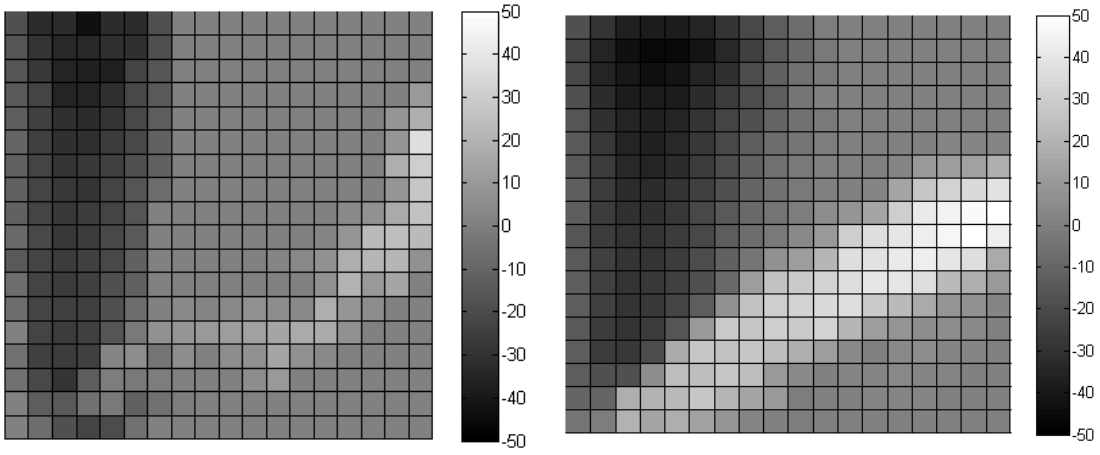
with function approximation (Figure 5.2b), the situation is even worse. Again we can see the poor ability of ER to back-propagate punishments but this time, instead of not only failing to back-propagate the punishment, some states in the doomed-to-fall area have positive expected rewards. Consequently, the value function shows no clear line marking the beginning of the windy area.

We can conclude that ER fails to back-propagate the punishment of failures. In fact, it can be observed that ER only learns from rewards higher than the initialized value. With function approximation, the agent does not only fail to back-propagate punishments, it generalizes positive rewards into the doomed-to-fall area. Because of these issues, ER typically yields a value function resulting in a failing policy leading straight towards the cliff. This explains unsafe behavior following from ER observed on the simplest walker.

In contrast, Figure 5.3a shows the value function of Sarsa(λ) in a similar situation. One can observe that with Sarsa(λ), the issues outlined above do not occur. The punishment is clearly back-propagated to the initial state and there are no positive expected rewards in the windy area.

As we will see, the issues described here are a result of ER replaying with uncertain Q-values caused by an incomplete sample database. Initialization causes the algorithm to be optimistic in the face of uncertain Q-values. This effect causes ER to forget trajectories leading to falls and generalize positive rewards of the goal state into the doomed-to-fall area. Section 5.2 will discuss this in detail and present a new algorithm to solve these issues.

When using function approximation, a second problem can be observed: sometimes local maxima show up in the value function. Mostly, these local maxima quickly disap-



(a) no function approximation

(b) tile coding function approximation

Figure 5.3: Typical projected value functions of the grid world using Sarsa(λ) after the goal has been discovered, *with* and *without* function approximation. Illustrates that Sarsa(λ) does back-propagate the punishment and does not produce positive rewards in the windy area.

pear but in some circumstances, they might grow to extremely large values. On the grid world, this mostly happens with high learning rates (α >0.3) or when replaying a long time without gathering new samples. Figure 5.4 shows two value functions with local maxima.

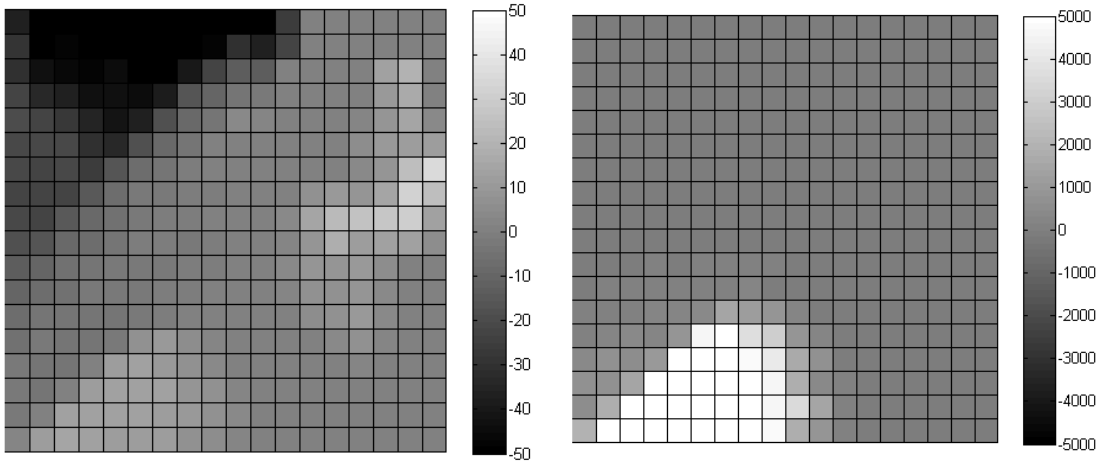
The local maxima, illustrated in Figure 5.4, are caused by updates on state-action combination which affect their own Q-value. This, together with approximation errors, can cause an avalanche effect resulting in rising Q-values. In Section 5.3, this will be discussed in detail. Along with this, a sample re-use variant of a solution to this problem suggested by Baird (1995) will be proposed.

5.2 Failing back-propagation

This section will discuss the inability of ER to back-propagate punishments and the observed generalization of positive expected values into the windy area. Firstly, the details of the process involved will be discussed. Next, a new algorithm will be proposed to address these issues.

5.2.1 Attitude towards uncertainty

Consider a value function initialized at zero and a database of samples containing several trajectories leading to a fall. Of a certain sample at state-action combination (s, a), the expected value corresponding to next action a'_1 at next state s' has been updated to a negative value due to a fall. All the other state-action combinations in $\{(s', a')|a' \in \mathcal{A}\}$



(a) (b)

Figure 5.4: Typical projected value functions of the grid world using ER after the goal has been discovered, showing local maxima in the value function. These maxima can grow to large values. In these cases, old samples were replayed a relatively long time without gathering new samples.

are unvisited and remain at their initialized expected value. An update on $Q(s, a)$ will then result in an update towards target value 0, since the Bellman operator will select an unvisited action in $\max_{a \in \mathcal{A}} Q(s', a)$. This will happen because the visited state-action pair has a lower value then the initialized value of 0. However, the values of the unvisited state-action pairs are not based on any actual experience and has thus carry no real information about the environment. Consequently, ER will back-propagate with a value initialized by the user while the actual future reward of that unvisited state-action pair remains uncertain.

This is an example of when the algorithm replays with an optimistic Q-value. Consequently, this will result in an agent which expects it can 'save' itself by choosing an unvisited action. In other words, it is optimistic in the face of uncertainty. Because of this, the algorithm prefers to replay with uncertain Q-values over values having a low expected value. We can state that the initialization of the value function dictates the attitude towards uncertainty. If the value function is initialized with high values, the algorithm will favor uncertain state-action pairs. If it is initialized with low values, it will favor certain or visited state-action pairs. This effect will cause the punishment is not back-propagated to previous state-action combinations is the value function is initialized optimistically, i.e. with values higher than the expected punishments. Note that when every state-action combination is visited, this problem does not occur. Thus, if there is lack of data of $Q(s', a')$-values along a doomed trajectory, and these values are initialized with optimistic values, ER will fail to back-propagate punishments.

As we have seen in the previous chapter, the problem discussed here does not occur with Sarsa(λ). Sarsa(λ) updates the entire followed trajectory when the agent arrives at

a fall. This makes it is closer to a Monte Carlo update (Sutton and Barto, 1998). Monte Carlo updates are updates based on the outcome of the entire trajectory. Therefore, it punishes the entire trajectories leading to falls automatically. It does not matter if a $Q(s', a')$-value that Sarsa(λ) uses for an update is uncertain, because the agent will evaluate them in the environment and update them accordingly. It is therefore that Sarsa(λ) *does* back-propagate punishments regardless of the initialization of the value function.

In Figure 5.2b in the previous section, we observed generalization of positive rewards into the doomed-to-fall area. In areas where success and failure are close to each other, positive rewards might be generalized to state-action pairs leading to failure. If the algorithm is optimistic towards uncertainty, these generalizations are easily 'accepted' into areas leading to failure. In other words, a positive expected value of a *safe* state-action pair might be generalized to an *unsafe* state-action pair. Due to optimism in the face of uncertainty, the ER algorithm select this value for replay even though that particular (s', a') will lead to a fall eventually. Additionally, with an incomplete data set, large approximation errors of the Q-values of unvisited (and therefore uncertain) state-action pairs might arise. With tile coding this usually occurs when close state-action pairs, have a large variance in their expected values. When optimistic in the face of uncertainty, these errors are more often used for replay then when we would be pessimistic in the face of uncertainty. This can result in a built-up of approximation errors resulting is wrong Q-values.

In the next section, a new algorithm called ER-σ will be proposed to deal with the issues outlined here.

5.2.2 ER-σ

Since the initialization of the value function determines the attitude towards uncertainty, an obvious solution seems to be to initialize the value function pessimistically, e.g. with a value lower than the to be expected rewards. However, this gives rise to some new problems. Firstly, we lose optimistic exploration. Optimistic exploration occurs when the value function is initialized with values higher then the to be expected rewards. When the agent takes a certain action, its received reward will almost certainty be lower than the initialized values. Therefore, actions not already taken before will have a higher expected value than actions already taken. And thus, unvisited state-action combinations will often be favored to visited ones. This type of exploration has shown to be an effective way of increasing learning speed (Sutton and Barto, 1998). Otherwise, pessimistic initialization would work well if we would not be using function approximation. In this section we will see that *with* function approximation, pessimistic initialization results in ineffective back propagation of positive rewards. This is due to the fact that the algorithm will now back-propagate only through state-action pairs actually visited. In other words, because ER is now pessimistic in the face of uncertainty and will therefor take less 'chances'. We will show that the new algorithm proposed in the next section will back-propagate positive rewards more effectively than pessimistic initialization.

If we desire back-propagation of punishments but at the same time efficient spreading

of positive rewards, the algorithm should be optimistic when the agent is in the safe areas but pessimistic when it is in the windy area. That is, the attitude of the algorithm towards uncertainty should be dependent of the state. This way, the algorithm will back-propagate punishments and still take chances when it is safe to do so. We hypothesize that this will increase sample efficiency on the simplest walker and LEO with respect to vanilla-ER.

