Insights in Reinforcement Learning

Formal analysis and empirical evaluation of temporal-difference learning algorithms

Hado van Hasselt

Nederlandse Organisatie voor Wetenschappelijk Onderzoek

This research was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 612.066.514.

SIKS Dissertation Series No. 2011-04
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by Wöhrmann Print Service

ISBN 978-90-39354964

Insights in Reinforcement Learning

Formal analysis and empirical evaluation of temporal-difference learning algorithms

Inzichten in Reinforcement Learning

Formele analyse and empirische evaluatie van algoritmes die leren van temporele verschillen

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. J.C. Stoof, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op maandag 17 januari 2011 des middags te 4.15 uur

door

Hado Philip van Hasselt

geboren op 12 september 1979 te Utrecht

Promotoren:	Prof.dr. J.-J.Ch. Meyer
	Prof.dr. L.R.B. Schomaker
Co-promotor:	Dr. M.A. Wiering

Dit proefschrift werd mede mogelijk gemaakt met financiële steun van de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO).

CONTENTS

CHAPTER 1

INTRODUCTION

In artificial intelligence the aim is to build intelligent entities (Russell and Norvig, 2009). According to some, an artificial entity can be called intelligent when it can successfully mimic human behavior (Turing, 1950). For others, an entity is intelligent if it can solve mathematical equations or check proofs (Appel et al., 1977). In any case, there are many useful applications for so-called intelligent machines (Holland, 1992; Kortenkamp et al., 1998). In this dissertation, we will not discuss whether such entities are in fact intelligent. Also, we will not consider the different problems one can encounter when trying to define what 'intelligence' in fact is (McCarthy and Hayes, 1968; Brooks, 1991; Dennett, 2005). Rather, we will focus on a specific skill that is related to intelligence, if not essential for it: the skill of learning.

The research field of machine learning constitutes a subset of the broader field of artificial intelligence. Machine learning researchers investigate how to construct algorithms that allow a computer to learn from observations (Mitchell, 1996; Bishop, 2006). For instance, the goal can be to have a computer program that can recognize faces (Turk and Pentland, 2002) or that can play a game of backgammon (Tesauro, 1994). These goals can be met by constructing a step by step recipe that the computer can follow in order to reach the desired result. However, for such an approach the programmer has to know a solution to the problem and implement this solution in full. Also, it requires that the programmer takes into account all possible situations the program might encounter. This is often infeasible. Therefore, it is often easier to use a machine learning algorithm that allows the computer to deduce solutions itself, based on its observations. This way, the computer can continue to learn from its experience while it is in operation and it may even discover solutions that the programmer would have never thought of.

In this dissertation we will discuss some advances and insights into the research field of reinforcement learning (Sutton and Barto, 1998). Reinforcement learning can be considered to be a subset of machine learning. In reinforcement learning, the focus is primary on algorithms that can learn through interaction with an environment. In other words, these algorithms learn through a form of trial-and-error.

A reinforcement learning algorithm is often called an 'agent' and the setting the agent finds itself in is called the environment of the agent. Later in this chapter we will discuss what our definition for an agent is. Then, we will discuss how reinforcement learning fits into this picture and discuss the contributions and views that are presented in this dissertation.

In general, a reinforcement learning agent chooses an action to perform in every situation it encounters. The action might induce a reward or a penalty and leads to a new situation that may or may not depend on the action that was taken. The reinforcement signal and the new situation are then used by the reinforcement learning algorithm to improve the performance of its future actions. The difference with many other machine learning algorithms is that the reinforcements do not tell the agent what it should have done, only how well the action that it actually did turned out. Such reinforced interactions with an environment occur naturally in many settings. For instance, when playing a game one may eventually obtain a reward or a penalty when the game is won or lost, but the opponent will usually not tell the player what a better move would have been. Likewise, a production plant may give feedback on the number of goods that are produced and this can be used by a reinforcement learning algorithm to try to increase these numbers.

In this dissertation we will mainly discuss algorithms that can be used by agents to *optimize* their behavior in terms of some reinforcement signal they receive from the environment. This means we want the agent to improve its behavior, rather than just to learn about its environment in a neutral sense. In other words, we investigate the problem of 'control', rather than the problem of 'prediction'. We strongly favor methods that are fast to compute, easy to implement, easy to understand intuitively and widely applicable. Informally, we refer to algorithms that adhere to these requirements as 'practical'. There are many possible practical methods and it is often hard to know beforehand which method is best for a given problem. At the end of this dissertation we will present some guidelines that state in which situation which algorithm is most likely to perform well, based on our experiments, analyses and observations.

In reinforcement learning, many algorithms have been proposed to solve the problem of learning good policies of behavior in arbitrary environments. In this dissertation, we look at a subset of these methods. In particular, we will examine model-free temporal-difference methods.

The adjective 'model-free' is somewhat of a misnomer, but it implies that the algorithms do not build an explicit model of the environment. The behavioral policy that the agent obtains through learning may contain an implicit model of the environment, for instance in the sense of a value function that describes the expected value of an action in every situation. However, no attempt is made to model the environment itself. Naturally there exist problems that are easier solved with a model-based approach. The idea behind the choice for model-free algorithms is that it is often easier to determine which actions are promising than it is to understand and model the entire world around oneself. Additionally, a model of the environment in general requires much more storage space than a policy of behavior that tells the agent what to do in each situation. In practice, this usually makes model-free algorithms

easier to implement and computationally faster.

The methods we investigate most are 'temporal difference' methods because they use earlier obtained knowledge to be able to improve their behavior after every action they take. The notion of temporal differences will be explained in more formal terms in Chapter 2.

In the remainder of this introduction, we will introduce some of the concepts that will be used in this dissertation. In Section 1.1, we will state the aim of this work and Section 1.2 discusses how this relates to previous work. We will present our definition of a learning agent in Section 1.3 and discuss how reinforcement learning algorithms can be used by these agents and how it relates to other research fields in Section 1.4. Finally, we give an overview of the dissertation in 1.5.

1.1 The Aim of this Dissertation

The aim of this dissertation is to investigate the properties of some of the reinforcement learning algorithms in order to be able to give suggestions beforehand on which algorithm is to be preferred on a given problem. Often, the precise characteristics of the problem and especially of the optimal solution are not known before learning commences. Therefore, such suggestions should be based on high-level observations that are more likely to be available about the type of problem that is to be solved. For instance, a relevant high-level observation could be whether the reinforcement signals are expected to contain a large amount of noise or not. The goal is to minimize the need for a domain expert, but to be able to use the basic knowledge about each problem efficiently.

We will look at low-level properties of many of the algorithms, since we believe many of the high-level behaviors that are obtained with use of these algorithms can only be fully understood when the low-level properties are fully known. For instance, we will show that an often used algorithm called Q-learning (Watkins, 1989) suffers from a bias that thus far seems to have been overlooked. This bias can have a direct impact on the behaviors that result from the use of this algorithm, or any of its many variants.

1.2 Previous Work

Much of the existing literature on reinforcement learning algorithms seems to broadly fall into one of three general categories. Naturally, these categories are subjective in nature and some papers will fall in more than one of the categories, or even in none of them. However, we feel the categorization is useful and fairly intuitive.

The first category is the general theoretical work. For instance, in this category algorithms are shown to converge to the exact desired result in the limit (e.g., Dayan and Sejnowski, 1994; Jaakkola et al., 1994; Szepesvári, 1998; Singh et al., 2000; van Hasselt and Wiering, 2007b; Sutton et al., 2008), or within a certain amount of time with high probability (e.g., Fiechter, 1994; Kearns and Singh, 1999; Even-Dar et al., 2002; Even-Dar and Mansour, 2003; Mannor and Tsitsiklis, 2004; Strehl et al., 2009). Although useful, unfortunately often these results are not mirrored completely in the actual performance of the different algorithms on real-world tasks. For instance, in Chapter 7 we will see that an algorithm that can not be proven to converge to the correct result will reach a far better performance on the well-known mountain car benchmark than any of the tested provably convergent algorithms. As another example, sometimes different algorithms share the same theoretic asymptotic convergence rate, while the finite time performance of one of these algorithms is vastly better on a subset of problems. An example of this can be found in Chapter 4.

The second category of literature proposes new algorithms or adaptations to old algorithms without a strict theoretical proof of general improvement. Usually, the proposed novelties are then shown to be useful according to some performance measure. This performance metric may be theoretical, such as a general bound on the expected number of times the algorithm performs a poor action (e.g., Singh and Yee, 1994; Strehl et al., 2006). More often, the performance metric is empirical and results from comparing the new algorithms to older algorithms on some benchmarks (e.g., Riedmiller, 2005; Geramifard et al., 2006; Wiering and van Hasselt, 2007). Sometimes both types of metrics are used. Naturally, new algorithms are often not fully understood right away and some turn out to be strict improvements over older algorithms, while others seem to be better only on the metrics they were tested on.

The third category covers purely empirical work. These papers describe the results of experiments with one or more existing algorithms (e.g., Michels et al., 2005; Taylor et al., 2006; Whiteson et al., 2007). A requirement for many venues to publish such a paper is that the domain on which the algorithms are tested is sufficiently complex, or of clear importance to people outside the research field. Examples include applications on power systems (e.g., Ernst et al., 2004) or in psychology research (e.g., Conn et al., 2008; Liu et al., 2008). The emphasis is then on the domain the algorithm is applied to, rather than on the algorithm itself.

In this dissertation, we will attempt to tread some middle ground between the first two types of papers. We will look at the expected actual performance of algorithms by considering both the asymptotic guarantees and the performance on some simple settings. This allows us to build intuitions about which algorithm to prefer in which setting. In some cases, we formalize these intuitions, for instance by showing that an algorithm converges in the limit to the

optimal behavior, but the emphasis is on the practical use of the algorithms. As such, we generalize older algorithms, provide alternatives and discuss the advantages and disadvantages of the different methods. Rather than trying to make statements about the general value of each algorithm, we try to pinpoint in which settings the algorithms can probably be expected to perform well.

We will show the results of small experiments with existing algorithms to be able to highlight the advantages and disadvantages of using one approach over another. Where possible, we will state general theoretical guarantees, but if these cannot be given in we will not refrain from stating our intuitions why one algorithm performs better than another in some domain, based on inspection on how the algorithms works. This would be hard to do if we would only run experiments on large complex domains, since then it becomes much harder to interpret why and how an algorithm reaches the behavior that is observed. Although we will discuss the limiting convergence properties of certain algorithms and we feel it is useful to have these guarantees in general, we will see that the practical performance of an algorithm often has little connection with the asymptotic guarantees that may of may not be given.

We hope this dissertation can be useful in gaining more understanding about the algorithms that are described. These algorithms include well established, often discussed algorithms and brand new algorithms. Eventually, this hopefully leads to a better understanding on when to use which algorithm and on how to construct even better algorithms, since there is no single algorithm that is best in all possible domains.

We examine which algorithms give a good trade-off between performance, speed and ease of implementation. This will lead up to some rules of thumb that we will present in Chapter 8. There, we condense our findings into some suggestions on when to use which algorithm when only very limited knowledge about the problem at hand is assumed. Naturally, if more is known about a particular problem, more specific algorithms can be used that incorporate this domain knowledge. Additionally, for every setting there will be an algorithm that performs better than the algorithms that we suggest. However, we have confidence that the suggested algorithms are fairly widely applicable and will be able to reach satisfactory performance levels in many problems of interest.

1.3 Agents

Agents are a common useful concept in computer science. Intuitively, agents can be viewed as entities that perform some task more or less independently while observing the environment. For instance, robots can be considered agents, but so can certain computer programs. A more formal definition is

easy to give, but unfortunately the literature seems to disagree on what this formal definition should be. We will first discuss some of the supposed requirements for a system to be called an agent. Then, we will discuss what the term 'agent' means in the context of reinforcement learning.

1.3.1 Definitions of Agents

The word 'agent' is used for a number of different concepts. One particularly broad definition is given by Russell and Norvig (2009), who state that

> An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

However, this particular definition is so broad that there are few systems that can not be considered an agent. In particular, the vague requirement that an agent 'can be viewed as perceiving its environment' is fairly subjective. For instance, would one consider a doorbell an agent since it can perceive a finger pressing and it can respond by making a suitable noise?

Others have tried to make finer distinctions between types of agents by listing properties that agents do or do not have. One such list of properties is given by Franklin and Graesser (1997) who list the properties *reactive*, *autonomous*, *goal-oriented*, *temporally continuous*, *communicative*, *learning*, *mobile*, *flexible* and *character*. Unfortunately, these properties are also not without problems. If one considers an agent to be a mapping from its (past) percepts to its actions, many of these properties stop to make sense. For instance, a goal-oriented agent is defined as an agent that 'does not simply act in response to the environment'. This property clearly hinges on the definition of the word 'simply'. Does this imply that if the reasoning process is complex enough one can speak of a goal-oriented agent, even if the whole reasoning process is deterministic? The problem here is that any deterministic process can in principle be replaced with a (potentially large) table that tells the agent what to do in each separate case. Then no actual reasoning is involved and the agent acts 'simply' in response to the environment (and potentially its past percepts). But if such a simple table-based agent perform precisely the same actions in the same situations as the reasoning agent, would one benefit from calling one goal-oriented and the other not?

Similar problems occur for some of the other properties. For instance, can an action that changes the environment be interpreted as communication? And can any process that is implemented on a digital computer ever truthfully be called temporally continuous?

Because of the aforementioned issues, we will refrain from further discussing such lists of properties. Our definition of an agent will be quite close to the one given above and and we can formalize it fairly simple as follows:

Definition 1.1 (Agents)**.** *An agent is a mapping from its internal state and its percepts to actions in an environment.*

The internal state of the agent can be important, but something can be considered an agent if its internal state is (nearly) empty. Then, the agent is actually a mapping from percepts to actions. In such a case, we will speak of a reactive agent.