In order to do this, we will introduce a measure for uncertainty u of a certain $Q(s', a')$-value and a function σ to determine to what extend to be optimistic or pessimistic. In the following, u and σ will be derived for linear function approximation. With linear function approximation, uncertainty of $Q(s', a')$ can be measured by the degree of features that have been visited of that state-action pair (s', a'). To what extent a state-action pair is visited, can be stored in a function similar to the value function. This function, W, is defined by a parameter vector w and the same feature vector as the state-action value function ϕ: $W(s, a) = w^T \phi(s, a)$.

By setting the value of W at every state-action combination we visit, to the value of the total feature membership, we keep track of the total visited feature membership over the state-space. This can then be used to determine the uncertainty of a $Q(s', a')$-value. At state s_k after taking action a_k, the value of W at that state-action pair is set as:

$$W(s_k, a_k) \leftarrow \sum_i \phi_i(s_k, a_k) \tag{5.1}$$

The parameter update rule will then be:

$$w_k \leftarrow w_k \vee \phi(s_k, a_k) \tag{5.2}$$

where $\vee$ is the maximum operator. By comparing the value of $W(s, a)$ with the total membership of (s, a) possible $\sum_i \phi_i(s, a)$, we get a degree of how 'visited' state-action combinations are. The less 'visited' a state-action combination is,the more uncertain its Q-value is and therefore, the more it resembles the initialized value. For a some state-action combination (s, a), we can define the uncertainty factor u can be as:

$$u(s, a) = \frac{\sum_i \phi_i(s, a) - W(s, a)}{\sum_i \phi_i(s, a)} \tag{5.3}$$

This resembles the share of features of a certain state-action combination which is *not* visited: $u = 0$ for state-action combinations actually visited and $u = 1$ if none of the features have been visited by any degree. In the latter case, that particular Q-value will still be at the initialized value. Since we want to get rid of the effect of initialization on replay, we will replace values corresponding to unvisited features with something more sensible. Specifically, a value $\sigma(s')$ as function of the current state indicating the safeness of that particular state s'. With this function, updates can be made with the following target value:

$$T = r + \gamma \max_{a' \in \mathcal{A}} \left(Q(s', a') + u(a')\sigma(s') \sum_i \phi_i(s', a') \right) \tag{5.4}$$

provided that θ is initialized with all zeros. This is not required, but for simplicity of notation this assumption is made in this section. By using the target value of Equation 5.4, we will effectively replay with a $Q(s', a')$-value in which the unvisited parameters have been replaced with the value of $\sigma(s')$. By doing so, our attitude towards uncertainty no longer depends on the initialization of the Q-function but it is now a function of the next state s'.

$\sigma(s')$ can be used to exploit the information available to us in state s'. To do this, the value of $\sigma(s')$ will be determined by looking at the expected future reward of the visited actions in state s'. That is, we can assess the safeness of a state using the expected values of known actions in state s'. Practically, this means that σ can be computed with the values of the parameters corresponding to the visited features of any action in s'.

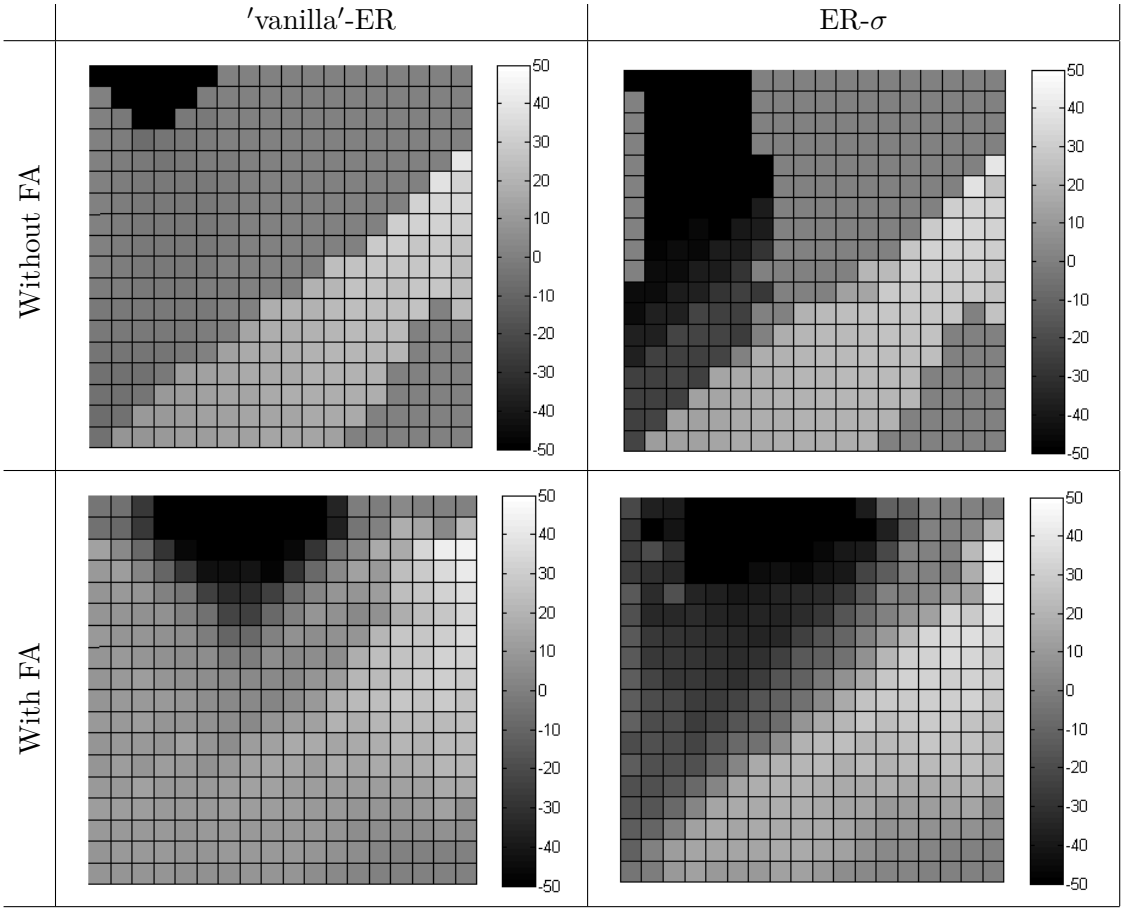
To find a suitable definition of $\sigma(s')$, we will consider three basic possibilities. Firstly, $\sigma(s')$ can be determined by assuming the worst case scenario: $\sigma(s') = \min_a (Q(s', a)/W(s', a))$. Secondly, the best case scenario can be assumed: $\sigma(s') = \max_a (Q(s', a)/W(s', a))$. And thirdly, the average of the visited parameters over all actions can be taken: $\sigma(s') = \sum_a Q(s', a) / \sum_a W(s', a)$. Additionally, we shall be considering interpolations between the worst case σ and the average σ and between the best case σ and the average σ. These possibilities are chosen because of two reasons. They consist of simple operations, keeping the computational load low. And secondly, they contain no tunable parameters.

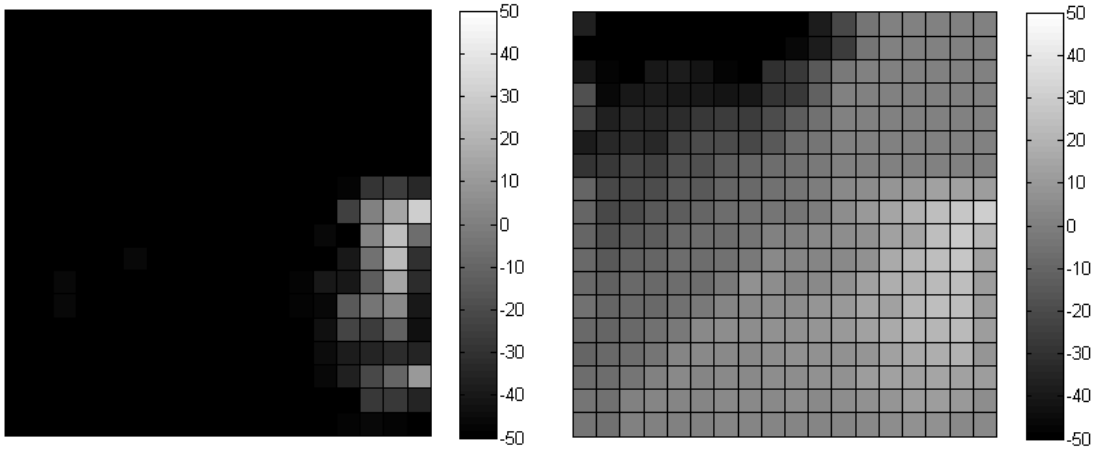
In Appendix B, the above definitions of σ have been evaluated on the simplest walker. Looking at the results we can observe that there is an optimum between assuming the value of worst case action and taking average of the value of the visited actions. Following this result, in the remainder of this thesis this particular definition of σ will be used. Algorithms 9 and 10 in Appendix 10 shows the new ER algorithm in pseudo code. This algorithm will be called ER-σ throughout the rest of this thesis.

We expect that this algorithm results in back-propagation of punishments and at the same time, effective back-propagation of positive rewards. To test this hypothesis, the new algorithm is applied on the same sample database and the exact same sample replays as was done to yield the value functions of Figure 5.2. In Table 5.1 resulting value functions of the new ER algorithm (ER-σ) are shown. It is visible that the new algorithm indeed back-propagates the punishments through the doomed-to-fall area. Additionally, no positive rewards can be found in this region of the state-action space when using function approximation. Consequently, ER-σ yields a safe policy since no positive reward will cause the agent to navigate to the windy area. Furthermore, the table shows that ER-σ back-propagates the positive reward though the state-action space more effectively than vanilla-ER: with ER-σ the reward of the goal has spread through a bigger region of the state-action space. This effect is also visible when compared to ER with pessimistic initialization, see Figure 5.5. In Chapter 6 the results of ER-σ on the benchmark problems will be presented.

Summarizing, this section discussed the issue of failing back-propagation due to optimism in the face of uncertainty. Being optimistic in the face of uncertainty during replay can result in unsafe policies. We argued that this optimism is a result of the initialization of the value function. A new algorithm was proposed, ER-σ, which makes

Table 5.1: Typical value functions of applying ER and the new ER algorithm (ER-σ) to the grid world. The punishments resulting from falling are back propagated properly with ER-σ. With function approximation positive reward are not generalized into the doomed-to-fall area resulting in value function yielding safe policy. Additionally, because optimism causes close state to have a high variance in target values, vanilla-ER exhibits more approximation errors. Per row, the same data set is used and the exact same samples are replayed.





(a) ER with pessimistic initialization.

(b) ER-σ

Figure 5.5: Typical projected value functions of the grid world using ER with pessimistic initialization and with the new algorithm ER-σ, indicating slower and more narrow back-propagation of goal-state rewards with pessimistic initialization. Pessimistic initialization causes the algorithm to be distrustful of generalization and will thus only back-propagate through actual visited trajectories. The same data set is used and the exact same samples are replayed.