Definition 1.2 (Reactive Agents)**.** *A reactive agent is a mapping from percepts to actions.*

Under this definition, a doorbell is also a reactive agent, be it a very simple one. Reactive agents are not necessarily deterministic, but we assume they will not change their behavior based on their past observations. A reactive agent might update its internal state to store information about its past percepts. The only restriction is that it does not use the internal state to influence its actions.

In the next subsection we will highlight one property that we believe can be formalized in a straightforward manner and that is important in the rest of this dissertation: the property of learning.

1.3.2 Learning Agents

Our definition of a learning agent is the following:

Definition 1.3 (Learning Agents)**.** *A learning agent is a mapping from its internal state and its percepts to actions in an environment, where the internal state changes over time under influence of the percepts and this can potentially result in changes in the perceived mapping from percepts to actions.*

This implies that if one observes a learning agent, one will usually see change its behavior over time, because its internal state changes. However, this does not need to happen. For instance, an agent that learns from its percepts that the actions it was performing yield desirable results, it may learn to keep performing these actions. The crux is that for each learning agent there must exist some situation in which the percepts indeed lead to a change in the behavior of the agent.

In this chapter, we do not specify how the internal state changes or how this potentially influences the mapping from percepts to actions, but in general the learning process will attempt to optimize some predefined criteria. In reinforcement learning, these criteria are communicated to the agent by means of reinforcements. In the rest of this dissertation we will discuss many algorithms that can be used to construct a learning agent and we will explicitly specify how these change the internal state of the agent and how this can affect the behavior of the agent.

1.4 Reinforcement Learning

The next chapter will give a more formal introduction to many of the concepts in reinforcement learning. This section will be more high-level and will discuss the connection between reinforcement learning and the definition of a learning agent that we gave above. Additionally, we will shortly discuss how reinforcement learning fits in the larger research fields of machine learning and artificial intelligence.

1.4.1 Learning Policies

An agent in reinforcement learning is quite simple: it consists of a policy of behavior and a learning algorithm to adapt this policy. Most other elements, such as the source of the reinforcements, post processing systems on the percepts and even the body of the agent are considered to be part of the environment. The policy of behavior is then adapted with use of the learning algorithm. This algorithm can take many different forms and will usually store some additional information in an internal state, such as an estimate of the expected value of an action, or a model of the environment.

The policy can be considered a function that maps situations to actions. In general, learning functions from data is the topic of machine learning and reinforcement learning can be considered a subfield of this broader field of research. Machine learning concerns itself with the design of algorithms that can learn from available data in an automatic, clearly formalized way, such that these algorithms can be implemented on a computer. Broadly speaking, machine learning algorithms can be divided into the following three categories:

- **Supervised Learning** in which the goal is to find a function that maps inputs to outputs and where the data from which such a mapping is learned consists of labeled samples of inputs with their corresponding target outputs. Supervised learning is a very active field and many good books have been written on the topic, such as those by Vapnik (1995) and Bishop (2006). Examples of supervised learning include classification of handwritten numerals (LeCun et al., 1989; Bottou et al., 1994), handwritten text (Schomaker, 1993) and prediction (Hastie et al., 2005).

- **Unsupervised Learning** in which the goal is to discover underlying patterns, clusters and regularities in a set of unlabeled data (Hartigan, 1975; Barlow, 1989). For instance, this is useful in visualizations of data to determine how to interpret and further process this data (Deboeck and Kohonen, 1998; Nattkemper and Wismüller, 2005) or to discover irregularities (Pearlmutter and Hinton, 1986).

- **Reinforcement learning** in which the goal is to find optimal strategies of behavior for an artificial agent when only reinforcement signals are available that do not specify how the agent should perform its desired task, but only indicate how well the agent is performing at the present time. Examples of possible applications include games (Tesauro, 1994), robotics (Peters et al., 2003) and control tasks (Werbos, 1989b). We will discuss reinforcement learning at length in the next chapter. Some introductory texts of varying complexity and completeness include the books by Bertsekas and Tsitsiklis (1996), Sutton and Barto (1998), Bertsekas (2007) and Szepesvári (2010).

Supervised learning needs the availability of pairs of inputs and corresponding targets from which it can learn. Usually, it is assumed that the mapping that is to be learned is fixed and that a set of labeled data is available to learn from at a certain time. Unsupervised learning does not assume targets are available and only a set of unlabeled data points is required. The most one can hope for in such a setting is to structure the data in a meaningful manner. Reinforcement learning can be interpreted as being somewhere between these two settings. Targets for the desired behavior of the agent are not assumed to be available, but we do assume there is some measure that can be observed that informs the agent how well it is doing. The agent can then obtain more information simply by trial and error. The appeal of this method is that it is usually much easier to construct a meaningful reinforcement scheme than it is to construct good policies of behavior from scratch. In fact, most human behavior is learned through a mechanism more resembling reinforcement learning than supervised learning; incentives and punishments are often encountered in real life, but precise descriptions on how to act are much less common.

For instance, in a game playing setting one may not know what a good strategy is to win the game. However, a reinforcement signal is easy to define: simply give a positive reinforcement when the game was won and a negative reinforcement when the game was lost. Using this method, Tesauro (1994) constructed an artificial backgammon player that played against itself until it was as good as the best human players at the time. Because no desired behavior was specified in advance, the artificial player could even improve the known opening theory of the game, by suggesting a move that was uncommon for master level human players. Later analysis proved the artificial player to be correct and many human players adopted the suggested improvement. This is another appeal of reinforcement learning: since no targets are used for the behavior, new previously unconsidered solutions may be found that the programmer of the agent need not have been able to construct. This contrasts with supervised learning, where the optimal behavior is known even if the mapping that results in the best general solution is not.

Naturally, the different fields influence each other, and many different machine learning techniques are used in reinforcement learning. In this dissertation, we will mainly focus on algorithms that learn a value for every action in every situation. This value then represents a measure for the expected reinforcement the agent will receive after performing the corresponding action. If these values are known, the optimal policy can be constructed simply by selecting the highest valued actions. Some methods to represent these values are taken from more general machine learning approaches, such as the neural networks that are discussed in Chapter 7. However, in general it is non-trivial to find the correct action values since the actions of the agent influence the reinforcements and an action may have an effect that is only observed much later.

1.4.2 Values, Policies and Models

In general, if one wants to learn new behaviors there are three general ways to do so. First, can store indirect measures that influence the behavior such as a value that represents the expected future reinforcement for each action. Second, one can directly adapt the policy of behavior itself. Third, one can learn a model of the environment and use this to reason about good policies of behavior. Each general methods corresponds to specific algorithms in reinforcement learning that update the behavior in that manner.

Each of the general methods has its own advantages and disadvantages. For instance, adapting the policy itself seems a convenient approach, but how does one then compare two different policies? And how does one decide how to adapt a policy in the first place? When values are used, an interesting question is how to use these values to construct a policy. Finally, constructing a model of the environment can be very complex and an obvious question is how precisely to use this model to construct a policy.

In this dissertation, we will mainly discuss value-based model-free reinforcement learning algorithms. The reason is not that we believe that the other approaches are less likely to produce good results. Rather, some interesting results can be obtained with use of values functions and we have therefore focused on this particular subfield of reinforcement learning.

A potential advantage of using values compared to using a model-based approach lies in the fact that there exist problems in which the desired behavior is far simpler than an accurate model of the environment. The backgammon game discussed above is an example of this. If one would want to model the potential outcomes of an action, one would need to take into account all the possible outcomes of each throw of the dice.[1] This means a tree of possi-

[1] A very small explanation for those who do not know the game of backgammon. Backgammon is a two-player game where each player throws two dice each turn to determine the possible moves. Then the player can usually choose from a fairly large number of legal moves

bilities branches out very quickly and it is impractical to consider all possibilities of a few turns in the future. However, learning the approximate value of a large number of rules through trial and error was proven to be quite successful. Additionally, in many problems large parts of the environment may be inconsequential for the choice of action and therefore it would be a waste of resources to construct a complete model for these parts. The problem is that often it is not known beforehand which parts of the environment are important, making it hard for model-based approaches to prune their models appropriately. Naturally, there also exist settings in which a model-based approach is far superior to a value-based approach, since it potentially holds more relevant information for the agent. In particular, it may be better to build a model if the environment is fairly simple and therefore easily modeled but there are many actions to choose between in each setting.

1.4.3 Practical Reinforcement Learning

As mentioned in the introduction, we call an algorithm practical if is it fast to compute, easy to implement, easy to interpret and widely applicable. In this section, we specify what we mean by each of these properties and why we consider them to be important.

The first requirement that we stated earlier is that an algorithm is fast to compute. In reinforcement learning, this may or may not be important in practice. If an algorithm is run on a problem where there is much time between each two consecutive decisions, it is less important that the used algorithm is computationally efficient. Then, it is much more important that the algorithm makes optimal use of the available information, since the gathering of more information can be costly. On the other hand, there are problems where a fast simulation of the problem exists and the time needed by the algorithm should not be too long between each two steps in order to be able to run in real time. In this dissertation we will mainly consider algorithms that run in linear time compared to the size of the percepts that are observed.

The second requirement is ease of implementation. Although this may not be very important for the quality of the behavior that is learned by the agents, we include this requirement for a couple of reasons. First, if an algorithm is easy to implement, the probability that an error is made during implementation is less. Second, the simpler the algorithm, the more likely its performance guarantees can be generalized to untried problems, as long as these are fairly similar. Finally, we feel that if two algorithms share more or less the same performance guarantees, an easy algorithm is to be preferred in the same way as Ockham's razor is used to argue that a simple theorem is better than a complex theorem. We also interpret the difficulty of tuning an algorithm to be part of the implementation complexity. Some algorithms

for each possible outcome of the dice.

have many parameters and only work for very specific settings of these parameters. Then, we prefer algorithms with less tunable parameters, or for which the range on which the parameters can be set for acceptable performance is large.

The third requirement is ease of interpretation. In practice, this mostly overlaps with ease of implementation, although conceptually the requirements are somewhat orthogonal. For a simple algorithm that is more easily understood, it will be easier to predict whether or not the algorithm can be successful in a certain specific setting. Furthermore, the argument on Ockham's razor also applies to the ease of interpretation.

The fourth requirement was that the algorithm in widely applicable. This means we do not want to restrict ourselves to much to a single class of problems that is to be solved. However, we do note that there is no cure-all algorithm that is simple, easy to implement and the best choice for all possible problems. This final requirement therefore mainly implies that if two algorithms are equally simple to implement and to interpret, they are equally fast and they produce equally good solutions, then the algorithm that performs better on most other problems is to be preferred in general. Of course, this does not mean it is also the best choice for any specific problem.

An implicit requirement for any practical algorithm is of course good performance. In reinforcement learning, this performance is usually measured by how quickly the algorithm finds good solutions and how good the final solutions are. Most algorithms continue to improve over time. Then, the final performance is usually the performance after some fixed allocated time, which is the metric that we will use in our experiments. In some cases we will discuss the asymptotic performance in the limit and discuss whether or not this is optimal. We will investigate the performance of many algorithms empirically in this dissertation and we will discuss both the speed of learning and the quality of the solution that is found eventually. However, we feel that sometimes too much importance has been put on just the performance of the algorithm, sometimes resulting in algorithms that perform only marginally better than previous algorithms on a limited set of problems while being very hard to implement and to interpret.

The focus on practical algorithms is one of the reasons we mainly focus on model-free temporal-difference methods. We do not believe these methods will always result in the best possible behavior for a given set of experiences, but these methods are fast to compute, easy to implement and when properly used they perform quite well. As such, we will devote a significant part of this dissertation to the analysis of and experiments with these algorithms, to build a better intuition on how these algorithms are best used. In this view, a partial goal of this dissertation is to help with a specific aspect of model-free temporal-difference methods: the ease of interpretation.

1.5 Overview

In the next chapter, we will give a more formal definition of the various concepts that are used in reinforcement learning. There, we will discuss policies, values, rewards, states and actions and the framework of Markov decision processes that is used to model the environment as a problem that can be solved by reinforcement learning. In Chapter 3, we discuss fairly general properties of finding the maximum of a set of noisy values. These properties are important in reinforcement learning, as well as in any other field that attempts to optimize a certain criterion in a noisy setting. We will use the analysis from Chapter 3 in Chapter 4 to show that Q-learning can suffer from large overestimations in its assessment of the value of an action. These overestimations are harmful, because they can affect the behavior of an agent that uses Q-learning in an undesirable manner. This is relevant, because Q-learning is one of the most widely used algorithms in reinforcement learning and the inspiration for many later algorithms that will suffer from similar overestimations. Additionally, in Chapter 4 we construct the Double Q-learning algorithm as an alternative to Q-learning. This algorithm is based on an alternative method to find the maximum of a set of noisy values that is discussed in Chapter 3.

In Chapter 5 we discuss several other model-free temporal-difference methods that can be used instead of Q-learning. We generalize some of these algorithms to a General Q-learning algorithm and discuss other improvements, such as the Expected Sarsa algorithm (van Seijen et al., 2009) that can be viewed as an improvement over the well-known Sarsa algorithm (Rummery and Niranjan, 1994). We also discuss other alternatives, such as QV-learning (Wiering, 2005; Wiering and van Hasselt, 2007), actor critic methods (Barto et al., 1983) and Acla (Wiering and van Hasselt, 2007). We point out differences and similarities between the algorithms and compare them empirically to see if our intuitions are confirmed. This shows that there can be large performance differences between the algorithms, although it is problem dependent which algorithm performs best.

In some cases, very little is known about a problem beforehand. However, in Chapter 5 we demonstrate that which algorithm is to be preferred can be very problem dependent. In Chapter 6 we present a partial solution for this by constructing policy-based ensembles of the different algorithms. We discuss many different ways to build these ensembles and show that the performance of such ensembles is comparable to or better than the best individual algorithms. Therefore, these ensembles can be useful to increase the likeliness of obtaining good policies of behavior after a small number of interactions with the environment. The only drawback is a somewhat increased computational load, but the order of complexity of the ensemble is never greater than that of the slowest constituting algorithm.

In Chapter 7 we extend our discussion to continuous states and actions. For some readers it may seem strange that we wait so long to extend our discussion to function approximation. This was a deliberate choice, because we want to discuss the properties of the algorithms in easily analyzable settings first, before introducing the additional complexity that function approximation introduces. However, most of the discussion in the preceding chapters is orthogonal to these issues, implying that the obtained results are also relevant in settings where large or continuous spaces force us to adapt the algorithms. In Chapter 7 we will also discuss continuous action spaces and how to deal with problems that have such action spaces. This will include a discussion on existing methods, such as policy gradient methods (Williams, 1992; Sutton et al., 2000; Baxter and Bartlett, 2001) and evolutionary algorithms (Holland, 1962; Rechenberg, 1971; Holland, 1975; Schwefel, 1977; Davis, 1991; Bäck and Schwefel, 1993). Further, we introduce a simple new temporal-difference algorithm for continuous state and action spaces called Cacla (van Hasselt and Wiering, 2007a; van Hasselt and Wiering, 2009) and present some experimental results that show that this algorithm is very competitive to the current state of the art.

Chapter 8 concludes the dissertation. Here we will discuss our general findings and give some pointers for future research. We will also summarize our findings with some rules of thumb that can be used to find a good algorithm for each problem, without having to try them all.

CHAPTER 2

REINFORCEMENT LEARNING: A SHORT INTRODUCTION

This chapter serves as an introduction in reinforcement learning. There are many texts for a more thorough treatment (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998; Bertsekas, 2007). We introduce the main concepts we will be dealing with and we introduce our notation. Our main focus will be model-free temporal-difference algorithms, but we will also give some pointers to other approaches when appropriate. The chapter is organized as follows.

We start with a description of so-called Markov Decision Processes (MDPs) in Section 2.1, which are the formal models that are used to model the problems we want to solve. In general, reinforcement learning algorithms assume such MDPs are given as input to the algorithm.

In Section 2.2 we introduce model-based dynamic programming as a method to solve these MDPs. Solving an MDP can mean one of two things. In some cases one is interested in the value of a certain policy of behavior and solving an MDP in this case means we predict the value of this given policy. However, in most cases one will be interested not in the value of some predetermined policy, but in the optimal policy. This is what we mean with 'reinforcement learning for control': solving an MDP to extract the optimal policy for this MDP. We will discuss model-based algorithms that use the structure of the MDP to find this optimal policy.

In Section 2.3 we drop the assumption of a known model and discuss model-free algorithms. This includes Monte Carlo methods and temporal-difference (TD) value-prediction methods. Additionally, TD algorithms for control are considered. This section also includes a discussion on eligibility traces, that generalize TD and Monte Carlo methods to fall into a single framework.

2.1 Markov Decision Processes

Reinforcement learning can be used to find optimal solutions for many problems, but of course these problems should be modeled in a way that the algorithms can be applied. For this, the framework of Markov Decision Processes (MDPs) is used (Bellman, 1957; Howard, 1960; Puterman, 1994; Boutilier et al., 1999).

We use a similar notation to that used in the book by Sutton and Barto (1998). An MDP can then be defined as a tuple (S, A, P, R, γ), with the follow-

ing definitions for its contents.

- S is a set of states, where $s_t \in S$ denotes the state the agent is in at time t.
- $A(s)$ is a set of available actions in state s, where $a_t \in A(s_t)$ denotes the action the agent performs at time t.
- $P : S \times A \times S \to [0,1]$ is a transition function where $P_{sa}^{s'}$ denotes the probability of ending up in state s' when performing action a in state s.
- $R : S \times A \times S \to \mathbb{R}$ is a reward function where $R_{sa}^{s'}$ denotes the expected reward when the agent transitions from state s to state s' after performing action a. The actual reward that is witnessed by the agent after performing action a_t and on transitioning to state s_{t+1} may contain noise and is denoted as r_{t+1}, where $E\{r_{t+1}|(s,a,s') = (s_t,a_t,s_{t+1})\} = R_{sa}^{s'}$.
- $\gamma \in [0,1]$ is a discount factor.

Other common styles of notation use i, j or $x \in X$ for states, $u \in U(i)$ for actions, $g(i,u,j)$ for costs, which correspond to negated rewards in our notation, $p_{ij}(u)$ for transition probabilities and α for the discount factor. It is also fairly common to use $J(x)$ for the state value that we will later denote with $V(s)$, and $J(x,u)$ for the action value that we will denote with $Q(s,a)$.

The MDP is sometimes called the environment to contrast it with the inner workings of the agent. An agent in reinforcement learning is usually assumed to be very simple, consisting mainly of an action selection policy $\pi : S \times A \to [0,1]$, where $\pi_t(s,a)$ denotes the probability that the agent will select action a to perform if it is in state s at time t.

Definition 2.1 (Deterministic policy)**.** *A deterministic policy π is a policy where $\forall s : \pi(s,a) = 1$ for exactly one $a \in A(s)$ and $\pi(s,b) = 0$ for all other $b \in A(s)$.*

Definition 2.2 (Stationary policy)**.** *A stationary policy is a policy that does not change over time, i.e. where $\forall t : \pi_t = \pi$.*

With a slight abuse of notation, we will use $\pi(s)$ to refer to the probability distribution or the probability mass function of the actions in state s. We will then use $a \sim \pi(s)$ to indicate that action a is chosen according to the probability function in state s. One interaction of an agent with an MDP consists of the agent observing the present state s_t and choosing an action a_t to perform according to its policy. The MDP then transitions to a new state s_{t+1} with probability $P_{s_t a_t}^{s_{t+1}}$ and returns a reward r_{t+1} with expected value $R_{s_t a_t}^{s_{t+1}}$. Normally, any physical presence of the agent itself is also considered part of the environment in this view.

Markov decision processes by definition fulfill the Markov property. This property is defined in general as follows.

Definition 2.3 (Markov property)**.** *A stochastic process has the Markov property if the conditional distribution of the next state of the process depends only on the current state of the process.*

For an MDP, this property implies that the transitions P and the rewards R do not depend on the states the agent visited in the past. This can be expressed formally with

$$E\{s_{t+1}|s_0,a_0,\ldots,s_t,a_t\} = E\{s_{t+1}|s_t,a_t\} \text{ , and}$$
$$E\{r_{t+1}|s_0,a_0,\ldots,s_t,a_t\} = E\{r_{t+1}|s_t,a_t\} \text{ .}$$

There exist problems in which the Markov property does not hold for the observable states and actions. These problems can usually be modeled with partially observable MDPs (POMDPs) (Sondik, 1971; Smallwood and Sondik, 1973; Monahan, 1982). POMDPs assume that there is some MDP that describes the problem at hand, but that the agent can not observe the full state. Rather, it is assumed that there is a separate set of observations O, where $o_t \in O$ is the observation the agent makes at time t. Then, the chain of observations may not have the Markov property, but the underlying process is still assumed to be Markov. Since most standard reinforcement learning algorithms assume the Markov property, there is a separate set of algorithms that are designed especially for these POMDPs (Lovejoy, 1991; Jaakkola et al., 1995; Kaelbling et al., 1995; Parr and Russell, 1995; Cassandra, 1998). A full discussion of these algorithms falls outside the scope of this dissertation.

2.1.1 Further Definitions

We now give some other definitions that will be useful later on. For a more in depth discussion on MDPs, we refer to the book by Puterman (1994).

Definition 2.4 (Finite MDP)**.** *A finite MDP is an MDP with finite state and action sets:* $|S| < \infty$ *and* $|A| < \infty$.

Definition 2.5 (Deterministic MDP)**.** *A deterministic MDP is an MDP where the transitions and reward are deterministic:* $P_{sa}^{s'}$ *equals one for precisely one state* s' *and zero for all other states and* $\forall t : r_{t+1} = R_{s_t a_t}^{s_{t+1}}$.

Definition 2.6 (Undiscounted MDP)**.** *An undiscounted MDP is an MDP where* $\gamma = 1$.

To ensure that the value-based algorithms we will encounter later can handle undiscounted MDPs, it is then usually assumed that the MDP reaches a terminal state with probability one in the limit. Otherwise, the value of a

state can in principle be unbounded, which may cause problems. There are other ways to solve this issue, but we will not into them here.

Definition 2.7 (Stationary MDP)**.** *A stationary MDP is an MDP where every element in the tuple* (S, A, P, R, γ) *is fixed and independent on the time step.*

An example of a non-stationary problem is a problem where there is more than one learning agent and the agents can only observe the behavior of the other agents. Taking the perspective of any one agent and assuming the other agents can also affect the environment, the transition and reward functions can then change over time.

Definition 2.8 (Ergodic MDP)**.** *An ergodic MDP is an MDP where every state can be accessed in a finite number of steps from any other state.*

Ergodic MDPs never restrict the agent to a subset of the state space. This is important for agents that learn from experience, because they can try any action without the possibility of making a mistake that can not be recovered from. State s' is called *accessible* from state s if there is a policy that leads from state s to s' in a finite number of steps. States s and s' are called *communicating* if s is accessible from s' and vice versa. An MDP is called *irreducible* if the set of states forms an equivalence class under this relation, i.e. if any state can be visited from any other state. By definition, an irreducible MDP is ergodic.

Definition 2.9 (Terminal state)**.** *A terminal, or absorbing, state is a state that only transitions to itself, with zero reward.*

Definition 2.10 (Episodic MDP)**.** *An episodic MDP is an MDP with a terminal state that is accessible from any state.*

Note that we only need a single terminal state for any episodic problem, since multiple terminal states would be indistinguishable from each other. If terminal states with different values are desired, one can model this as an equivalent MDP with a single terminal state with different rewards on the incoming transitions. The definition of an episodic MDP does not imply that an agent actually reaches the terminal state, whether this happens may depend on the policy. Furthermore, we will refer to MDPs where the terminal state is only accessible from a subset of the state space as semi-episodic.

Sometimes it is useful to talk about MDPs that do not allow cycles in the resulting Markov chains, for any policy. In a sense, these MDPs are strictly episodic.

Definition 2.11 (Acyclic MDP)**.** *An acyclic MDP is an MDP where for any state s' that is accessible from state s it holds that state s is not accessible from state s'.*

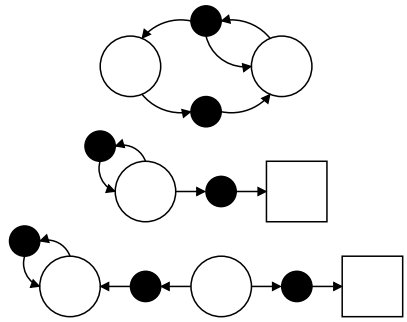

Figure 2.1: Small examples of an ergodic, episodic and semi-episodic MDP.

Figure 2.1 shows an ergodic, an episodic and a semi-episodic MDP. We denote states in such figures with open circles, actions with closed circles and terminal states with squares. Rewards are not shown, but would be shown next to the incoming connections to each state if they were. In the ergodic MDP, the upper action has two outgoing connections. This indicates a stochastic transition: if the agent selects this action it might end up in either of the two states with some non-zero probability. None of the shown MDPs are acyclic.

Since the terminal state never transitions to any other state, an episodic MDP can never be ergodic. In practice, most algorithms are trained on episodic MDPs by placing them back in a starting state after an episode has ended. This can equivalently be modeled by allowing the discount factor to differ for each transition. One can then construct an equivalent MDP with a single action in the terminal state that transitions according to the starting state probability distribution to one of the possible starting states. The transition from the terminal to starting state then should have a discount factor of zero in order for full equivalence between the two models in terms of optimal values and policies. We will not treat such variable-discount MDPs in detail. However, if at any point we refer to an ergodic MDP in theoretical analysis, the analysis will usually also hold for episodic MDPs with little adaptations, as long as these episodic MDPs can be transformed into an equivalent ergodic variable-discount MDP.

2.1.2 Modeling a problem as an MDP

In a large portion of the reinforcement learning literature, it is assumed that an MDP is given and then the goal is to find a suitable algorithm to solve it. However, any problem must be modeled as an MDP before it can be solved and there is usually more than one way to do this. In this subsection, we briefly consider some of the difficulties and design choices that arise from this. For a

more general treatment, one can for instance consult chapter 5 of the book by Powell (2007).

There is no general method to model a problem as an MDP. In fact, often the best model will be dependent on which algorithm one wants to use and the best algorithm for the job in turn depends on the model.

2.1.2.1 State space

Earlier work on MDPs often assumed that the states of the model are in some way easily distilled from the problem that is to be solved (Bellman, 1957; Puterman, 1994). The book by Powell (2007) gives more insight into this issue, but explicitly assumes that the state variables are chosen in such a way that the Markov property holds. Powell defines the state variable as follows:

Definition 2.12 (Definition 5.4.1 from Powell, 2007)**.** *A* state variable *is the minimally dimensioned function of history that is necessary and sufficient to compute the decision function, the transition function, and the contribution function.*

We comment that in many cases this definition is stricter than absolutely necessary. For instance, consider a problem in which there are many different actions and each action has its own stochastic rewards and transitions to new, different realities. Then, applying the definition above in a strict sense would require us to model each of these realities in a different identifiable state. However, when the same action is optimal in all states, modeling the problem with a single abstract state would suffice to find an optimal solution for the actual problem. Naturally, most algorithms would find the optimal action a lot quicker in the simpler single-state model. In other words, we prefer the following definition:

Definition 2.13. *A* state variable *is the minimally dimensioned function of history that is necessary and sufficient to compute the decision function.*

These different definitions additionally show a potential advantage of model-free approaches: since the second definition of the required or desired state variable is weaker, a larger set of models fulfills the requirement. It is possible for a problem to be prohibitively hard to model or even impossible to model precisely for a given state variable, while being far easier to approximately solve. In a sense, human decision making is similar. People often rely on heuristics to successfully make decisions in settings that would be far too hard to solve if all the available state information would be taken into account and one would try to construct a reliable model. These heuristics can be interpreted as the decision function from the definition. Of course, this does

not mean a model-free approach is better in all possible settings and there are other examples in which the use of a model does help to reach better decisions.