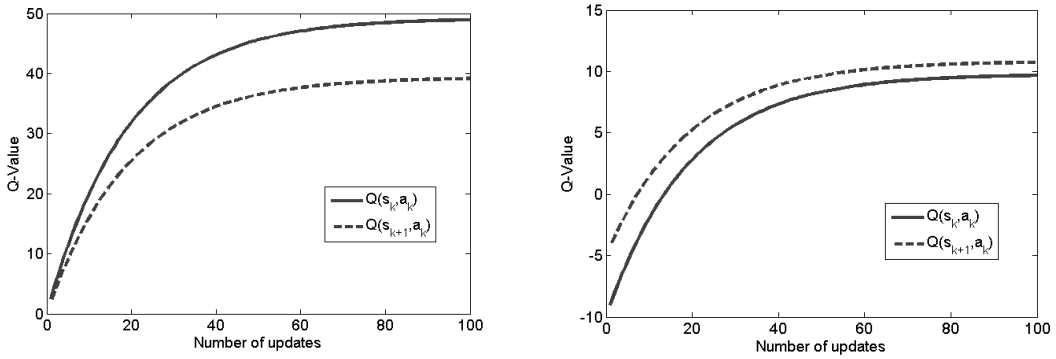
the attitude towards uncertainty a function of the next state s' through function $\sigma(s')$. A definition of $\sigma(s')$ has been found by evaluating several options on the simplest walker and selecting the one yielding the best performance. On the grid world, this algorithm results in back-propagation of punishments while effective propagating positive rewards.

5.3 Local maxima in the value function

This section will discuss the local maxima emerging in the value function as can be observed in Figure 5.4. Firstly, the details of the process involved will be discussed. Next, a new algorithm will be proposed which combines an earlier found solution with ER.

5.3.1 Self-affecting states

Experience replay produces more problems when two succeeding states s_k and s_{k+1} are close to each other. In this case, due to function approximation, an update on $Q(s_k, a_k)$ can directly influence the value of $Q(s_{k+1}, a_k)$ due to generalization. If $\max_{a \in \mathcal{A}} Q(s_{k+1}, a) = Q(s_{k+1}, a_k)$, the update on $Q(s_k, a_k)$ will then influence its own target value. For instance, consider that $Q(s_k, a_k)$ is increased in value by an update during replay. Now assume that due to function approximation, $Q(s_{k+1}, a_k)$ also increases in value by this update. The next time $Q(s_k, a_k)$ is updated, it will again be increased in value because



(a) Positive reward

(b) Negative reward

Figure 5.6: Q-values of a self-affecting state on the grid world. In case (a), the values are initially 0 and the reward is positive: $Q(s_k, a_k)_{\text{init}} = 0$, $Q(s_{k+1}, a_k)_{\text{init}} = 0$, $r = 10$. In case (b), the values are initialized negative and the reward is negative as well: $Q(s_k, a_k)_{\text{init}} = -10$, $Q(s_{k+1}, a_k)_{\text{init}} = -5$, $r = -1$. Other parameters are $\gamma = 0.98$ and $\alpha = 0.25$. These examples have been made by creating artificial samples in the grid world.

its target value has been increased by the previous update. This becomes problematic when there is no sample in the database to correct the value of $Q(s_{k+1}, a_k)$. This process can repeat itself and result in a local maximum.

Figure 5.6 illustrates what happens with Q-values when updates are made on such self-affecting states. It can be observed that these states can grow to values many times their initial target value. Additionally, even if the initial target value and the Q-value of the self-affecting state are initially negative, it can rise to positive values.

Close states do not always have to result in growing Q-values immediately. Initially, the next action a_{k+1} may not be selected to be the current action a_k. In this case, updates on $Q(s_k, a_k)$ do not influence itself directly. However, the value of $Q(s_{k+1}, a_k)$ is still affected. Although $Q(s_k, a_k)$ will not grow in this case, the effect may eventually lead to a growing Q-value since a_k might become the greedy action. Also note that when a sample at (s_{k+1}, a_k) is gathered, the value of $Q(s_{k+1}, a_k)$ may be corrected and the Q-value may seize to grow and return to 'normal' values.

5.3.2 Residual gradient ER

In on-line learning, this issue has been explored by Baird (1995). In this study, a simple problem containing self-affecting states was introduced which failed to converge using ordinary methods. The proposed solution for this is to take into account the effect of the update of $Q(s_k, a_k)$ on the value of $Q(s_{k+1}, a_k)$. These algorithms are called residual algorithms.

In practice, residual RL make updates on the value of $Q(s_k, a_k)$ while at the same time, compensating for the effect on $Q(s_{k+1}, a_k)$. A large disadvantage of doing this is

that convergence is usually a lot slower compared to classical TD methods (Baird, 1995). However, this has not yet been investigated with ER.

Baird (1995) derived his so-called residual gradient algorithm in the case of a value function. The following will very briefly review this.

Instead of updating θ with only the gradient at the current state:

$$\theta_k = \theta_k + \alpha \left(r_{k+1} + \gamma V(s_{k+1}) - V(s_k)\right) \frac{\delta}{\delta\theta} V(s_k) \tag{5.5}$$

the gradient at the next state can be taken into account as well:

$$\theta_k = \theta_k + \alpha \left(r_{k+1} + \gamma V(s_{k+1}) - V(s_k)\right) \left(\frac{\delta}{\delta\theta} V(s_k) - \frac{\delta}{\delta\theta} \gamma V(s_{k+1})\right) \tag{5.6}$$

The above equation can be extended to the action-value function case. With linear function, this yields the following update rules:

$$\tilde{\phi} = \phi(s_k, a_k) - \gamma\phi(s_{k+1}, a_k) \tag{5.7}$$

$$\theta_k = \theta_k + \alpha(r_{k+1} + \gamma Q(s_{k+1}, a_{k+1}) - Q(s_k, a_k))\tilde{\phi} \tag{5.8}$$

Convergence of this method can be slow because we are limiting the effect of an update on some parameters. This is especially the case for values of γ close to 1. In order to make the algorithm faster, the constraint on the gradient can be relaxed somewhat. For this, in addition to *gradient residual* algorithm, Baird also introduced the *residual* algorithm. This includes a factor ζ in [0,1]. The resulting residual gradient calculation is the following:

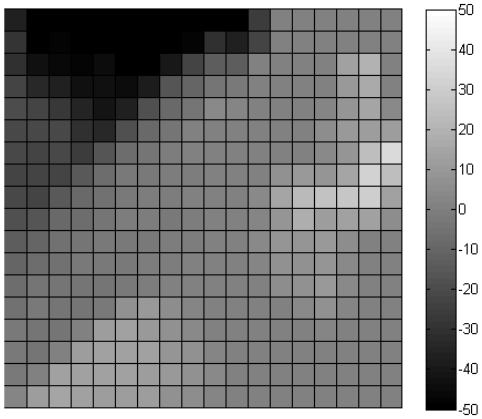
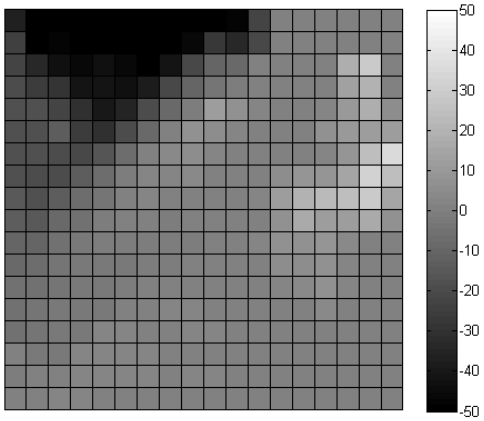
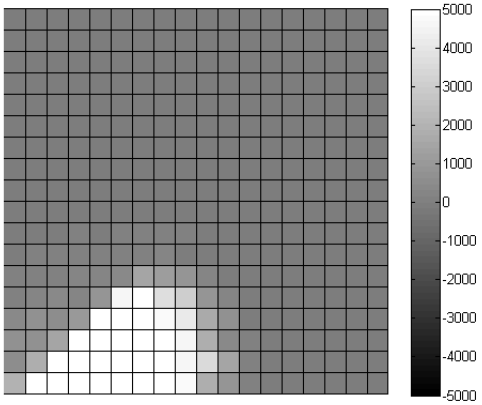
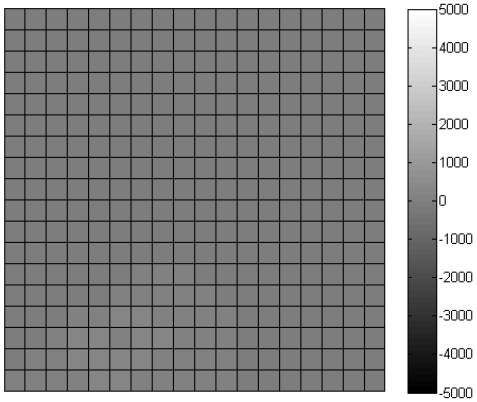
$$\tilde{\phi} = \phi(s_k, a_k) - \zeta\gamma\phi(s_{k+1}, a_k) \tag{5.9}$$

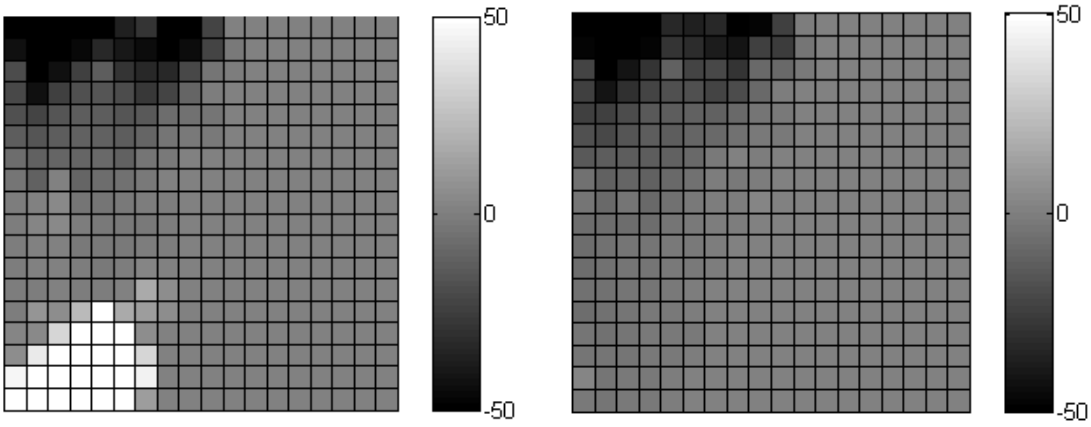
This factor can either be tuned or calculated. Calculating ζ involves storing two more value functions and introducing another tunable parameter however. Because the use of two more value function would negatively affect the computational load and because tunable parameters are best kept minimized in RL, this method is omitted in this thesis. For further details on this, the reader is directed to (Baird, 1995).

The residual parameter update rules defined by Equations 5.7 and 5.8 were combined with ER, yielding residual gradient ER. This algorithm is still very similar to vanilla-ER but updates on the value of $Q(s_k, a_k)$ will effect the value $Q(s_{k+1}, a)$ only by a very small amount. Algorithms 11 and 12 in Appendix C show this ER algorithm in pseudo code.