If one is modeling a task with a low dimensional, continuous state space there is the choice on whether to discretize the state space in the modeling phase or to use an algorithm that can handle continuous state spaces. Many algorithms that are designed to work in continuous state spaces do little more than first find a suitable discretization of the state space and then solve the resulting discrete MDP with a suitable discrete reinforcement learning algorithm. This is similar as the example before: discretizing the state simplifies the problem. However, the solution in the discretized model may not be optimal in the actual continuous case. We will encounter these issues in more detail in Chapter 7.

2.1.2.2 Action space

The performance of most algorithms is very dependent on the number of available actions, so it may be beneficial to limit this number in the modeling phase, preferably without limiting the quality of the resulting solution. Similar to the state space, one may encounter the choice between using a model with continuous or discrete actions. A large majority of algorithms in the field of reinforcement learning assume discrete, finite action sets. However, in Chapter 7 we will discuss the continuous case.

2.1.2.3 Transitions and after states

The transition function may be the result of interactions with a physical or simulated system and usually arises quite naturally when the models for the state and action spaces are selected. However, these transitions can also be a consideration in the modeling phase. For instance, some work has been done on improving performance of some algorithms by considering the after state, or post-decision state (Powell, 2007).

After states represent the known deterministic result of an action in a certain state. If the MDP is deterministic in its transitions and these transitions are known, an after state is simply equal to the state preceding the present state. However, in many cases the effect of an action may be split into two parts: a known deterministic part and an unknown, possibly stochastic part. In such cases it may be beneficial to process all the known effects of an action, observe the resulting state and only then try to estimate the value of this state. This can be accomplished by constructing after states explicitly in the modeling phase. The resulting MDP can be seen as a copy of the original MDP, but with one added state and action between each action and state. If in the original MDP action a in state s would lead randomly to one of the states in a set X, in the resulting MDP this action leads deterministically to

an after state s' which incorporates all the deterministic effects of the action. In this after state s', the agent then has only a single action which leads randomly to one of the states in X, as in the original process. For completeness, we note that to make the two MDPs fully equivalent one would also need to make small adaptations to the rewards and discount factor. Although these MDPs are equivalent in terms of optimal policies and in terms of the values that corresponding states would have, they are not necessarily equivalent in terms of how long it takes any given algorithm to solve them.

2.1.2.4 Reward function

A very important consideration in the modeling phase is the choice of a reward function. Remarkably, the reward function in the vast majority of reinforcement learning literature is assumed to be given. Some problems have properties that lead naturally to a reward function. For instance, if the goal in a certain problem is to maximize revenue, it is quite natural to use these revenues as rewards. However, in other problems this is less trivial.

For instance, consider a maze problem where we want to use a value-based reinforcement learning algorithm to learn how to escape this maze as quickly as possible. A natural way to model goal states in a reward function is to give a positive reward on reaching the goal state and zero reward on all other steps. In the maze this would translate to a positive reward for exiting the maze and zero reward on each other step. However, this reward function would then require a discount factor strictly lower than one. Otherwise, any action that eventually exits the maze is optimal and there is no incentive to reach the exit quickly. An alternative method would be to set the reward for each transition to some negative value and to use this as an incentive to exit the maze quickly. Which of the two reward models results in the fastest learning can be dependent on which algorithm is used and on the type of exploration.

When selecting a reward function it is important to make sure that the reward function, together with the discount factor, has the following two properties. First, optimizing the discounted cumulative reward should result in the intended behavior. This seems trivial, but in some cases it is not so easy to make sure that this holds. In particular, the most pure formulation of the problem in terms of rewards and discount factor may be prohibitively difficult to solve. In many cases, it can be beneficial to add intermediate rewards to speed up learning, but then care should be taken that these intermediate rewards do not influence the optimal policy. Second, the reward function should have the Markov property. Again, this may seem trivial, but there have been cases in which this requirement was not met. For instance, in the reinforcement learning competition 2009 (Wingate et al., 2009) one of the problem domains featured a simulated helicopter. The goal was to hover the

helicopter for at least 6.000 time steps, which equated to 10 minutes of simulated time. When the helicopter would crash sooner than that, a negative reward was given that was relative to the remaining time until the 10 minutes had passed. Since the number of passed time steps was not part of the state description, this reward function does not have the Markov property. As a result, no method based on value functions performed well, because the problem was ill-defined for these algorithms. This is noticeable in the results of the competition, which show that only evolutionary methods that took the cumulative result of a whole episode into account were successful.

Even a seemingly straightforward problem domain, such as aforementioned example with revenue is less trivial than it may at first seem. For instance, assume that the problem is too large to solve optimally in reasonable time. Then the question is what is more desirable: to reach reasonably high revenues with high probability, or to reach very high revenues with lower probability. Perhaps there are additional aspects that need to be taken into account, such as a minimum amount of revenue that is absolutely necessary in order to stay in business. In other words, if one simply uses revenue as a reward function, a reinforcement learning agent will typically try to optimize a linear function of these reward, but the actual goal may be a non-linear function of the revenue. There may even be certain subgoals that need to be reached within a certain amount of time, introducing the need to balance short-term and long-term returns. This brings us to the last part of the model, the discount factor.

2.1.2.5 Discount factor

The discount factor determines the value of an action or a state, together with the reward function. Most reinforcement learning algorithms optimize the discounted cumulative reward. Even if we compare different algorithms in terms of their performance on average rewards per step, instead of in terms of how they optimize the discounted rewards, in many cases algorithms that use the discounted cumulative reward paradigm outperform algorithms that explicitly try to optimize the average rewards.

It is somewhat surprising to see that often the discount factor is considered a given in the reinforcement learning literature. This differs from the common practice in engineering and control theory, where anything not present in the real world is considered part of the solution or—in other words—of the algorithm. A problem with the discount factor is perhaps that this single value serves different purposes.

First, one can consider it to be part of the problem, or perhaps more accurately of the intended solution. In this case, one considers immediate rewards more important than later rewards and expresses this with a discount factor lower than one. An example of such a case would be a maze problem, where

a positive reward is given on exiting the maze and zero reward is given on every other time step. A discount factor lower than one then implies that the optimal behavior is to exit the maze as quickly as possible. Another important example of such a discount factor occurs in problems in which an interest rate x is present, where one can define the discount factor $\gamma = 1/(1+x)$.

Second, the discount factor is used to ensure that the value of any state or action is finite, assuming that the reward on any step is finite. This is a useful property in the analysis of algorithms which assume uniform boundedness of values. Additionally, it avoids practical numerical issues in settings where the undiscounted values might become very large.

Third, the discount factor can be shown to be of theoretical importance in the convergence analysis of some algorithms. In these cases, it can be shown that the algorithm's convergence rate is dependent on the discount factor. Often, this is the case because the discount factor can be seen to function as a contraction multiplier. The dependence is then such that a lower discount factor implies faster convergence, although this does not necessarily hold in all cases. On the other hand, when average or total rewards are to be optimized, high discount factors often result in a formulation closer to the problem that we actually want to solve. Setting the discount factor too low may result in fast convergence on the thus formulated MDP, but this MDP might not correspond to the actual problem that we want to solve. For example, consider a cleaning robot that has the choice of clearing some rubble for a low reward. If the discount factor is too low, the robot will think it is optimal to clear the rubble even if it is almost out of fuel and it will then not reach its docking station in time. In settings where a range of choices for the discount factor all correspond to the same optimal behavior, different algorithms can prefer different discount factors in terms of their best performance (e.g., Wiering and van Hasselt, 2009).

For a similar discussion on discount factors, we refer to the work by Schwartz (1993), who proposes to use undiscounted average rewards as both a performance criterion and as a goal for the value function. This then removes the need of a discount factor in some problems which might otherwise suffer from potentially infinite values. Average reward MDPs were investigated earlier by Howard (1960) and algorithms to solve these MDPs were analyzed in more detail later by Mahadevan (1996). We will not discuss average reward reinforcement learning in detail in this dissertation, since the discounted reward setting is more extensively researched and better understood.

In summary, we believe the strict separation that is present in some of the literature between the modeling phase and the choice of algorithm is an artificial one. For instance, there is no good reason why it would not be possible for multiple agents to operate on the same problem with different objectives. Agents with short term objectives might then use lower discount factors than agents with long term objectives. It is even possible to specify completely

different objectives by specifying a different reward function for each agent. Such a framework is called a stochastic game (Shapley, 1953) or a Markov game (Zachrisson, 1964) and was investigated in the context of reinforcement learning by Littman (1994), amongst others.

In single agent settings there are also design choices to be made for the reward function and the discount factor. For instance, as mentioned above, in a maze setting one indicates the desired behavior by specifying a reward function that is positive when exiting the maze and zero everywhere else with a discount factor strictly lower than one. Alternatively, one can give a negative reward on every step and use a discount factor equal to one. It is easily checked that in both cases the optimal behavior is the same: exit the maze as soon as possible. However, different algorithms may perform better with different models. This implies that there is no one way to optimally model a problem as an MDP: it depends on the interplay between model and algorithm.

2.1.3 Value Functions

In reinforcement learning, the value of a state or an action plays a central role. In some cases, one may be interested in the value of a certain policy of behavior in a given problem. In most cases, one wants to optimize the total return in terms of cumulative rewards. In this section, we formalize the notion of value in terms of the structure of the MDP. This allows us to talk about the value of a state, action or policy unambiguously in the remainder of this dissertation. In the next section we discuss some methods to learn these values, but first we introduce our notation and definitions.

2.1.3.1 State Values

When we are given an MDP and a policy π, it is possible to determine the value V^π of following this policy when starting in state s. This value is defined as the cumulative discounted reward:

$$V^\pi(s) = E\left\{\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} | s_t = s, \pi\right\} \ . \tag{2.1}$$

Sometimes one is interested in the value of some given policy, but more often we will be interested in maximizing the value. The optimal value of a state is the maximal possible value that can be obtained with any policy. We denote this optimal value with V^*, where

$$V^*(s) = \max_\pi V^\pi(s) \ . \tag{2.2}$$

The goal is then to find the optimal, stationary policy π^* that maximizes the value for each state. By definition

$$V^{\pi^*} = V^* \ . \tag{2.3}$$

The value defined in equation (2.1) can also be defined recursively:

$$\begin{aligned} V^\pi(s) &= E\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s, \pi\right\} \\ &= E\left\{r_{t+1} + \gamma\left(r_{t+2} + \gamma r_{t+3} + \ldots\right) | s_t = s, \pi\right\} \\ &= E\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, \pi\right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P_{sa}^{s'}\left(R_{sa}^{s'} + \gamma V^\pi(s')\right) \ . \end{aligned} \tag{2.4}$$

This then specifies a linear system of $|S|$ equations that can in principle be solved to find the value of each state. Naturally, for this definition we assume the set of states and the set of actions are of finite size. Similarly, the optimal value function can also be described with a recursive definition:

$$V^*(s) = \max_a \sum_{s'} P_{sa}^{s'}\left(R_{sa}^{s'} + \gamma V^*(s')\right) \ . \tag{2.5}$$

Unfortunately, this system of equations is non-linear due to the max operator. Therefore, in general it is harder to solve analytically. Equation (2.5) is knwon as the Bellman optimality equation (Bellman, 1957).

2.1.3.2 Action Values

Similar to state values, we can look at the value of a certain action in a state. Formally, an action value of an action a in a state s under a policy π is defined as the expected cumulative discounted reward when performing that action and following policy π afterward. For historical reasons, action values are often called Q values and we denote these $Q(s,a)$, which is defined as

$$Q^\pi(s,a) = E\left\{\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} | s_t = s, a_t = a\right\} \ . \tag{2.6}$$

Similar to the state value in (2.4), we can also define this value recursively:

$$Q^\pi(s,a) = \sum_{s'} P_{sa}^{s'}\left(R_{sa}^{s'} + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')\right) \ . \tag{2.7}$$

Comparing equations (2.4) and (2.7), we see that the main difference is that the policy is pushed inside expectation of the state transitions. A similar property holds for the optimal action value function, which is defined with the following set of non-linear equations:

$$Q^*(s,a) = \sum_{s'} P_{sa}^{s'}\left(R_{sa}^{s'} + \gamma \max_{a'} Q^*(s',a')\right) \ . \tag{2.8}$$

Storing action values requires more space than storing state values: $|S \times A|$ compare to $|S|$. However, action values have the important advantage over state values that when they are found the optimal policy can be easily constructed, simply by selecting the action with the highest value in each state. In contrast, when only state values are known one must solve equation (2.5) for each state in order to find this optimal action. This difference is especially important for model-free algorithms that approximate the state or action values, since then equation (2.5) can not be solved since P and R are not known.

2.2 Dynamic Programming

Dynamic programming is a collection of methods that assume the knowledge of a full model of the environment and use this model to determine values or optimal policies. Dynamic programming techniques can be seen as precursors of many of the reinforcement learning techniques we will discuss later. In this section, we shortly discuss some of the different methods to solve MDPs. We will only look at methods that in some way use state or action values as defined in the former section. In principle, it is also possible to search for an optimal policy directly, if some proper criterion to be optimized is formulated. Some of these policy-search algorithms will be discussed in Chapter 7.