Baird showed that for on-line learning, residual gradient updates prevented divergence of the Q-values on his problem. In our case, we expect that residual updates will prevent or at least reduce the emergence of local maxima. To verify this, the new algorithm was applied on the grid world. Table 5.2 shows typical value functions of vanilla-ER and residual gradient ER when applied on the same sample database and using the exact same number of replays as was done to yield the value functions of Figure 5.2. It can be observed that with residual gradient ER, there are no strong maxima in the bottom left corner. It is unclear from this however, how residual gradient updates will effect learning speed. In the next chapter the results of residual gradient ER on

Table 5.2: Typical value functions of applying ER and the new residual ER algorithm to the grid world. Local maxima are less likely to occur and to grow to extremely large values. Per row, the same data set is used and the exact same samples are replayed. Old samples were replayed a relatively long time without gathering new samples.

′vanilla′-ER	residual gradient ER
	
	



(a) ER-σ.

(b) residual ER-σ

Figure 5.7: Typical value functions of applying all ER combinations to the grid world. Residual ER-σ shows no growing local maximum. Using residual gradient might yield higher approximation errors however. For both cases, the same data set is used and the exact same samples are replayed and old samples were replayed a relatively long time without gathering new samples.

the benchmark problems will be presented. We hypothesize that self-affecting states are the cause of the observed 'standing still' behavior of LEO. Therefore, we expect that combining this method will increase the performance of the learning algorithm.

Summarizing, self-affecting states can lead to erroneous and rising Q-values. In the grid world this was visible as local optima in the value function. In this section, we propose to combine residual gradient with experience replay. On the grid world, this diminishes growing maxima visibly. We expect that this will improve learning speed. On-line residual algorithms are known to have a much slower convergence however. Experiments need to show whether this is the case with ER as well.

5.4 Residual ER-σ

Residual gradient ER and ER-σ can be combined in one algorithm, residual ER-σ. Algorithms 13 and 14 in Appendix C show this ER algorithm in pseudo code. Figure 5.7 shows typical value functions of all algorithms proposed in this chapter with the sample sample database and the exact same replays. Residual ER-σ shows no sign of a growing local maximum which is visible with ER-σ alone.

Chapter 6

Results and discussion

We verified the performance of the new algorithms using the same benchmark problems as in Chapter 4, the inverted pendulum, the simplest walker and LEO. In this chapter, the results of applying the algorithms proposed in the previous chapter (ER-σ, residual gradient ER and residual ER-σ) will be presented. For comparison, the results of Sarsa(λ) and vanilla-ER will be presented as well. After this, a discussion of the results will follow. The ER algorithms used in this chapter can be found in Appendix C.

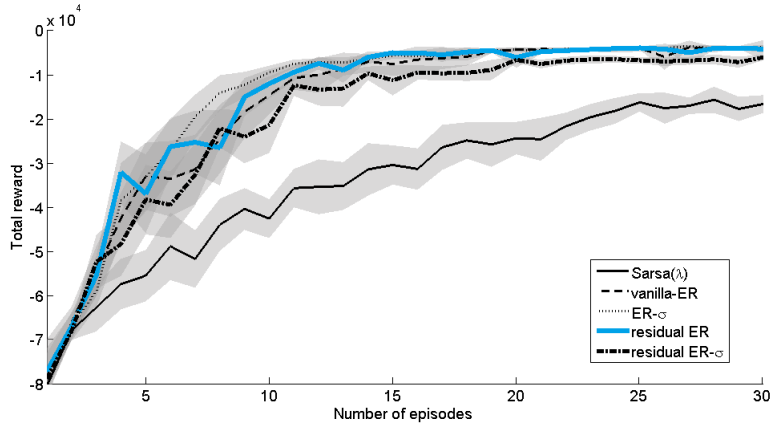
6.1 Simulation results

Figures 6.1 and 6.2 show the simulation results in the form of graphs. Figure 6.1 shows the performance of the learning algorithms against learning time. Like before, on the inverted pendulum this is represented by the total accumulated reward. The performance on the simplest walker and LEO is represented by the traveled distance over 100 and 25 seconds respectively. Figure 6.2 show the cumulative number of falls during the learning process on the simplest walker and LEO. Each graph shows the average of 20 runs with a 95% confidence interval of the average.

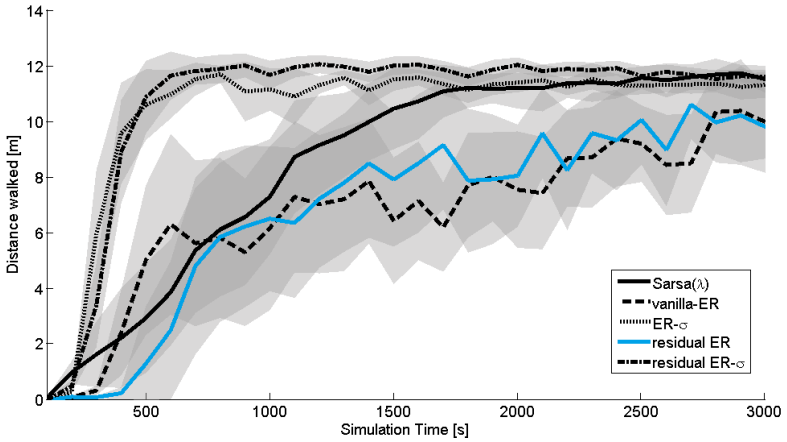
Tables 6.1, 6.2 and 6.3 list the results numerically per benchmark problem. The tables list the rise time, end performance and the number of falls if applicable. The rise time shows how long it takes to get to 90% of the end performance. The end performance is defined as the highest performance after a successful gait has been learned.

The inverted pendulum

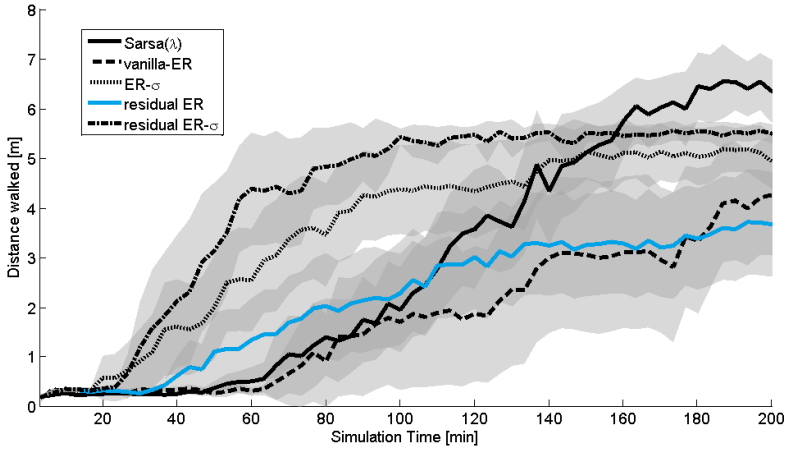
One can observe from Figure 6.1a and Table 6.1 that the performance of the new algorithms do not significantly differ from vanilla-ER on the inverted pendulum. ER-σ is slightly faster than the other ER algorithms with a reduction of the rise time of 18% with respect to vanilla-ER. When combined with residual ER however, performance drops slightly for unknown reasons. It is clear that all ER-algorithms still show a much better performance than Sarsa(λ). Additionally, it can be noted that the residual variant of vanilla-ER does not result in a significant difference in performance.



(a) The inverted pendulum

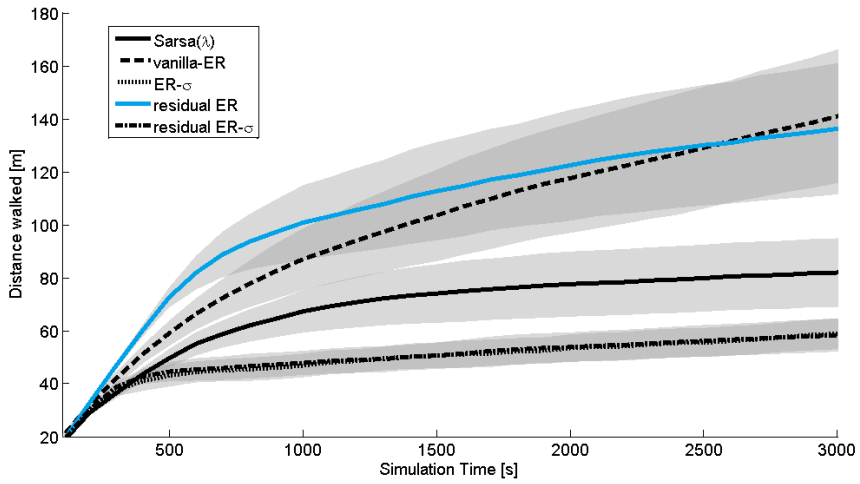


(b) The simplest walker

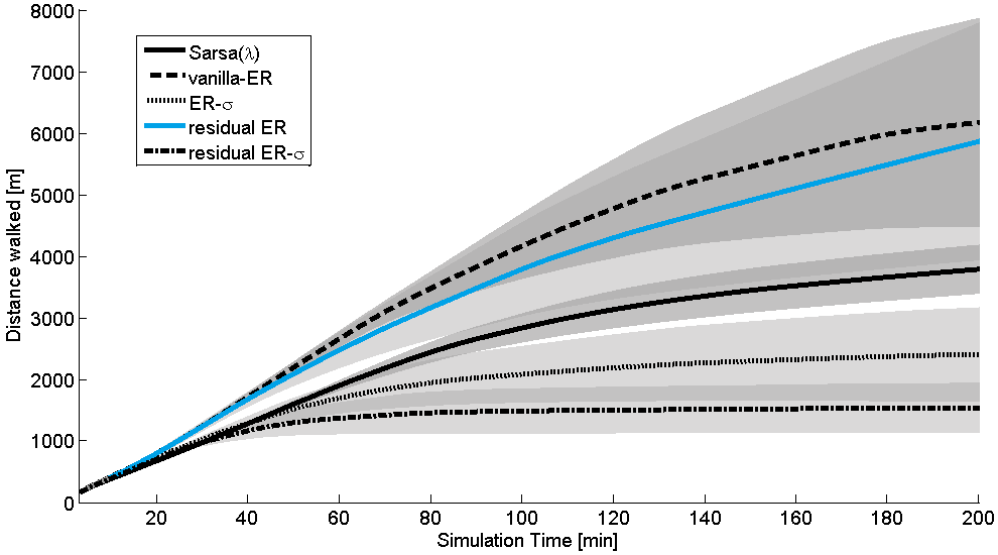


(c) LEO

Figure 6.1: Simulation results of the inverted pendulum, the simplest walker and LEO. The lines represent the average of 20 runs, the shaded areas represent the 95% confidence interval of the average.



(a) The simplest walker



(b) LEO

Figure 6.2: Number of falls on the simplest walker and LEO against learning time. The lines represent the average of 20 runs, the shaded area represent the 95% confidence interval of the average.