The two main dynamic programming techniques we will discuss are value iteration and policy iteration, but first we introduce some additional notation and the concept of contraction mappings.

2.2.1 Contraction Mappings

The idea of dynamic programming and indeed of reinforcement learning is that we can iteratively apply some update to the values of states or actions which results in a better approximation of the true state. The obtained approximation can be better because the resulting value is closer to the true value of the current policy, as defined by the Bellman equations (2.1) and (2.6), or because it is closer to the optimal value, as defined in the Bellman optimality equations (2.5) and (2.8). Later, when considering simulation based reinforcement learning, we introduce updates that only improve the values in expectancy instead of on each step.

All these updates have in common that they adapt the value function. These updates are therefore operators that map functions to functions. We denote such operators with $\mathcal{T} : \mathbb{R}^X \to \mathbb{R}^X$, where $\mathbb{R}^X$ denotes a space of bounded real-valued functions over a set X. Using this notation, we can define the operator $\mathcal{T}^\pi : \mathbb{R}^S \to \mathbb{R}^S$ as the mapping

$$(\mathcal{T}^\pi V)(s) = \sum_a \pi(s,a) \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V(s') \right) \quad . \tag{2.9}$$

A fixed point of an operator $\mathcal{T}$ is a function f, such that $\mathcal{T}f = f$. For operator $\mathcal{T}^\pi$ as defined in (2.9) the fixed point is V^π, since by definition

$$(\mathcal{T}^\pi V^\pi)(s) = V^\pi(s) \ , \tag{2.10}$$

of more concisely, $\mathcal{T}^\pi V^\pi = V^\pi$. Similarly, we can define $\mathcal{T}^*$ as

$$\forall s \in S: \quad (\mathcal{T}^* V)(s) = \max_a \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V(s') \right) \ , \tag{2.11}$$

and note that its fixed point is V^*, since from equation (2.5) it follows that $\mathcal{T}^* V^* = V^*$.

An operator $\mathcal{T}: \mathbb{R}^X \to \mathbb{R}^X$ is called a contraction mapping with factor κ if for any two functions $f, g \in \mathbb{R}^X$ the following equation holds:

$$\|\mathcal{T}f - \mathcal{T}g\| \leq \kappa \|f - g\| \ , \tag{2.12}$$

where $\|f\|$ is a sup-norm defined as $\sup_{x \in X} |f(x)|$ and the domain X is implicitly given by the function f. If $\kappa = 1$, the mapping is called a non-expansion. If $\kappa < 1$, the contraction has a unique fixed point, defined by the equation $\mathcal{T}f = f$. Furthermore, this fixed point is guaranteed to be reached by repeatedly applying the mapping (Banach, 1922). Consider the distance $\|\mathcal{T}^n f - \mathcal{T}^n g\|$, where $\mathcal{T}^n$ stands for applying the operator n times, i.e. $\mathcal{T}^2 f = \mathcal{T}(\mathcal{T}f)$, $\mathcal{T}^3 f = \mathcal{T}(\mathcal{T}(\mathcal{T}f))$, and so on. Then, in general we have

$$\|\mathcal{T}^n f - \mathcal{T}^n g\| \leq \kappa \|\mathcal{T}^{n-1} f - \mathcal{T}^{n-1} g\| \leq \ldots \leq \kappa^n \|f - g\| \ .$$

Since this holds for arbitrary functions $f \in \mathbb{R}^X$, this also holds when f is the fixed point. By definition of the fixed point, we then have $\mathcal{T}^n f = f$ and we get

$$\|f - \mathcal{T}^n g\| \leq \kappa^n \|f - g\| \ .$$

Since $g \in \mathbb{R}^X$ is also arbitrary, this means that for any function $g \in \mathbb{R}^X$, we have

$$\lim_{n \to \infty} \|f - \mathcal{T}^n g\| = 0 \ ,$$

when $\kappa < 1$. This shows that repeatedly applying a contraction mapping on any function in its domain results in convergence to the fixed point of the contraction mapping in the limit.

It is easy to prove that if you have a contraction mapping $\mathcal{T}_1$ with factor κ_1 and a contraction mapping $\mathcal{T}_2$ with factor κ_2, the combined operator $\mathcal{T}_1 \mathcal{T}_2$ is also a contraction mapping, with factor $\kappa_1 \kappa_2$:

$$\|\mathcal{T}_1 \mathcal{T}_2 f - \mathcal{T}_1 \mathcal{T}_2 g\| \leq \kappa_1 \|\mathcal{T}_2 f - \mathcal{T}_2 g\| \leq \kappa_1 \kappa_2 \|f - g\| \ .$$

Contraction mappings are important in the theoretical analysis of reinforcement learning and dynamic programming algorithms. If we know that

Algorithm 1 Value Iteration

Initialize V_0, choose ϵ, K.
repeat
  for all $s \in S$ **do**
    $V_{k+1}(s) = (\mathcal{T}^* V_k)(s)$
  end for
until either $\|V_{k+1} - V_k\| < \epsilon$ or $k > K$

an algorithm can be seen as a contraction mapping with a factor lower than one and we know that the fixed point of the contraction is the Bellman optimality equation, then we know that applying the algorithm will lead to optimal values in the limit. If the contraction factor is known, an upper bound on the rate of convergence can be given.

We will now discuss some model-based dynamic programming algorithms. For convenience, we define an operator $\mathscr{E} : \mathbb{R}^S \to \mathbb{R}^{S \times A}$ as follows

$$(\mathscr{E}V)(s,a) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V(s') \right) \quad , \tag{2.13}$$

and we note that $(\mathcal{T}^\pi V)(s) = \sum_a \pi(s,a)(\mathscr{E}V)(s,a)$ and $(\mathcal{T}^* V)(s) = \max_a (\mathscr{E}V)(s,a)$. Since the domain and range of $\mathscr{E}$ are not equal, this operator can not be applied iteratively. It is only defined for notational convenience and can be interpreted as a one step lookahead for a given state value function when the state and action are both given.

2.2.2 Value Iteration

Value iteration is an iterative algorithm that can be used to find the optimal value function, and thus the optimal policy (Bellman, 1957). The idea is to repeatedly apply the operator $\mathcal{T}^*$, as defined in (2.11) on some initial finite value function. The value iteration algorithm is shown in Algorithm 1.

We know $\mathcal{T}^*$ has a unique fixed point in V^* that is obtained in the limit, if we can show that the operator is a contraction with some factor $\kappa < 1$. This is indeed the case:

$$\begin{aligned}
\|\mathcal{T}^* V - \mathcal{T}^* V'\| &= \max_{s \in S} \left| \max_a \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V(s') \right) - \max_a \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V'(s') \right) \right| \\
&\leq \max_{s \in S} \max_a \left| \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V(s') \right) - \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V'(s') \right) \right| \\
&\leq \max_{s \in S} \max_a \max_{s'} \left| \left(R_{sa}^{s'} + \gamma V(s') \right) - \left(R_{sa}^{s'} + \gamma V'(s') \right) \right| \\
&= \max_{s' \in S} \left| \gamma V(s') - \gamma V'(s') \right| \\
&= \gamma \|V - V'\|
\end{aligned}$$

Algorithm 2 Policy Iteration

1: **repeat**
2: Initialize V_0, π_0, $k = 0$, choose K, set $stable = false$.
3: **s**olve $\forall s \in S : V_{k+1}(s) = (\mathcal{T}^{\pi_k} V_{k+1})(s)$
4: **for all** $s \in S$ **do**
5: $\pi_{k+1}(s,a) = \begin{cases} 1/M & \text{if } (\mathcal{E}V)(s,a) = (\mathcal{T}^* V)(s) \ , \\ 0 & \text{otherwise} \ , \end{cases}$
6: **end for**
7: **if** $\pi_{k+1} = \pi_k$ **then**
8: $stable = true$
9: **end if**
10: $k = k + 1$
11: **until** either $stable$ or $k > K$

We identify $\kappa = \gamma$ and therefore value iteration converges to the optimal value function in the limit if $\gamma < 1$. There are some other requirements on the MDP, such as that any state must be reachable. In some specific cases, it can even be guaranteed that the converges occurs after a finite number of iterations.

In this algorithm we introduce two possible stopping criteria. The algorithm stops if either the maximal amount any value is changed in the last iteration is lower than some threshold ϵ, or if the number of iterations transcends K. After the algorithm has terminated, we can find the (approximate) optimal policy with

$$\pi(s,a) = \begin{cases} 1/M & \text{if } (\mathcal{E}V)(s,a) = (\mathcal{T}^* V)(s) \ , \\ 0 & \text{otherwise} \ , \end{cases} \tag{2.14}$$

where M is the number of actions that are optimal in state s according to V.

Value iteration is only applicable if the transition function P and reward function R are known. Even if this is the case, the algorithm can be slow if the state space is too large. For a more thorough treatment, for instance see the books by Puterman (1994) or by Bertsekas (2007).

2.2.3 Policy Iteration

Similar to value iteration is the algorithm called policy iteration (Howard, 1960). The difference is that instead of using the operator $\mathcal{T}^*$, policy iteration uses $\mathcal{T}^\pi$, where π is the current policy. It can be shown that this operator converges to the fixed point V^π. The idea is to find this value function or an approximation thereof, and then use it to improve the current policy. Then, the value function will no longer be accurate and the procedure repeats itself. The algorithm is shown in Algorithm 2.

In line 3 of Algorithm 2, we solve a $|S| \times |S|$ system of linear equations. This can be done in $O(|S|^3)$ time, but this may be too costly if the state space

is large. Alternatively, it is also possible to use an iterative method, as in the value iteration algorithm that computes

$$\forall s \in S: \quad V_{l+1}(s) = (\mathcal{T}^{\pi_k} V_l)(s) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V_l(s') \right) ,$$

for all $s \in S$ until either $\|V_{l+1} - V_l\|$ is smaller than some threshold, or L iterations have been performed. Then $V_{k+1} = V_L$ or, if the threshold condition is met at iteration l, $V_{k+1} = V_l$.

There are two ways in which the policy evaluation step can be relaxed. First, only some of the state values may be updated. The most extreme example of this is when only a single state is updated between each two policy improvement steps. Second, if more than one state is updated, one may update the values only partially towards V^{π_k}, for instance by performing a fixed number of iterations. The most extreme example of this is when only one update is performed. Algorithms that only partially update the function V_k are usually called generalized policy iteration or modified policy iteration (Porteus, 1971; Van Nunen, 1976; Puterman and Shin, 1978; van der Wal, 1978; Sutton and Barto, 1998).

Interestingly, policy iteration generates an improving sequence of policies. This implies that $V^{\pi_{k+1}}(s) \geq V^{\pi_k}(s)$ for all s and all $k \geq 0$ (Puterman, 1994; Bertsekas and Tsitsiklis, 1996). In practice, policy iteration may converge in a fairly low amount of iterations, although like value iteration it can be prohibitively slow is the state space is reasonably large and it requires knowledge of R and P.

2.3 Model-Free Value Learning

The dynamic programming algorithms outlined in the former section have the major disadvantage that they require a model. A good model of the problem that needs to be solved may not be available, although we may have access to a simulated or real physical system with which the agent can interact.

There are two extensions to the dynamic programming methods in the former section that mitigate these issues. First, we can use asynchronous updates that do not update the whole state space in every update. Second, we can use simulated, possibly stochastic updates in place of the models.

2.3.1 Asynchronous Updates

In the methods that were outlined in the former section the whole MDP is taken as a given input and the algorithms process this in some way to output the policy. In most problems, there are parts of the state space that are more interesting than other parts, since they would be visited more often by any good policy, or since the potential rewards are higher. Therefore, it makes

sense to consider only updating a subset of the states in each iteration. The most extreme case of asynchronous updates only updates the value of a single state at a time.

2.3.2 Stochastic Updates

A stochastic update gives the result of a single experience of the agent when it performs an action a in a state s. Stochastic updates are quite naturally combined with asynchronous updates, since if we learn by looking at the experience of an agent, the experiences that are processed by the learning algorithm will correspond to the state the agent happens to be in. Of course, the learning algorithm will not necessarily process each experience immediately when it is available, but in general it is a good idea to at least process some of the information before the whole state space is visited or a terminal state is reached, since this can take prohibitively long.

If we want to estimate the value V^π of a given policy, there are in principle two different types of experience that can be used. These two methods correspond to two different ways to estimate the value that we want to know. In Section 2.1.3 we discussed that $V^\pi(s)$ can equivalently be defined in two different ways:

$$V^\pi(s) = E\left\{\sum_{i=1}^{T-t} \gamma^{i-1} r_{t+i} | s_t = s, \pi\right\} \quad , \tag{2.15}$$

$$V^\pi(s) = E\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, \pi\right\} \quad . \tag{2.16}$$

where for simplicity we assume that the MDP is episodic, such that in at most T steps the terminal state is reached. These definitions can be turned into two different updates, which we will now discuss.

2.3.3 Monte Carlo Methods

For the first update, we use the actual sum of discounted rewards that are obtained until the end of the current episode. The current value of each state s can then be updated with this value in order to get a better estimate of the value of the policy that was followed. Since in general the rewards can contain noise and the state transitions may be stochastic, it is a good idea to average different runs from each state. We denote the start of episode k with t_k and its end with T_k. The update to the value of the state then becomes

$$V_{k+1}(s_t) = V_k(s_t) + \alpha_k(s_t)\left(\sum_{i=1}^{T_k-t} \gamma^{i-1} r_{t+i} - V_k(s_t)\right) \quad . \tag{2.17}$$

Here $\alpha_k(s) \in [0,1]$ is a learning rate parameter and $t_k \leq t < T_k$. Note that we effectively turned equation (2.15) into an update, where we use the learning

rate to average over the different outcomes we observe from a state in different episodes. Such updates are generally referred to as Monte Carlo methods (Michie and Chambers, 1968; Hastings, 1970).