	Metric	Average	Difference with Sarsa(λ)
Sarsa(λ)	Rise time	22	-
	End performance	-1.673·10^4	-
vanilla-ER	Rise time	11	-50 %
	End performance	-4065	+76%
ER-σ	Rise time	9	-60 %
	End performance	-3692	+78%
residual ER	Rise time	10	-55 %
	End performance	-3913	+77%
residual ER-σ	Rise time	11	-50 %
	End performance	-6200	+63%

Table 6.1: Results of the inverted pendulum. Rise time is in number of episodes and end performance is the accumulated reward.

	Metric	Average	Difference with Sarsa(λ)
Sarsa(λ)	Rise time	1500	-
	End performance	11.7	-
	Falls	74	-
vanilla-ER	Rise time	3400	+126 %
	End performance	11.7	0%
	Falls	150	+51%
ER-σ	Rise time	500	-67 %
	End performance	11.5	-2%
	Falls	44	-41 %
residual ER	Rise time	3400	+126%
	End performance	11.7	0%
	Falls	140	+48%
residual ER-σ	Rise time	500	-67 %
	End performance	12.0	+3%
	Falls	45	-39 %

Table 6.2: Results of the simplest walker. Rise time is in seconds and end performance in meters.

	Metric	Average	Difference with Sarsa(λ)
Sarsa(λ)	Rise time	160	-
	End performance	6.5	-
	Falls	3792	-
vanilla-ER	Rise time	>200	>+25%
	End performance	NA	-
	Falls	>6170	>+93%
ER-σ	Rise time	133	-17 %
	End performance	5.2	-20%
	Falls	2416	-36 %
residual ER	Rise time	>200	>+25%
	End performance	NA	-
	Falls	>5862	>+93%
residual ER-σ	Rise time	87	-46 %
	End performance	5.5	-15%
	Falls	1550	-59 %

Table 6.3: Results of LEO. Rise time is in minutes and end performance is in meters.

The simplest walker

It can be seen from Figure 6.1b and Table 6.2 that ER-σ and residual ER-σ perform much better than Sarsa(λ) and vanilla-ER on the simplest walker. The rise time has decreased by 67% percent for both with respect to Sarsa(λ). For both algorithms, the number of falls that occurred during the learning process decreased significantly as well: a decrease of approximately 40% with respect to Sarsa(λ).

Looking at the performance of the residual algorithms, we can note that residual ER-σ shows a slightly better end performance than ER-σ. Additionally, we can see that residual gradient ER has a similar performance as vanilla-ER. The performance gain resulting from ER-σ clearly stands out however: when compared with vanilla-ER, it is almost 7x faster on average.

LEO

Looking at the results, we can see that ER-σ and residual ER-σ perform significantly better than Sarsa(λ) and vanilla-ER on LEO as well (Figure 6.1c and Table 6.3). The rise time has decreased by 17% percent for ER-σ and 46% for residual ER-σ with respect to Sarsa(λ). The number of falls decreased for both algorithms as well; down with 36% and 59% respectively.

Furthermore, it can be observed that the end performance of all ER algorithms is lower than that of Sarsa(λ). Remarkably, all ER algorithms found a different (slightly slower) gait then Sarsa(λ), on all runs. With ER, one foot was always kept in front of the other. Walking was done by moving the rear foot to the spot right behind the front foot, after which the front leg makes a step whilst standing on the rear foot.

Looking at the performance of the residual algorithms, there is a remarkable im-

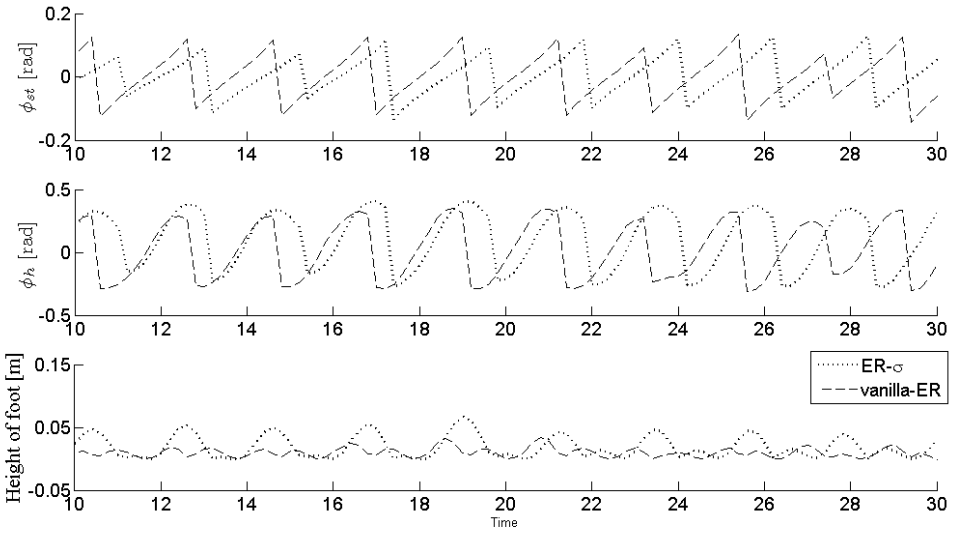


Figure 6.3: Typical learned evolution of the angle of the stance leg, the hip joint and of the height of the foot of the simplest walker . One can observe that the policy following from ER-σ lifts its swing foot higher from the ground, indicating a safer behavior.

provement when using the residual variant of ER-σ. We can observe that residual ER-σ learns faster and with less falls non-residual counterpart: the residual variant of ER-σ yields a rise time decrease of 35% and a decrease in the number of falls of 36% with respect to its non-residual counterpart. Furthermore, we can note that although the residual variant of vanilla-ER initially shows a better performance, it does not result in a big improvement overall.

6.2 Discussion

Looking at the results, we can see that the performance gain resulting from ER-σ clearly stands out. From this, we can conclude that slow and unpredictable learning of ER can indeed be largely attributed to the replay of uncertain Q-values. We can see that performance of ER is greatly affected by the attitude of the algorithm is towards these uncertainties.

In Figure 6.3 typical behaviors of the simplest walker following from vanilla-ER and ER-σ are shown. As can be seen, the policy following from ER-σ lifts its swing foot higher than vanilla-ER i.e. a safer route is taken with ER-σ. From this we can conclude see that ER-σ indeed yields a safer policy. This can be contributed to the fact that the agent actually learns from falling, i.e. it learns to avoid unsafe areas. For walking problems this strategy results in a significant performance gain.

From the results, we can clearly see that the improvement of performance of ER-σ with respect to vanilla-ER is most prominent on walking problems. This has several

reasons:

Failing trajectories very close to successful trajectories The pendulum problem does not contain failing trajectories which reside very close to successful ones. On the pendulum, slight deviations of a successful policy are still likely to result in a high performance. It is therefore less problematic if uncertain (generalized) Q-values are back-propagated. Because of this, being optimistic in the face of uncertainty does therefore not harm performance to a great extent. Whereas with walking, slight deviations from an successful trajectory might immediately result in a failed episode. Accepting uncertain Q-values resulting from generalization is therefore more problematic.

High dimensionality Sample efficient learning on high-dimensional systems means the agent has to learn from an incomplete database. The higher the dimensionality, the more incomplete the database likely is. That is, with high dimensional systems, large regions of the state action-space may not be covered by samples since gathering these samples will take too long. The more incomplete the database, the more uncertain Q-value will be encountered during replay. So with high dimensional systems, performance of the learn algorithm is more effected by the attitude towards uncertainty.

Box rewards With box-rewards, the algorithm has to fully rely on back-propagation to learn. If unsafe or unwanted areas are built into the reward function, less reliance on back-propagation is needed to avoid these areas. For instance with the simplest walker, a slight reward can be given to promote lifting the foot. Since we are supplying information on how to execute its task, this strategy would involve using prior knowledge however.

As stated earlier, Baird (1995) reports significantly slower learning with residual algorithms. We have shown that with sample re-use, this is not necessarily the case with ER. The residual algorithms all performed equal or better than their non-residual counter parts. The main reason why on-line residual algorithms are slow, is that the gradient decent step of an update can be very small i.e. the effect of an update on the parameters can be very small. With on-line algorithms, a new update means gathering a new sample. With sample re-use we do not have this restriction and can use samples an infinite amount of times. From this we can conclude that residual algorithms are not necessarily slow in terms of the number of samples, but rather in terms of the number of updates.

From the results on LEO, we observed that the end performance resulting from ER was lower than that of Sarsa(λ). This is due to the fact that ER results in a different gait than Sarsa(λ). In other words, ER finds a different maximum than Sarsa(λ). This could be caused by the reward function, which was specifically tailored for Sarsa(λ). The fact that Sarsa(λ) find a good gait with this reward function gives no guarantee that other method will do so as well. In fact, with a finite amount of samples, there is no guarantee that sample re-use algorithms will find an optimal solution at all. Because the given set of data is finite, the agent cannot be expected to always come up with an optimal policy

(Lange et al., 2012).Another cause could be that the gait found by ER needs less data in order to be derived and is therefore found before the gait found by Sarsa(λ).

As we have seen in the results, the residual ER-σ algorithm shows a remarkable performance increase on LEO. Additionally, inspecting the behavior of the robot during learning, we observed LEO's behavior of standing still and balancing a lot less during learning for both residual algorithms. This indicates that self-affecting states indeed do negatively influence performance on some problems. Looking at the results, we can see that this can be solved by making residual gradient updates. However, it is at this point not fully understood why the results are more distinct on LEO than on the other problems. We hypothesize that this effect is due to the high dimensionality of LEO, and the fact that the robot can balance or stand still, yielding a high amount of possible self-affecting states. The high dimensionality of LEO may result a high number of missing samples which are needed to compensate for the increase of $Q(s', a)$ at an update on $Q(s, a)$.

The performance of residual ER-σ on the simplest walker and LEO are very promising. Indeed, the number of sample needed to learn can be brought down significantly with respect to Sarsa(λ). However, we showed that although ER is a promising technique, it gave rise to some unforeseen issues. Additionally, ER gives no guarantee on learning speed or end performance. In this light, we can note that Sarsa(λ) has the following advantage over ER: because of the Monte Carlo character of this algorithm, it can handle uncertainties better than ER. That is, Sarsa(λ) evaluates every $Q(s', a')$-value is makes. The results of this evaluation is coupled back through the use of the eligibility trace.

Finalizing, it is interesting to note that the issues discussed in this thesis can ultimately be attributed to having limited data. If an infinite amount of data would be available, every state-action combination would be visited. Because of this, the database would contain information on every state-action combination possible and therefore, the algorithm will base its decisions on actual information of the environment. Additionally, state self-affection will not be as troublesome because any unwanted effects on $Q(s', a)$ at an update on $Q(s, a)$ can be corrected. This can be done since we will have a sample at (s', a) at our disposal.

Furthermore, we can state that a good representation of the value function for the given data and the given problem can some a lot of problems as well. For instance, one can imagine a representation for the given data so that there are no self-affecting states. Or, one can imagine a representation in which generalization of Q-values of successful state-action combinations might be prohibited to state-action combinations which are known to lead to failure. Realizing such representations is difficult however and often requires prior knowledge.

Several other studies report problems with sample re-use caused by imperfect representation. Fonteneau et al. (2012) demonstrated that sub optimal policies resulting from FQI can be attributed to function approximation. In this study it was proposed that the use of function approximation can be avoided altogether by using artificial trajectories . Kalyanakrishnan and Stone (2011) stated that the success of sample re-use techniques relies heavily on having a good representation. They argue that this causes a tension

between sample efficiency on the one hand and resilience to imperfect representations on the other. The approach proposed in this study was to focus on learning a representation parallel to learning a policy.

Chapter 7

Conclusions and future work

7.1 Summary and conclusions

Service robots are expected to become increasingly important in the near future. The main reason these robots have not entered our daily live yet is that service robots generally perform a large variety of tasks in unknown and changing environments. This makes manually programming of such robots difficult. Letting robots learn tasks through interaction with the environment therefore becomes an attractive alternative.

An important learning paradigm for robots is *Reinforcement Learning* (RL). RL can solve complex problems without requiring any prior knowledge on how to solve the problem or by making any restricting assumptions about the environment the learner is in. Having real robots learn remains a difficult challenge however. This mainly because of the inherent high dimensionality of robots, time variant dynamics, hardware wear, limited computational and memory capabilities. To meet these challenges, a robot called LEO has been developed in the Delft BioRobotics Lab

LEO is a 2D biped robot built to learn to walk through RL. When learning from scratch however, the robot breaks down before it has learned a successful gait. Most of the hardware damage can be attributed to varying torques applied in a high frequency and to falls of the robot during learning. A possible solution to this problem would be to have LEO learn *before* it breaks down. This can be done by minimizing the number of interactions with the environment needed to learn a satisfactory policy.

In RL, each interaction with the environment is called a sample. An important and promising way of reducing the number of samples needed is to re-use samples instead of discarding them after using them once. This thesis studied the re-use of old samples in order to yield a low sample complexity for walking robots.

In Chapter 3 the most important sample re-use techniques were discussed and a novel framework was presented to study their respective properties. Within this framework, each of the discussed sample re-use method is considered as a special case of a general algorithm. The framework illustrated that sample re-use methods differ in two major aspects:

The value function update The value function can be updated in two distinctive

ways: by gradient descent steps on the error following from *single samples* or by minimizing the error following from all samples in one *projection*. Using projection updates can solve stability issues of the value function during learning. This way of updating can also negatively affect the computational complexity however, depending on function approximator used.

Composition of sample database The sample database can be composed in several ways with sample re-use: it can contain either all or a selection of previous samples and in addition, artificial samples might be included following from a trained model. It is difficult to determine a suitable sample database before hand. Sometimes using only recent samples might give better results. For some (real world) problems one should approach carefully when using model learning because approximating samples to approximate a value function can introduce serious errors.

Experience Replay (ER) was chosen to demonstrate and investigate the performance of sample re-use techniques on walking problems. The choice for this method has been made for several reasons. Namely, analysis is relatively easy with this method, ER has high control over computational complexity and has shown to combine well with the function approximation already used on LEO. In the variant of ER used in this thesis, the old samples are presented in the form of reversed trajectories and are used to improve the value function by means of the Sarsa update rule.

Chapter 4 provided an empirical analysis of ER by comparing it with Sarsa(λ). In order to evaluate the performance of ER, three benchmark problems were used: the inverted pendulum, the simplest walker model and LEO . ER on the simplest walker and LEO proved not very sample efficient compared to Sarsa(λ). Inspecting the behavior of the simplest walker, one could observe that ER resulted an unsafe policy. Additionally, with LEO it was observed that sometimes during learning, the robot tended to stand still or balance without any intention of walking forward.

In Chapter 5, a grid world was presented to analyze these issues. Typical value functions during learning of the grid world problem showed that punishments were not back-propagated with ER. Additionally, in areas which were bound to result in failure, the value function often showed a positive expected return. Furthermore, we observed that sometimes the value function showed growing local maxima.

We argued that failing back-propagation of punishments is a result of uncertain Q-values used in updates. With linear function approximation, initialization of the value function determines the attitude towards uncertainty of these values: if the value function is initialized with relatively high values, the algorithm will favor uncertain state-action pairs over values with a low expected return. While this effect is useful for effective exploration, during replay of old samples this results in not learning from punishments and ultimately, unsafe policies. It was hypothesized that optimism in safe areas and pessimism in unsafe areas would would solve these issues and increase learning speed.

A new algorithm was derived, ER-σ, which makes the attitude towards uncertainty dependent of the state of the to-be-used Q-value. With this algorithm, samples will effectively be replayed with a $Q(s', a')$-value in which the unvisited parameters have been replaced with the value of the function $\sigma(s')$. $\sigma(s')$ can be used to exploit the

information available to us in state s'. The exact value of $\sigma(s')$ can be determined by considering the outcomes of visited actions in s'. We showed that for the simplest walker, there is an optimum between assuming the worst case action and assuming the average outcome of the actions.

In the grid world, ER-σ clearly resulted in proper back-propagation of punishments. Additionally, positive rewards in safe areas were back-propagated through a larger portion of the state-action space with respect to vanilla-ER. Additionally, from results on the benchmark problems we observed that ER-σ yielded better performance than both Sarsa(λ) and vanilla-ER on all three benchmark problems, most notably on the walking problems. On the simplest walker we observed a safer policy following from ER-σ when compared with the policy following from vanilla-ER. From these results, we can conclude that the observed unsafe policies were indeed a result of being optimistic in the face of uncertainty. In the discussion, it was argued that this is particularly important for walking problems.

Chapter 5 additionally investigated local optima emerging in the value function. We attribute this effect to self affecting states. These are close subsequent states that, when they are updated, effect their own target value for the next update. This can create erroneous and rising Q-values. In on-line learning, this effect was explored by (Baird, 1995). In this study, *residual algorithms* were introduced. We combined the residual gradient algorithm by Baird (1995) with ER to yield residual gradient ER. In the grid world problem, this reduces local optima visibly.

From the results on all benchmark problems, we observed that residual gradient ER showed no significant improvement with respect to vanilla-ER however. Combining residual updates with ER-σ did result in a significant improved performance on LEO and a slightly better end performance on the simplest walker. From these results, we can conclude that residual updates at least do not negatively influence performance with ER. This stands in contrast with the common knowledge that residual algorithms can result in very slow convergence in on-line learning methods. Additionally, we can conclude that self-affecting states indeed do negatively influence performance of ER in some circumstances. The results suggest that residual updates can be used to solve this. From the results it is difficult to draw conclusions on the exact circumstances or type of problem which benefit from residual updates however.

The simulation results also showed that the end performance of LEO is a bit lower when learning with ER than when learning with Sarsa(λ). This can be a result of the reward function which was specifically tailored for Sarsa(λ). It might be that for sample re-use techniques a more suitable reward function exists. But still, this would give no guarantee of a good end performance.

Even though the end performance is slightly lower, the performance of residual ER-σ on the simplest walker and LEO is very promising. This shows that the number of samples needed to learn can be brought down significantly with respect to Sarsa(λ) on walking robots. However, we have seen that the performance of ER is negatively effected by some unforeseen issues issues. Consequently, one has to be aware of the consequences of having limited data and an imperfect representation when using ER. Several studies suggest that this holds for other sample use techniques as well (Kalyanakrishnan and

Stone, 2011; Fonteneau et al., 2012)

In summary, this thesis provided a theoretical comparison of sample re-use techniques in the form of a unified framework. From this, ER was selected to be used for evaluation and analysis. From our initial experiments, we have seen that ER can result in slow and unpredictable learning when compared with Sarsa(λ). There are indeed no guarantees that ER will always yield a good performance. This thesis showed that by exploiting information in the available data and using knowledge of the representation,the performance of ER can be increased significantly. Doing this yielded very promising results on LEO with respect to Sarsa(λ). Whether this is enough to have LEO learn without mechanical failure remains to be seen from experiments on the real robot.

7.2 Future work

We propose the following improvements or future research directions:

- In this thesis, residual ER-σ and ER-σ yielded promising results. Whether this is enough to have LEO learn before breaking down must turn out from future experiments on the real robot however. Currently, LEO breaks down as fast as 5 minutes and learning with the highest performing algorithm (residual ER-σ), still takes longer than that. However, in recent research at the Delft BioRobotics Lab (Meijdam et al., 2013) showed that the damage on LEOs gear boxes can be reduced significantly by letting the selection of the current action be a function of the last taken action. Combining this method with sample re-use could let LEO succeed in learning without mechanical failure.

- Some opportunities to further increase learning speed of ER might be through prioritized sweeping (Moore and Atkeson, 1993) or database pruning. Additionally, further efficiency might be achieved by using an Expected Sarsa (van Seijen et al., 2009) variant of ER. Expected Sarsa does not require the averaging effect when making on-policy updates following a stochastic policy. Therefore, the learning rate can be set to 1. This might enable faster learning.

- From the literature it follows that the performance of sample re-use is strongly dependent on the representation of the value function (Kalyanakrishnan and Stone, 2011; Fonteneau et al., 2012). The analysis and evaluation throughout this thesis was done with binary tile coding function approximation. An important question is to whether the issues discussed in this thesis occur with other types of function approximation. If they do, a next step would be to investigate whether the solutions proposed in this thesis be used with other function approximators. Moreover, one could wonder whether there is a general way of exploiting data in order to yield better and more consistent results with any type of representation.

- From the results in this thesis we have concluded that residual updates can improve learning speed on some problems. Future work is needed however to clarify on what specific problems or in which circumstances residual updates increases learning speed. For this, more and a higher variety of experiments are needed.

Appendices

Appendix A

Efficiency of replay techniques

To test the effectiveness of each replay method, the following experiment was done: First, a database of samples was collected with the Sarsa(λ) algorithm until 200 episodes had been completed. This is well beyond the learning time of the algorithm to ensure there are enough near-optimal trajectories. After this, ER with separate samples, trajectories and reversed trajectories was used on the database, each starting with newly initialized value function. After every 75,000 sample replays, the performance is evaluated. Since the different algorithms all learn from the same database, this will indicate the effectiveness of the algorithm per sample replayed.

The above is executed 20 times. Each time with a new database was collected by Sarsa(λ). The results are shown in figure A.1.

From the results, we can see that replaying trajectories in temporal order has the lowest performance gain per replayed sample. At 225,000 replays the performance of reversed trajectories is slightly better then that of single samples.