If the MDP and the policy are fully deterministic, we can safely use $\alpha_k(s) = 1$, for all t and s since then any state can only occur once in each episode. If either is not deterministic, the possibility exists that a state is visited more than once in a single policy. One can then choose to update using only the summed rewards after the first visit of the state, resulting in first-visit Monte Carlo, or one can use the summed rewards after each visit, resulting in every-visit Monte Carlo (Singh and Sutton, 1996).

For simplicity assume that no state is visited more than once.[1] Then the return after each episode is clearly an unbiased estimate for V^π. Consider a learning rate of $\alpha_k(s_t) = 1/n_k(s_t)$, where $n_k(s)$ is the number of times state s will have been updated after the current update. Then, for each state $V_k(s)$ will be equal to the average over all the returns after visiting state s in the first k episodes. If the number of times each state is visited increased, the variance of $V_k(s)$ decreases and the in the limit V_k converges: $\lim_{k\to\infty} V_k = V^\pi$.

A disadvantage of Monte Carlo methods is that the returns can have considerable variance. If we reach a state that was already visited many times, we might want to use its value, instead of the considerably noisier rewards that actually result from performing the policy from that state onwards. This can be done with temporal-difference learning, as outlined in the next subsection.

2.3.4 Temporal-Difference Learning

In temporal-difference (TD) learning (Sutton, 1984, 1988), each state value gets updated with a one step Monte Carlo update, using the actual one-step return and the value of the next state. Using the values of consecutive states instead of waiting until the end of an episode is called bootstrapping (Sutton, 1988; Sutton and Barto, 1998) and has a number of advantages. First, it can result in considerably lower variance than using the actual return of only the present episode. Second, Monte Carlo methods are not easily extended to non-episodic tasks, but for TD learning this is no problem.

For Monte Carlo methods equation (2.15) was turned into an update, in TD learning equation (2.16) is used. When the model is not known, we can sample an experience consisting of a state transition and a reward. Such a sample can contain noise, so again we will use an update that averages over the samples with a learning rate. This results in TD learning:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha_t(s_t)\big(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\big) \quad . \tag{2.18}$$

[1] This is the case when the MDP is acyclic, but that would be a stronger assumption. If the MDP is acyclic, no state is visited more than once in an episode for any policy. We only require that this holds for the policy under consideration.

This update can be viewed as a minimizing the expected TD error $E\{\delta_t | s_t = s\}$, where the TD error δ_t is defined as

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \ .$$

When applied in a stationary ergodic MDP, this update can be shown to converge in the sense that $\lim_{t\to\infty} V_t = V^\pi$ (Sutton, 1988; Dayan, 1992; Jaakkola et al., 1994; Tsitsiklis, 1994) as long as the learning rates are chosen such that

$$\forall s : \sum_{t=0}^{\infty} \alpha_t(s) = \infty \ , \quad \sum_{t=0}^{\infty} (\alpha_t(s))^2 < \infty \ . \tag{2.19}$$

These conditions on the learning rates are often referred to as the Robbins-Monro conditions (Robbins and Monro, 1951). The first condition ensures that no matter how poor some samples are, the whole possible value function space stays reachable. The second condition ensures that in the limit the updates become small enough to ensure stability.

The update in (2.18) is not the only possible temporal-difference update. In fact, in itself it is of limited use, since it only gives us the value of the current policy and does not tell us what a good, or optimal policy would be. In the next section we will discuss Q-learning, which can be used to estimate the optimal action value function Q^*, and some variants.

2.3.5 Bias and Variance

We have discussed two different methods to approximate V^π without using a model: Monte Carlo methods and TD learning. We have specified some cases in which TD learning is preferred, such as in non-episodic tasks since Monte Carlo waits until the end of an episode before updating. This does not mean that TD learning is a fully superior algorithm in all cases.

Following Sutton and Singh (1994), we assume for the moment that the MDP is acyclic. We have already mentioned that Monte Carlo estimates are unbiased estimates for V^π. This can easily be checked, because $V^\pi(s)$ is defined as the expected value of the discounted future return when following policy π from state s. Monte Carlo methods simply sample such trajectories, and must therefore be unbiased. On the other hand TD learning introduces bias by using the state value of the next state in its estimate. Especially when this state has not been visited yet, this value will be contentless and arbitrary and will therefore introduce a bias toward the arbitrary initial initialization of the values.

On the other hand, Monte Carlo methods can suffer from considerable variance. Each update uses a sample from the whole trajectory following the state that is to be updated and the variances of the consecutive random rewards are therefore compounded. In other words, if we assume the variance of each reward is σ^2, then the variance of a Monte Carlo update is between σ^2

and $\sigma^2/(1-\gamma)^2$, depending on how many steps the episode on average takes from the state under consideration. For TD learning the variance is then always σ^2, since it only uses a single stochastic reward for its updates. Moreover, the bias of all state values decreases under TD learning, ensuring that the updates become better over time. In contrast, in Monte Carlo methods the expected target of the update has the same variance independent on the number of updates that have already occurred. For an example of an MDP in which this implies faster convergence for TD learning than for Monte Carlo methods see for instance the recent book by Szepesvári (2010). However, in deterministic settings, Monte Carlo will converge must faster than TD learning because then the variance is no problem and it is unbiased.

2.3.6 Eligibility Traces

Many machine learning methods feature a parameter that can affect the trade-off between bias and variance, and reinforcement learning is no exception. This trade-off can be accomplished with a so-called eligibility trace (Sutton, 1984, 1988). For each state, we store an eligibility parameter $e_t(s)$ that indicates how recent this state was visited. The more recent, the higher the eligibility to receive an update to its value. Whenever a state is visited, the eligibility is increased. For all other states the eligibility is decreased. This has the result that updates can be propagated back towards earlier states. The parameter that regulates the decrease of the eligibility traces is denoted by λ, where $\lambda \in [0,1]$. The decrease per time step of the eligibility trace is then $\lambda\gamma$.

If we set λ to zero, we obtain normal TD learning as outlined in the former subsection. Therefore, this method is often referred to as TD(0). On the other hand, if we set λ to one, we obtain Monte Carlo methods, with the difference that the present algorithm can already start updating before an episode has ended and is therefore also applicable in non-episodic tasks. For any $\lambda \in [0,1]$, TD(λ) can be shown to converge to V^π for any fixed policy π with probability one (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Jaakkola et al., 1994).

The algorithm is shown in Algorithm 3. Since a TD learning agent is localized in a state, we need a distribution that tells us the probability of each state for being the starting state. We denote this distribution with $I : S \to [0,1]$, such that $I(s)$ is the probability that the agent starts an episode in state s. For simplicity, we assume that this probability distribution is the same for each episode in episodic tasks. Earlier, we discussed an alternative way to model episodic tasks with $\gamma = 0$ on transitions to a terminal state. If then time t is the last time step of an episode, this would automatically lead to $e_{t+1}(s) = \lambda\gamma_t e_t(s) = 0$ for all s, as required.

In Algorithm 3 we abstract over when to stop updating. This may be when

Algorithm 3 TD(λ)

1: Given γ, π, I and an MDP to act on.
2: Initialize V, $s \sim I$, $\forall s : e(s) = 0$.
3: **repeat**
4: Choose $a \sim \pi(s)$
5: Perform a, observe r and s'
6: $\delta = r + \gamma V(s') - V(s)$
7: $e(s) = 1$
8: **for all** $s \in S$ **do**
9: $V(s) = V(s) + \alpha(s)e(s)\delta$
10: **end for**
11: **if** s' is terminal **then**
12: **for all** $s \in S$ **do**
13: $e(s) = 0$
14: **end for**
15: $s \sim I$
16: **else**
17: **for all** $s \in S$ **do**
18: $e(s) = \lambda\gamma e(s)$
19: **end for**
20: $s = s'$
21: **end if**
22: **until** sufficient convergence

the updates become sufficiently small or after some fixed number of updates. In line 7, we set the trace to one. This method is called replacing traces. Alternatively, we could add one to the trace: $e_{t+1}(s_t) = e_t(s_t) + 1$. This method is called accumulating traces. For a discussion on these two methods we refer to the paper by Singh and Sutton (1996).

Although Algorithm 3 updates the state values after each step, one could also wait until an episode has ended to do so. These updates are then called offline update—or batch updates—whereas updates that are performed after each step are called online updates. It can be shown that when offline updates are used with a proper choice of learning rate, accumulating eligibility traces with $\lambda = 1$ are equivalent to every-visit Monte Carlo and replacing traces with $\lambda = 1$ are equivalent to first-visit Monte Carlo (Singh and Sutton, 1996).

Intuitively, the equivalence between TD(λ) and Monte Carlo can be seen by looking at each consecutive update. First, the value of a state s_t gets the normal TD update. This means it is updated towards $r_{t+1} + \gamma V_t(s_{t+1})$. Then, new information about the value of the next state s_{t+1} is received. The value of this state is updated, but we can also use this information to update the value of s_t. Assume $\lambda = 1$ and assume $\alpha_t(s)$ is equal to some fixed value α for

all s. Then, at time $t+2$ the value of s_t is updated as follows:

$$V_{t+2}(s_t) = V_{t+1}(s_t) + \alpha e_{t+1}(s_t)\big(r_{t+2} + \gamma V_{t+1}(s_{t+2}) - V_{t+1}(s_{t+1})\big) \ .$$

For simplicity, assume for a moment that the MDP is acyclic and that therefore the value of s_{t+2} was not updated since s_t was updated and $V_{t+1}(s_{t+2}) = V_t(s_{t+2})$. Likewise, assume $V_{t+1}(s_{t+1}) = V_t(s_{t+1})$. We can then make the following derivation:

$$\begin{aligned}
V_{t+2}(s_t) &= V_{t+1}(s_t) + \alpha e_{t+1}(s)\Big(r_{t+2} + \gamma V_{t+1}(s_{t+2}) - V_{t+1}(s_{t+1})\Big) \\
&= V_{t+1}(s_t) + \alpha\gamma\Big(r_{t+2} + \gamma V_t(s_{t+2}) - V_t(s_{t+1})\Big) \\
&= V_t(s_t) + \alpha\Big(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\Big) + \alpha\gamma\Big(r_{t+2} + \gamma V_t(s_{t+2}) - V_t(s_{t+1})\Big) \\
&= V_t(s_t) + \alpha\Big(r_{t+1} - V_t(s_t)\Big) + \alpha\gamma\Big(r_{t+2} + \gamma V_t(s_{t+2})\Big) \\
&= V_t(s_t) + \alpha\Big(r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}) - V_t(s_t)\Big) \ .
\end{aligned}$$

We see that this is equivalent to using a two-step update in place of a one-step update as in TD(0). Repeating this procedure, we can arrive at the more general

$$V_{t+n}(s_t) + \alpha\Big(\sum_{m=1}^{n} \gamma^{m-1} r_{t+m} + \gamma^m V_t(s_{t+m}) - V_t(s_t)\Big) \ .$$

It is possible to show the aforementioned equivalence between Monte Carlo and TD(1) under weaker conditions, but we will not do so here. For general $\lambda \in [0,1]$ we can view the total eligibility trace update for a state as a mixture of different n-step updates, where $n \in \{1,2,\ldots\}$ (Sutton and Barto, 1998). Each n-step return is denoted as $R_t^{(n)}$, which is defined as

$$R_t^{(n)} = \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n V_t(s_{t+n}) \ .$$

The total λ return is then

$$R_t^{\lambda} = (1-\lambda) \sum_{i=1}^{T} \lambda^{i-1} R_t^{(n)} \ ,$$

which is a weighted sum of the n-step returns with a normalizing factor of $(1-\lambda)$ and where T is the time step on reaching the terminal state, or $T = \infty$ for non-episodic tasks. For any value of $\lambda \in [0,1]$, TD(λ) can be shown to converge in the limit to V^{π} with probability one (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Jaakkola et al., 1994).

In practice, often intermediate values of λ such as $\lambda = 0.8$ or $\lambda = 0.9$ seem to work best. These values apparently result in a good trade-off between bias and variance in many problems. However, it is not simple to find a general

procedure to set λ, because the ideal setting is problem dependent. For instance, in a fully deterministic problem, there is no variance in the updates and the best method is simply to use a single Monte Carlo estimate and update all the way towards this estimate with $\lambda = 1$ and $\alpha = 1$. In other cases, it is potentially better to use an adaptive trace parameter. The fixed λ parameter is then replaced with a time- and state-dependent parameter $\lambda_t(s)$. It makes sense to set this parameter to one the first time a state is visited and then to decrease it as more information is gathered about the value of this state. Adaptive traces have been explored by Sutton and Singh (1994). This has not immediately resulted in algorithms that are superior in every sense, since the algorithms that are proposed that have better performance on the problems they are tested on are more computationally expensive than ordinary TD(λ).

2.3.7 Fast Eligibility Traces

In Algorithm 3, on every time step the value of all states is considered. One could implement this more efficiently by storing information on which states have non-zero traces, but this means we could still update many different values on each step. In a worst-case setting, the algorithm makes $O(|S|)$ updates every time step.

Wiering and Schmidhuber (1998) have proposed a way to make these updates more efficient by observing that you do not need to update a state value until its value is actually needed again. Therefore, you can wait until updating the value of a state until a transition to this state occurs and its value is needed to determine the temporal-difference error. Furthermore, the trajectory that is used to update all state values is the same for all states. The only difference is the time step on which the relevant state is visited. This implies one can store the value of the whole trajectory of an episode in a single variable.

The resulting algorithm stores two global values L_t and D_t that are updated as follows

$$L_{t+1} = \lambda\gamma L_t \ ,$$
$$D_{t+1} = D_t + \delta_t L_t \ ,$$

where $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$ is the state TD error at time t. If the values are initialized as follows: $L_0 = 1$ and $D_0 = 0$, it is then easily verified by induction that

$$L_t = (\lambda\gamma)^t \ ,$$
$$D_t = \sum_{i=1}^{t} (\lambda\gamma)^i \delta_i \ .$$

Algorithm 4 Fast TD(λ)

Given γ, π, I and an MDP to act on.
Initialize V, $s \sim I$, $D = 0$, $L = 1$, $\forall s : e(s) = 0, d(s) = 0$.
repeat
  if s is terminal **or** $\lambda\gamma L < \epsilon_M$ **then**
    for all $s \in S$ **do**
      $V(s) = V(s) + \alpha(s)e(s)\big(D - d(s)\big)$
      $d(s) = 0$
      $e(s) = 0$
    end for
    $L = 1$
    $D = 0$
    $s \sim I$
  else
    Choose $a \sim \pi(s)$
    Perform a, observe r and s'
    $\delta = r + \gamma V(s') + \gamma\alpha(s')e(s')\big(D - d(s')\big) - V(s)$
    $L = \lambda\gamma L$
    $D = D + L\delta$
    $e(s) = 1/L$
    $s = s'$
    $V(s) = V(s) + \alpha(s)e(s)\big(D - d(s)\big)$
    $d(s) = D$
  end if
until sufficient convergence

The value of $V_0(s_0) + \alpha_0(s_0)D_t$ is then equal to the value of $V_t(s_0)$ if Algorithm 3 was used. In other words, the same state value can be calculated without having to update the value of s_0 on every time step. Of course, this only holds for s_0. Suppose we want to know the update for s_1. Then, we can also use the value of D_t if we note that the total update for s_1 is equal to the total update for s_0, except that δ_0 is not used and the total return for s_0 was multiplied once more with $\lambda\gamma$. Therefore, if D_t is the total error for s_0, then $(D_t - \delta_0)/(\lambda\gamma)$ is the total error for s_1. Equivalently, this can be written as $(D_t - D_1)/L_1$. It can be shown that this holds in general (Wiering and Schmidhuber, 1998), and therefore

$$V_t(s_{t'}) = V_{t'}(s_{t'}) + \alpha_{t'}(s_{t'})\frac{D_t - D_{t'}}{L_{t'}} \ ,$$

for all $0 \leq t' \leq t$. To be able to use this, we need to store $D_{t'}$ for all $s_{t'}$. But this is easy, since that is the value of D at the time step on which that state is visited. Therefore, in addition to the updates to D_t and L_t we store $e_t(s_t) = 1/L_{t+1}$ to use later to update the value of s_t and we store $d_t(s) = D_t$ if $V(s)$ is

updated at time t.

Putting this together yields the fast TD(λ) algorithm, which is shown in Algorithm 4. If one stores the states visited in an episode, one can compute lines 5–9 at most for these states instead of for all states. If further the action selection in line 14 operates in $O(1)$, the whole algorithm uses only $O(1)$ computation per time step.

On line 16, we use a slightly different TD error. The reason for this is that because we postpone the updates, the value of $V(s')$ will not yet have been updated since the last visit to this state. However, we know what the update for this state would be, so we simply add this to the update. This is equivalent to first updating $V(s')$ with the updates until the current time step, as was done in the original approach. To ensure that $V(s)$ is up to date, then another update must occur to the value of state s. The update occurs at line 21. The parameter ϵ_M on line 4 ensures that L never falls below machine precision, which will otherwise often inevitably happen when $\lambda\gamma < 1$.

We conclude this section by noting that the model-free Monte Carlo and temporal-difference algorithms that we have discussed approximate V^π. In most cases, we will be more interested in the optimal values V^* or Q^*, or more specifically, in the optimal policy π^*. In the next section, we discuss methods that can be used to approximate the optimal policy.

2.4 Learning Action Values

In this section, we discuss ways to learn the values of actions. In some cases, we may be interested in the value function Q^π of a given policy π. However, in many cases we will be more interested in the optimal value for each action $Q^*(s,a)$. If the action value function Q^* is known, it is easy to find the optimal policy by simply selecting the highest valued action in each state.

We will start with a short revision of dynamic programming for action values and then we will move to value-based model-free temporal-difference algorithms, which will be the main topic of this dissertation.

2.4.1 Dynamic Programming For Action Values

As explained in Section 2.2, dynamic programming techniques adapt Bellman equations to updates that are applied iteratively. If a single application of this update is interpreted as an operator $\mathcal{T}$, this procedure is guaranteed to find the fixed point if $\mathcal{T}$ is a contraction mapping. If the discount factor γ is smaller than one, $\mathcal{T}^\pi$ as defined in equation (2.9) and $\mathcal{T}^*$ as defined in equation (2.11) are indeed such contraction mappings and applying these operators iteratively respectively yields the fixed points V^π and V^*.

Similar to state values, we define operators that implement the Bellman equations for action values. The operator $\mathcal{T}^\pi : \mathbb{R}^{S\times A} \to \mathbb{R}^{S\times A}$ with fixed point

Q^π is defined as

$$(\mathcal{T}^\pi Q)(s,a) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right) \quad . \tag{2.20}$$

Similarly, we can define $\mathcal{T}^* : \mathbb{R}^{S\times A} \to \mathbb{R}^{S\times A}$ as the operator with fixed point Q^*:

$$(\mathcal{T}^* Q)(s,a) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma \max_{a'} Q(s',a') \right) \quad . \tag{2.21}$$

We use the same notation as for the state value operators with fixed points V^π and V^*. These operators are very similar and it will be clear from the context which one is intended.

It is easy to show that $\mathcal{T}^*$ is a contraction mapping with fixed point Q^*. First we start by confirming that $\mathcal{T}^* Q^* = Q^*$ is simply the Bellman optimality equation for action values, and therefore Q^* is a fixed point for $\mathcal{T}^*$. Furthermore:

$$\begin{aligned}
\|\mathcal{T}^* Q - \mathcal{T}^* Q'\| &= \max_{s\in S} \max_{a\in A(s)} \left| (\mathcal{T}^* Q)(s,a) - (\mathcal{T}^* Q')(s,a) \right| \\
&= \max_{s\in S} \max_{a\in A(s)} \left| \sum_{s'} P_{sa}^{s'} \left(\gamma \max_{a'} Q(s',a') - \gamma \max_{a'} Q'(s',a') \right) \right| \\
&\leq \max_{s'\in S} \left| \gamma \max_{a'} Q(s',a') - \gamma \max_{a'} Q'(s',a') \right| \\
&\leq \max_{s\in S} \max_{a\in A(s)} \gamma \left| Q(s,a) - Q'(s,a) \right| \\
&= \gamma \|Q - Q'\| \quad .
\end{aligned}$$

This holds for all Q, Q'. Since $\mathcal{T}^* Q^* = Q^*$, this implies

$$\|\mathcal{T}^* Q - Q^*\| = \|\mathcal{T}^* Q - \mathcal{T}^* Q^*\| \leq \gamma \|Q - Q^*\| \quad ,$$

which shows the convergence of Q to Q^* under the max norm with a factor of at most γ when applying operator $\mathcal{T}^*$. This means we could implement value iteration with action values to approximate Q^*, with ensured convergence in the limit.

Similarly, $\mathcal{T}^\pi$ is a contraction mapping with factor γ and fixed point Q^π. This can be used as the policy evaluation step in a policy iteration algorithm with action values. The policy improvement step can then be accomplished by using a policy that is greedy in the action values. A greedy policy is defined as follows:

Definition 2.14. *[Greedy policy] An action a is greedy in a state s for an action value function Q if $Q(s,a) = \max_{a'} Q(s,a')$. A policy π is greedy when in all states s the action selection probability $\pi(s,a)$ is equal to zero for all non-greedy actions.*

Multiple actions may be greedy in a given state. Any policy that always selects one of these actions is called greedy, regardless of how the probabilities of selecting these multiple greedy actions are distributed. Naturally, we do require that π is a proper policy distribution in the sense that in all states $\sum_{a \in A(s)} \pi(s,a) = 1$.

Although it is convenient to have action values, dynamic programming with action values suffers from the same limitations as dynamic programming with state values. The most important of these are the requirement of a model and the computational requirements for larger state and action spaces.

Similar to the state values in Section 2.3, we can transform the Bellman equations for action values into iterative, sampled updates. For clarity, we repeat the Bellman equations using the expectancy operator for both Q^* and Q^π and both for a one-step lookahead and for a full episode lookahead, resulting in the following three equations:

$$Q^\pi(s,a) = E\left\{ \sum_{i=1}^{T-t} \gamma^{i-1} r_{t+i} | s_t = s, a_t = a, \pi \right\} , \tag{2.22}$$

$$Q^\pi(s,a) = E\left\{ r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a, \pi \right\} , \tag{2.23}$$

$$Q^*(s,a) = E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a, \pi \right\} . \tag{2.24}$$

We do not specify an equation for a backup over a whole episode for Q^*, since this cannot be sampled, unless the policy is greedy in which case it is equivalent to (2.22).

2.4.2 Monte Carlo Methods and Action Values

Equation (2.22) can be sampled with Monte Carlo methods, similar to how we discussed in the case of state values. The difference with the algorithm for state values is that the result is an approximation for Q^π rather than V^π. The disadvantage in this case of using action values is that all actions that are not selected in an episode are not updated, which implies that if on average there are M actions per state, we may need M times as many episodes to reach the same accuracy as when using state values. This difference in convergence rate is additionally dependent on whether there are actions with a zero probability of being selected, which can therefore effectively be ignored, and whether there are actions with low probability of being selected, which might take quite long to be approximated with reasonable accuracy.

2.4.3 Temporal-Difference Learning: Q-learning and Sarsa

We now turn to consider temporal-difference algorithms that can be distilled from the Bellman equations (2.23) and (2.24). First we consider the update

corresponding to equation (2.23). We assume Q_t is an increasingly good approximation for Q^π. Then, we can sample equation (2.23) and update using a learning rate $\alpha_t(s_t,a_t) \in [0,1]$ to average out stochastic noise to get

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t)+\alpha_t(s_t,a_t)\left(r_{t+1}+\gamma Q_t(s_{t+1},a_{t+1})\right) \ . \quad (2.25)$$

This update was first investigated by Rummery and Niranjan (1994) and is known under the name Sarsa, because it uses an experience sample consisting of the tuple $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$ for each update. Similar to the convergence of TD learning to V^π, it can be shown that this algorithm converges to Q^π, provided that all actions are selected according to a fixed policy π and some technical restrictions on the learning rates. For this reason Sarsa is called an *on-policy* algorithm.

Similarly, we can assume Q is an increasingly good approximation of Q^* and sample (2.24) to get the Q-learning update which was first proposed by Watkins (1989):

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1}+\gamma \max_a Q_t(s_{t+1},a)\right) \ . \quad (2.26)$$

Q-learning can be shown to converge to the optimal value function Q^* (Watkins and Dayan, 1992; Jaakkola et al., 1994; Tsitsiklis, 1994; Littman and Szepesvári, 1996). The conditions for convergence are that in every state every action is eventually selected an infinite amount of times and the learning rates are chosen such that

$$\forall s,a : \sum_{t=0}^{\infty} \alpha_t(s,a) = \infty \ , \quad \sum_{t=0}^{\infty} (\alpha_t(s,a))^2 < \infty \ . \quad (2.27)$$

In a sense these requirements are minimal, since any algorithm must adhere to similar requirements in stochastic environments in order to converge.

Q-learning learns about the optimal policy regardless of the policy that is being followed. For this reason Q-learning is called *off-policy*, which means it learns about one policy while following another policy. Of course, if the environment is stochastic, Q-learning will need to sample each action in every state an infinite number of times to fully average out the noise, but in many cases the optimal policy is learned long before the action values are highly accurate. If the environment is deterministic, it is optimal to set the learning rate equal to one. Then, Q-learning reduces to a form of value iteration, since it performs an asynchronous update equal to (2.21).

If one uses Sarsa with a policy that slowly becomes greedy, Sarsa can also be shown to converge to Q^* (Singh et al., 2000). We will discuss the proof for this statement in the next section, since we will use it later to prove convergence of other temporal-difference algorithms.

2.4.4 Convergence

In some cases, we will prove that an algorithm converges to some fixed point. For instance, it is good to know if it can be proven that the value function converges to the optimal value function for a certain algorithm. To prove such statements, we will use the following lemma that was proposed by Singh et al. (2000). This lemma applies to stochastic processes, such as the Markov chains that are obtained by interaction of an algorithm with an MDP. More general results have been presented in the literature (Szepesvári and Littman, 1999; Bertsekas, 2007), but the lemma below suffices for our purposes.

Lemma 2.1. *Consider a stochastic process* (ζ_t, Δ_t, F_t)*, where* $\zeta_t, \Delta_t, F_t : X \to \mathbb{R}$ *satisfy the equations*

$$\Delta_{t+1}(x_t) = (1 - \zeta_t(x_t))\Delta_t(x_t) + \zeta_t(x_t)F_t(x_t) \ ,$$

where $x_t \in X$ *and* $t = 0, 1, 2, \ldots$*. Let* P_t *be a sequence of increasing* σ*-fields such that* ζ_0 *and* Δ_0 *are* P_0*-measurable and* ζ_t, Δ_t *and* F_{t-1} *are* P_t*-measurable,* $t \geq 1$*. Assume that the following hold:*

1. *the set* X *is finite,*

2. $\zeta_t(x_t) \in [0,1]$*,* $\sum_t \zeta_t(x_t) = \infty$*,* $\sum_t (\zeta_t(x_t))^2 < \infty$ *w.p.1 and* $\forall x \neq x_t : \zeta_t(x) = 0$*,*

3. $||E\{F_t|P_t\}|| \leq \kappa ||\Delta_t|| + c_t$*, where* $\kappa \in [0,1)$ *and* c_t *converges to zero w.p.1,*

4. $Var\{F_t(x_t)|P_t\} \leq K(1 + \kappa||\Delta_t||)^2$*, where* K *is some constant,*

where $||\cdot||$ *denotes a maximum norm. Then* Δ_t *converges to zero with probability one.*

For a proof of this and similar lemmas, we refer to previous work (Jaakkola et al., 1994; Littman and Szepesvári, 1996; Szepesvári and Littman, 1999; Singh et al., 2000; Bertsekas, 2007).