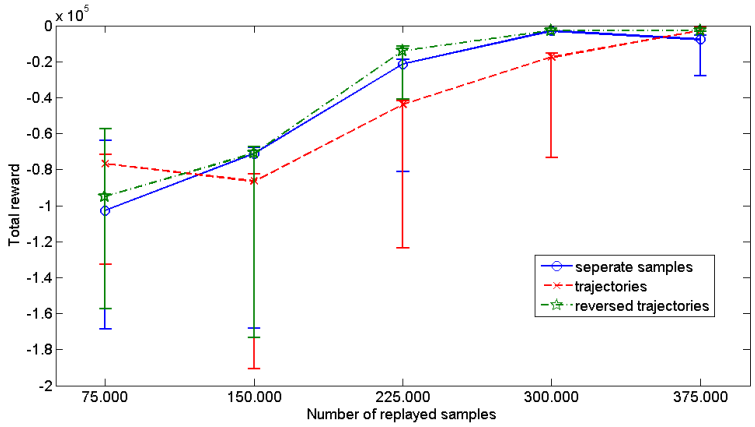


Figure A.1: Performance on the pendulum problem against the number of replays of the three replay methods. The error bars indicate the 95% confidence interval of the total reward of an evaluation run.

Appendix B

$\sigma(s')$ for walking problems

To find a suitable function $\sigma(s')$ for walking problems, the following experiment was done: ER with different definitions of $\sigma(s')$ was used on the simplest walker. Each 100 seconds of learning time, the walked distance was evaluated. The performance of specific definition of $\sigma(s')$ is defined as the average of each recorded walking distance. 20 runs were done for each case.

The definitions of $\sigma(s')$ include the following:

1. The worst case scenario: $\sigma(s') = \min_a (Q(s', a)/W(s', a))$.
2. The average over the visited actions: $\sigma(s') = \sum_a Q(s', a) / \sum_a W(s', a)$.
3. The best case scenario: $\sigma(s') = \max_a (Q(s', a)/W(s', a))$.
4. Interpolation between the worst case and the average.
5. Interpolation between the average and the best case.

Note that for some actions $W(s', a)$ may be 0 and thus $Q(s', a)/W(s', a)$ cannot be calculated. In this case that particular state-action pair is not visited and will not be included in the calculation of σ. Fortunately, since we are making trajectories through the state-action space, there always is at least one visited state-action pair available.

The above options are not exhaustive, these methods have been selected because they are simple operations ranging from worst cast to bad case and contain no tunable parameters. Assuming the worst case will put the emphasis on avoiding unsafe areas, assuming the best case will put the emphasis on following positive rewards.

Figure B.1 presents the results. As can be seen, the interpolation between the worst case and the average performs the best. Furthermore, the performance greatly drops as we move towards assuming the best case. Following these results, we shall be using the 4-th definition of σ in the enumeration above throughout this thesis.

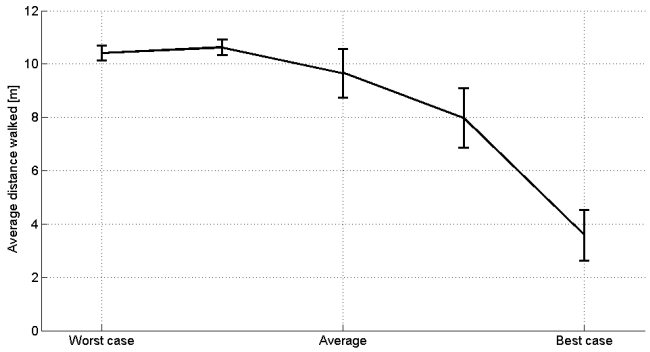


Figure B.1: Performance on the pendulum problem against the number of replays of the three replay methods. The error bars indicate the 95% confidence interval of the total reward of an evaluation run.

Appendix C

Algorithms

C.1 Experience Replay

Algorithm 7: Main ER

Input : number of replays K, discount factor γ, learning rate α, exploration rate ϵ

$\theta \leftarrow \mathbf{0}$;
initialize w as all zero vector with same size as θ: $w \leftarrow \mathbf{0}$;
$\mathcal{D} \leftarrow \emptyset$;
for *every episode* e **do**
    $k \leftarrow 0$;
    observe initial state s_0;
    for *every time-step* k **do**
        $a_k \leftarrow \begin{cases} \arg\max_a \phi(s_k, a)^T \theta & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$;
        execute a_k and observe s_{k+1} and r_{k+1};
        $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup \{(s_k, a_k, s_{k+1}, r_{k+1})\}$;
    end
    $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup k$;
    $\theta \leftarrow learnER(\theta, \mathcal{D}, K, \gamma, \alpha, \epsilon)$;
end
output: θ

Algorithm 8: LearnER

Input : θ, $\mathcal{D}$, number of replays K, discount factor γ,learning rate α, exploration rate ϵ

for $i = 1$ *to* K **do**
  select a random trajectory e' uniformly distributed over $\{1, \ldots, e\}$;
  draw trajectory length k from $D_{e'}$;
  for $t = k$ *to* 1 **do**
    draw sample $(s_t, a_t, s_{t+1}, r_{t+1})$ from $\mathcal{D}_{e'}$;
    $a_{t+1} \leftarrow \begin{cases} \arg\max_a \phi(s_{t+1}, a)^T \theta & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$;
    $\theta \leftarrow \theta + \alpha \left(r_{t+1} + \gamma \phi(s_{t+1}, a_{t+1})^T \theta - \phi(s_t, a_t)^T \theta \right) \phi(s_t, a_t)$;
    $t \leftarrow t - 1$;
  end
end

Output: θ

C.2 ER-σ

Algorithm 9: Main ER-σ

Input : number of replays K, discount factor γ, learning rate α, exploration rate ϵ

$\theta \leftarrow \mathbf{0}$;
initialize w as all zero vector with same size as θ: $w \leftarrow \mathbf{0}$;
$\mathcal{D} \leftarrow \emptyset$; $l \leftarrow 1$;
for *every episode* e **do**
  $k \leftarrow 0$;
  observe initial state s_0;
  for *every time-step* k **do**
    $a_k \leftarrow \begin{cases} \arg\max_a \phi(s_k, a)^T \theta & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$;
    execute a_k and observe s_{k+1} and r_{k+1};
    $w_k \leftarrow w_k \vee \phi(s_k, a_k)$;
    $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup \{(s_k, a_k, s_{k+1}, r_{k+1})\}$;
  end
  $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup k$;
  $\theta \leftarrow learnER\sigma(\theta, w, \mathcal{D}, K, \gamma, \alpha, \epsilon)$;
end

Output: θ

Algorithm 10: learnERσ

Input : θ,w, $\mathcal{D}$, number of replays K, discount factor γ,learning rate α, exploration rate ϵ

for $i = 1$ *to* K **do**
  select a random trajectory e' uniformly distributed over $\{1, \ldots, e\}$;
  draw trajectory length k from $D_{e'}$;
  for $t = k$ *to 1* **do**
    draw sample $(s_t, a_t, s_{t+1}, r_{t+1})$ from $\mathcal{D}_{e'}$;
    $\sigma \leftarrow \left(\frac{\sum_a \phi(s_{t+1},a)^T \theta}{\sum_a \phi(s_{t+1},a)^T w} + \min_a \left(\phi(s_{t+1}, a)^T \theta) / \phi(s_{t+1}, a)^T w) \right) \right) /2$;
    $u(a) = \frac{\sum_i \phi_i(s_{t+1},a) - \phi(s_{t+1},a)^T w}{\sum_i \phi_i(s_{t+1},a)}$;
    $a_{t+1} \leftarrow \begin{cases} \arg\max_a \left(\phi(s_{t+1}, a)^T \theta + u(a) \sigma \sum_i \phi_i(s_{t+1}, a) \right) & \text{w.p. } 1 - \epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$
    $\delta \leftarrow r_{t+1} + \gamma \left(\phi(s_{t+1}, a_{t+1})^T \theta + u(a_{t+1}) \phi(s_{t+1}, a_{t+1})^T \mathbf{1} \sigma \right) - \phi(s_t, a_t)^T \theta$;
    $\theta \leftarrow \theta + \alpha \delta \phi(s_t, a_t)$;
    $t \leftarrow t - 1$;
  end
end

Output: θ

C.3 Residual gradient ER

Algorithm 11: Main residual gradient ER

Input : number of replays K, discount factor γ, learning rate α, exploration rate ϵ

$\theta \leftarrow \mathbf{0}$;
initialize w as all zero vector with same size as θ: $w \leftarrow \mathbf{0}$;
$\mathcal{D} \leftarrow \emptyset$; $l \leftarrow 1$;
for *every episode* e **do**
  $k \leftarrow 0$;
  observe initial state s_0;
  for *every time-step* k **do**
    $a_k \leftarrow \begin{cases} \arg\max_a \phi(s_k, a)^T\theta & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$ execute a_k and observe s_{k+1} and r_{k+1};
    $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup \{s_k, a_k, s_{k+1}, r_{k+1}\}$;
    $k \leftarrow k+1$;
  end
  $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup k$;
  $\theta \leftarrow learnresidualER(\theta, \mathcal{D}, K, \gamma, \alpha, \epsilon)$;
end
output: θ

Algorithm 12: learnresidualER

Input : θ, $\mathcal{D}$, number of replays K, discount factor γ,learning rate α, exploration rate ϵ

for $i = 1$ *to* K **do**
  select a random trajectory e' uniformly distributed over $\{1, \ldots, e\}$;
  draw trajectory length k from $D_{e'}$;
  for $t = k$ *to 1* **do**
    draw sample $(s_t, a_t, s_{t+1}, r_{t+1})$ from $\mathcal{D}_{e'}$;
    $a_{t+1} \leftarrow \begin{cases} \arg\max_a \phi(s_t, a)^T\theta & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$;
    $\theta \leftarrow \theta + \alpha\left(r_{t+1} + \gamma\phi(s_{t+1}, a_{t+1})^T\theta - \phi(s_t, a_t)^T\theta\right)(\phi(s_k, a_k) - \gamma\phi(s_{k+1}, a_k))$;
    $t \leftarrow t-1$;
  end
end
Output: θ

C.4 Residual ER-σ

Algorithm 13: Main residual ER-σ

Input : number of replays K, discount factor γ, learning rate α, exploration rate ϵ

$\theta \leftarrow \mathbf{0}$;
initialize w as all zero vector with same size as θ: $w \leftarrow \mathbf{0}$;
$\mathcal{D} \leftarrow \emptyset$; $l \leftarrow 1$;
for *every episode* e **do**
  $k \leftarrow 0$;
  observe initial state s_0;
  for *every time-step* k **do**
    $a_k \leftarrow \begin{cases} \arg\max_a \phi(s_k, a)^T \theta & \text{w.p. } 1-\epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$;
    execute a_k and observe s_{k+1} and r_{k+1};
    $w_k \leftarrow w_k \vee \phi(s_k, a_k)$;
    $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup \{s_k, a_k, s_{k+1}, r_{k+1}\}$;
    $k \leftarrow k + 1$;
  end
  $\mathcal{D}_e \leftarrow \mathcal{D}_e \cup k$;
  $\theta \leftarrow learnresidualER\sigma(\theta, w, \mathcal{D}, K, \gamma, \alpha, \epsilon)$;
  $t \leftarrow t - 1$;
end
output: θ

Algorithm 14: learnresidualERσ

Input : θ,w, $\mathcal{D}$, number of replays K, discount factor γ,learning rate α, exploration rate ϵ

for $i = 1$ *to* K **do**
  select a random trajectory e' uniformly distributed over $\{1, \ldots, e\}$;
  draw trajectory length k from $D_{e'}$;
  for $t = k$ *to 1* **do**
    draw sample $(s_t, a_t, s_{t+1}, r_{t+1})$ from $\mathcal{D}_{e'}$;
    $\sigma \leftarrow \left(\frac{\sum_a \phi(s_{t+1},a)^T \theta}{\sum_a \phi(s_{t+1},a)^T w} + \min_a \left(\phi(s_{t+1}, a)^T \theta) / \phi(s_{t+1}, a)^T w) \right) \right) /2$;
    $u(a) = \frac{\sum_i \phi_i(s_{t+1},a) - \phi(s_{t+1},a)^T w}{\sum_i \phi_i(s_{t+1},a)}$;
    $a_{t+1} \leftarrow \begin{cases} \arg\max_a \left(\phi(s_t, a)^T \theta + u(a) \sigma \sum_i \phi_i(s_{t+1}, a) \right) & \text{w.p. } 1 - \epsilon \\ \text{uniform random action from } \mathcal{A} & \text{w.p. } \epsilon \end{cases}$
    $\delta \leftarrow r_{t+1} + \gamma \left(\phi(s_{t+1}, a_{t+1})^T \theta + u(a_{t+1}) \phi(s_{t+1}, a_{t+1})^T \mathbf{1} \sigma \right) - \phi(s_t, a_t)^T \theta$;
    $\theta \leftarrow \theta + \alpha \delta \left(\phi(s_k, a_k) - \gamma \phi(s_{k+1}, a_k) \right)$;
    $t \leftarrow t - 1$;
  end
end

Output: θ

Bibliography

S. Adam, L. Busoniu, and R. Babuska. Experience replay for real-time reinforcement learning control. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 2012.

J.S. Albus. A new approach to manipulator control: the cerebellar model articulation controller(cmac). *Journal of Dynamic Systems, Measurement, and Control*, 97:220–227, 1975.

C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11:11–73, 1997.

L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

Leemon C. Baird, III, Scott Fahlman, and Leslie Kaelbling. Reinforcement learning through gradient descent, 1999.

D.P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 3 edition, 2007.

D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

J.A. Boyan and A.W. Moore. Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems*, 7, 1995.

R. Brafman and M. Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 953–958, 2001.

L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

L. Busoniu, A. Lazaric, M. Ghavamzadeh, R. Munos, R. Bubuska, and de Schutter. B. Least-squares methods for policy iteration. In *Reinforcement Learning: State-of-the-Art (Adaptation, Learning, and Optimization)*, pages 75–109. Springer-Verlag Berlin Heidelberg, 2012.

R. Caruana and A. Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *In Proc. 23 rd Intl. Conf. Machine learning (ICML 06*, pages 161–168, 2006.

A. Cherubini, F. Giannone, L. Iocchi, M. Lombardo, and G. Oriolo. Policy gradient learning for a humanoid soccer robot. *Robotics and Autonomous System - Special Issue on "Humanoid Soccer Robots"*, 57(8):808–818, 2009. ISSN 0921-8890.

S. Collins, A. Ruina, R. Tedrake, and M. Wisse. Efficient bipedal robots based on passive-dynamic walkers. *Science 307.5712*, 307.5712:1082, 2005.

R. Coulom. *Apprentissage par renforcement utilisant des reseaux de neurones, avec des applications au controle moteur*. PhD thesis, Institut National Polytechnique De Grenoble, 2002.

M. Deisonroth and C. Ramussen. Pilco: a model-based and data-efficient approach to policy learning. In *Proceedings of the Twenty-Eighth Interational Conference on Machine Learning (ICML)*, 2011.

L.T. Dung, T. Komeda, and M. Takagi. Efficient experience reuse in non-markovian environments. In *Proceedings 2008 International Conference on Instrumentation, Control and Information Technology*, pages 3327–3332, Tokyo, Japan, 2008.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

R. Fonteneau, S.A. Murphy, L. Wehenkel, and D. Ernst. Batch mode reinforcement learning based on the synthesis of artificial trajectories. *Annals of Operations Research*, pages 1–34, 2012.

T. Gabel and M. Riedmiller. Cbr for state value function approximation in reinforcement learning. In *In Proceedings of the 6th International Conference on CaseBased Reasoning (ICCBR 2005*, pages 206–221. Springer, 2005.

M. Garcia, A. Chatterjee, A. Ruina, and M. Coleman. The simplest walking model: Stability, complexity, and scaling. In *ASME Journal of Biomechanical Engineering 120*, pages 281–288, 1998.

J. Gauci and K.O. Stanley. A case study on the critical role of geometric regularity in machine learning. In *In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 628–633, 2008.

Matthieu Geist and Olivier Pietquin. An algorithmic survey of parametric value function approximation. *IEEE Transactions on Neural Networks and Learning Systems*, 24(6): 845 – 867, 2013.

G.J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, 1995.

H. Hachiya, J. Peters, and M. Sugiyama. Efficient sample reuse in em-based policy search. *Machine Learning and Knowledge Discovery in Databases, Lecture Notes in Computer Science*, 5781:469–484, 2009.

T. Hester and P. Stone. Learning and using models. In *Reinforcement Learning: State-of-the-Art (Adaptation, Learning, and Optimization)*, pages 111–141. Springer-Verlag Berlin Heidelberg, 2012.

D. Hobbelen, T. De Boer, and M. Wisse. 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)*, pages 2486–2491, 2008.

IFR Statistical Department. World robotics 2010 - service robots. Technical report, VDMA Robotics + Automation association, 2010.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

S. Kalyanakrishnan and P. Stone. Batch reinforcement learning in a complex domain. In *Proceedings 6th International Conference on Autonomous Agents and Multi-Agent Systems*, pages 650–657, San Francisco, CA, 2007.

S. Kalyanakrishnan and P. Stone. On learning with imperfect representations. In *Proceedings of the 2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, pages 17–24. IEEE, April 2011. ISBN 978-1-4244-9886-4.

M. Kearns and S. Singh. Near-optimal reinforcement learning in polynomial time. In *Proceedings of the Fifteenth Interational Conference on Machine Learning (ICML)*, pages 260–268, 1998.

Rob Knight, Ulrich Nehmzow, and Colchester Co Sq. Walking robots Ů a survey and a research proposal, 2002.

J. Kober and J. Peters. Reinforcement learning in robotics: A survey. In *Reinforcement Learning: State-of-the-Art (Adaptation, Learning, and Optimization)*, pages 579–610. Springer-Verlag Berlin Heidelberg, 2012.

M.G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research 4*, pages 1107–1149, 2003.

S. Lange, T. Gabel, and R. Riedmiller. Batch reinforcement learning. In *Reinforcement Learning: State-of-the-Art (Adaptation, Learning, and Optimization)*, pages 45–73. Springer-Verlag Berlin Heidelberg, 2012.

Long-Ji Lin. Self-improving reactive agents based on reinforcment learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

H. Meijdam, M.C. Plooij, and W. Caarls. Learning while preventing mechanical failure due to random motions. In *Proc. IEEE/RSJ INternational Conference on Intelligent Robots and Systems*, page to appear, Japan, November 2013.

F. S. Melo, S. P. Meyn, and M. Isabel Ribeiro. An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th International Conference on Machine Learning*, 2008.

W.T. Miller, E. An, F. Glanz, and M. Carter. The design of cmac neural networks for control. 1:140–145, 1990.

J. Moody and M. Saffell. Learning to trade via direct reinforcement learning. In *IEEE Transactions on Neural Networks*, volume 12.4, pages 875–889, 2001.

A.W. Moore and C.R. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993.

J. Morimoto, J. Nakanishi, G. ENdo, G. Cheng, C.G. Atkeson, and G. Zeglin. Poincare-map-based reinforcement learning for biped walking. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2381–2386, 2005.

M. Ogino, Y. Katoh, M. Aono, M. Asada, and K. Hosoda. Reinforcement learning of humanoid rhythmic walking parameters based on visual information. In *Advanced Robotics*, pages 677–697, 2003.

D. Ormoneit and S. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49 (2):161–178, 2002.

M. van Otterlo and M. Wiering. Reinforcement learning and markov decision processes. In *Reinforcement Learning: State-of-the-Art (Adaptation, Learning, and Optimization)*, pages 3–42. Springer-Verlag Berlin Heidelberg, 2012.

J. Peters and S. Schaal. Policy gradient methods for robotics. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006.

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

G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

E. Schuitema. *Reinforcement Learning on Autonomous Humanoid Robots*. PhD thesis, Delft University of Technology, 2012.

E. Schuitema, M. Wisse, T. Ramakers, and P. Jonker. The design of leo: A 2d bipedal walking robot for online autonomous reinforcement learning. *Intelligent Robots and Systems (IROS), IEEE/RSJ International*, 2010.

A.L. Schwab and M. Wisse. The basin of attraction of the simplest walking model. In *Proceedings ASME Design Engineering Technical Conferences*, Pittsburgh, Pennsylvania, 2001.

D. SIlver, R. Sutton, and M. Muller. Sample-based learning and search with permanent transient memories. In *Proceedings of the Twenty-Fifth Interational Conference on Machine Learning (ICML)*, pages 968–975, 2008.

S. Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. In *MACHINE LEARNING*, pages 123–158, 1996.

W.D. Smart and L.P. Kaelbling. Practical reinforcment learning in continuous spaces. In *Proceedings of the 17th Interational Conference on Machine Learning (ICML)*, pages 903–910, Stanford University, US, 2000.

R.S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 1991.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 1998.

R.S. Sutton, C. Szepesvari, A. Geramifard, and M. Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, 2008.

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

S. Timmer and M. Riedmiller. Fitted q-iteration with cmacs. In *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning(ADPRL 07)*, Honolulu, USA, 2007.

M. Toussaint and S. Vijayakumar. Learning discontinuities with products-of-sigmoids for switching between local models. In *Proceedings of the 22nd international conference on machine learning*, 2005.

H. van Hasselt. Reinforcement learning in continuous state and action spaces. In *Reinforcement Learning: State-of-the-Art (Adaptation, Learning, and Optimization)*, pages 207–251. Springer-Verlag Berlin Heidelberg, 2012.

H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering. A theoretical and empirical analysis of expected sarsa. In *ADPRL 2009: Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 177–184, March 2009.

S. Vijayakumar, T. Shibata, and S. Schaal. Reinforcement learning for humanoid robotics. In *Autonomous Robot*, page 2002, 2003.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

P. Wawrzynski. Real-time reinforcement learning by sequential actorŰcritics and experience replay. *Neural Networks*, 22:1484–1497, 2009.

D. Wettschereck, D. W. Aha, and T. Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 1997. Special Issue on Lazy Learning Algorithms.

WHO. Investigating in the health workforce enables stronger health systems. In *Fact sheet 2007, Belgrade, Copenhagen*, 2007.