The idea of the lemma is usually to apply it with $X = S \times A$, $\zeta = \alpha$ and $\Delta = Q - Q^*$. The maximum norm specified in the lemma can then be understood as satisfying the following equation:

$$\|\Delta_t\| = \max_s \max_a |Q_t(s,a) - Q^*(s,a)| \ . \tag{2.28}$$

Often, the first, second and fourth assumption are easily adhered to, and to apply the lemma we only need to show that the contraction in the third assumption holds. We will see examples of this later in this dissertation.

2.4.5 Eligibility Traces

Both Sarsa and Q-learning can be extended with eligibility traces. If Sarsa is used to approximate Q^π, convergence to this value function can be shown quite straightforwardly for general $\lambda \in [0,1]$. For $\lambda = 1$, the algorithm reduces to action value-based Monte Carlo, as briefly discussed in Section 2.4.2. For $\lambda = 0$, the algorithm is the Sarsa algorithm described above. For intermediate λ, an action value is updated with a mixture of multi-step returns. In Singh and Sutton (1996) it was suggested to implement replacing traces with $e_t(s_t,a_t) = 1$ and $e_t(s_t,a) = 0$ for all $a \neq a_t$. Accumulating traces simply use $e_t(s_t,a_t) = e_t(s_t,a_t) + 1$.

For Q-learning, eligibility traces are less straightforward. There exist at least three variants. All variants compute something like

$$\forall s \in S, a \in A(s): \quad Q_{t+1}(s,a) = Q_t(s,a) + \alpha_t(s,a) e_t(s,a) \delta_t \ ,$$

but they differ in the definition of δ_t and in the update to $e_t(s,a)$. A naive method simply uses

$$\delta_t = r_{t+1} + \gamma \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t) \ ,$$

for all (s,a). Another variant is Watkins' Q(λ) (Watkins, 1989), that was the first method to be proposed. It performs the same update as the naive methods, but sets all the traces in the current state to zero if an exploratory action is chosen. This ensures that Q-learning only uses trajectories that follow the greedy policy, but this does limit the use of the traces to speed up learning, since all traces are cut short when an exploratory action is taken. The last variant we discuss was proposed by Peng and Williams (1996). This variant uses a different TD error δ'_t for all state-action pairs except the current one. This update for this version is

$$\forall s \in S, a \in A(s), (s,a) \neq (s_t,a_t): \quad Q_{t+1}(s,a) = Q_t(s,a) + \alpha_t(s,a) e_t(s,a) \delta'_t \ ,$$

where

$$\delta'_t = r_{t+1} + \gamma \max_a Q_t(s_{t+1},a) - \max_a Q_t(s_t,a) \ .$$

The current state-action value $Q_t(s_t,a_t)$ is updated simply according to equation (2.26). This update makes intuitive sense, since the sum of consecutive TD errors then telescopes if $\lambda = 1$ and the maximum action in state s_t does not change between t and $t+1$, for instance because updates are deferred until later such as when using offline updates. Then the method yields similar results as Monte Carlo update. This telescoping does not occur for naive Q(λ).

2.4.6 Exploration Techniques

So far we have stated that action values can be used to construct a policy, but we have not discussed in any detail how such a policy is constructed. In

definition 2.14, we defined the greedy policy as the policy that chooses the highest valued action in each state. Therefore, this policy can be constructed by setting the probability of all actions that do correspond to the highest action value to zero. However, this policy is of limited use in combination with a model-free algorithm, since the samples it yields will usually be limited to a small part of the state space. In general, one should balance the exploitation of the knowledge obtained so far, which can be done by choosing greedy actions, and the exploration of the state and action space in order to find new interesting actions.

So, we need policies that explore. The most extreme example is the random policy, which selects a random action on each step. One way to regulated the trade off between exploration and exploitation is to use a parameter $\epsilon \in [0,1]$, which represents the probability of using the random policy. Then a greedy action is selected with probability $(1-\epsilon)$. This yields ϵ-greedy exploration, which is defined as follows:

Definition 2.15 (ϵ-greedy)**.** *A policy is ϵ-greedy if it selects a random action with probability $\epsilon \in [0,1]$ and selects a greedy action with probability $(1-\epsilon)$.*

A requirement for the convergence of some algorithms is that the exploration policy is greedy in the limit with infinite exploration. These properties can be formalized as follows.

Definition 2.16 (Greedy in the limit)**.** *A policy π_t is greedy in the limit with respect to some action value function Q_t if for all states s it holds that*

$$\lim_{t\to\infty} \sum_a \pi_t(s,a) Q_t(s,a) = \max_a Q_t(s,a) \ .$$

Definition 2.17 (Infinite exploration)**.** *A policy π_t ensures infinite exploration if for all s and a it holds that $\sum_0^\infty \pi_t(s,a) = \infty$.*

Note that a policy can be greedy in the limit for a stationary value function Q, but also for a non-stationary value function Q_t. An example of a family of policies that is greedy in the limit with infinite exploration in a finite ergodic MDP is an ϵ-greedy policy where $\epsilon_t = 1/n_t(s_t)^x$, where $n_t(s_t)$ denotes the number of times state s_t was visited in the first t time steps and $x \in (0,1]$.

In practice, ϵ-greedy exploration works fine, but not great. The main problem is that it does not differentiate between potentially good actions that are not greedy at the moment and actions that are known to be worthless. A better type of exploration can take into account the values of different actions. An action with a larger value should have a larger probability of being selected and actions that are known to have very low values may be neglected. One way to do this is to use a so-called Boltzmann distribution that yields the following policy:

$$\pi_t(s,a) = \frac{e^{Q_t(s,a)/\tau}}{\sum_b e^{Q_t(s,b)/\tau}} \ .$$

This definition fulfills the desired properties and requirements we have of a policy: the action selection probabilities sum to one and higher values correspond to larger selection probabilities. The τ parameter is called the temperature and regulates how greedy the policy is. When τ decreases towards zero, the policy becomes more greedy and when it increases toward infinity, the policy becomes more random.

A practical issue associated to consider with the implementation of Boltzmann exploration involves the possibility of numerical problems in some MDPs. When τ is fairly low, $Q_t(s,a)/\tau$ may become quite large. For instance, for $Q_t(s,a) = 10$ and $\tau = 0.1$, the value of $e^{Q_t/\tau}$ is already larger than 10^{43}. To distinguish between actions with fairly close values, the value of τ must be small enough, so it is not an option simply to always set τ above some threshold. Additionally. in some problems the action values may be much larger still. However, the numerical problems can quite easily be avoided with use of the following observation:

$$\frac{e^{x_i/y}}{\sum_j e^{x_j/y}} = \frac{e^{z/y}}{e^{z/y}} \frac{e^{x_i/y}}{\sum_j e^{x_j/y}} = \frac{e^{x_i/y+z/y}}{\sum_j e^{x_j/y+z/y}} = \frac{e^{(x_i+z)/y}}{\sum_j e^{(x_j+z)/y}} \quad ,$$

which holds for all x_i, y and z. This implies that if we add some value to all action values, the policy stays unchanged. It makes sense to use the maximal action value for this, or some constant if the range of action values is approximately known. The adapted Boltzmann exploration then becomes

$$\pi_t(s,a) = \frac{e^{(Q_t(s,a)-\max_c Q_t(s,c))/\tau}}{\sum_b e^{(Q_t(s,b)-\max_c Q_t(s,c))/\tau}} \quad . \tag{2.29}$$

The range of $e^{(Q_t(s,a)-\max_a Q_t(s,a))/\tau}$ is $(0,1]$ compared to $(0,\infty)$ for the unadapted values of $e^{Q_t(s,a)/\tau}$. Therefore, with this adaptation the numerical issues are prevented.

As an illustration of this problem and our proposed solution for it, assume that we are in a state with two actions that the discounted cumulative return for the optimal action is equal to $Q(s,a_1)$, while the discounted cumulative return for the suboptimal action is equal to $Q(s,a_2)$, where by definition $Q(s,a_1) > Q(s,a_2)$. Assume for a moment that we have access to these exact values or to fairly good approximations therefore and we are using Boltzmann exploration. The probability of selecting a_1 is then equal to

$$\pi(s,a_1) = \frac{e^{Q(s,a_1)/\tau}}{e^{Q(s,a_1)/\tau} + e^{Q(s,a_2)/\tau}} \quad .$$

Suppose we want to select the optimal action with a probability of at least p. Then, the value of τ should be at most

$$\tau \leq \frac{Q(s,a_1) - Q(s,a_2)}{\log(p) - \log(1-p)} \quad .$$

If $Q(s,a_1) = 100$ and $Q(s,a_2) = 99$, this means that $\tau \leq 1/\log(9) \approx 0.455$. But then $e^{Q(s,a_1)/\tau} \geq e^{219} \geq 10^{95}$. If we want the policy to become greedy, the value of τ needs to decrease even further and quickly the values used in the naive Boltzmann implementation can become huge.

In contrast, our alternative solution in equation (2.29) uses

$$e^{(Q(s,a_1)-\max_a Q(s,a))/\tau} = e^0 = 1 \ ,$$

for any desired greediness of the policy and for any action values. Using this formulation, it becomes more likely that the value of the suboptimal action falls below machine precision. However, this only happens if the probability of selecting this action falls below this precision, so this is not a problem at all in practice.

2.5 Conclusion

In this chapter, we have introduced a number of core topics in reinforcement learning which will be important in the rest of this dissertation. These include dynamic programming, Monte Carlo methods and temporal-difference learning both for prediction and control. The problem of prediction is defined as the problem of finding the value V^π or Q^π of a given policy. The problem of control is the problem of finding the optimal policy π^*, which can be accomplished by finding the optimal action value function Q^*.

In the next chapters, we will continue with model-free temporal-difference algorithms, such as Q-learning and Sarsa. We will also present some new algorithms and will discuss the properties and merits of all these approaches. However, in Chapter 3 we first discuss an estimation bias that occurs in optimization algorithm such as Q-learning. This discussion is later used to explain why Q-learning can suffer from large overestimation of the action values, which was not realized before.

CHAPTER 3

ESTIMATION BIASES IN MAXIMIZATION

Reinforcement learning for control, the main topic of this dissertation, is concerned with finding the best action in every situation. This naturally implies some maximization process is involved. In some cases, this maximization is explicit, such as in the value iteration algorithm depicted in Algorithm 1 in Chapter 2 that uses the optimal Bellman operator $\mathcal{T}^*$. With explicit maximization, we mean that the algorithm at some point selects the maximal element from some set. Other examples of explicit maximization in reinforcement learning include the Q-learning update in equation (2.26) and (ϵ-)greedy policies, where the greedy action is determined by a maximization over the values of the available actions.

It is straightforward to select the maximum from a finite set with deterministic values. However, if one needs the maximal expected value from a set of random variables and the only available information is a set of noisy estimators for these random variables, the estimation process can be biased. In this chapter we discuss two such biases. The first bias occurs when one simply uses the maximum from the set of estimates as an estimate for the maximal expected value. This bias is positive, which implies that one can expect to overestimate the actual maximal expected value. The second bias occurs when one uses a second set of estimates to get an unbiased approximation for the value of the maximal element from the first set. We will show that this bias is negative. This last result seems to be new. In Chapter 4, we will use these results to show why Q-learning overestimates the action values by large amounts in some settings. In that chapter we will also construct the Double Q-learning algorithm, a variant of Q-learning that uses the second approach and will therefore in some cases underestimate the action values.

3.1 Introduction

In this chapter, we analyze two methods to find an approximation for the maximum expected value of a set of random variables. The first method, which we call the single estimator, simply uses the maximum of a set of estimators as an approximation. A method that uses this approach to approximate the value of the maximum expected value will be positively biased. This result in itself is not new, having been discussed in settings that assumes human agents, such as economics (Van den Steen, 2004) and decision making (Smith and Winkler, 2006). It is a bias related to the so called Winner's Curse in

auctions (Capen et al., 1971; Thaler, 1988) and as we will see, it can be shown to follow from Jensen's inequality (Jensen, 1906).

The second method, which we call the double estimator, uses two estimates for each variable and can therefore uncouple the selection of an estimator and its value. Although this method is simple to implement, we are aware of only one previous paper that discusses a similar approach in the context of reinforcement learning (Mannor et al., 2007). In that paper, a model-building approach is discussed that approximates the reward function R and the state transition function P from observed experiences in order to approximate the value function for MDPs. These estimates will contain noise. Then, when in the control case the value of the maximal action is considered, a similar bias occurs as the one we will shortly discuss in the single estimator. To solve this issue, Mannor et al. (2007) propose to split the experiences used to update the model in a calibration set and a validation set. It is claimed this removes the bias. However, we will shortly prove that this is not the case. Rather, instead of a positive bias, a negative bias can result from this approach. We will prove this below in Lemma 3.3.

The double estimator approach that we will discuss is similar to the practice of using a validation set as part of the training procedure in many machine learning applications. The performance on such a validation set is an unbiased estimate for the actual performance of the algorithm. However, in the context of maximization, the estimate obtained by the double estimator is not an unbiased estimate for the maximum of a stochastic set, as we will show later in this chapter. We do not know of any earlier work that discusses this bias.

In Section 3.2, we will first introduce some notation that will be useful later on. In Sections 3.3 and 3.4, we will discuss the single and double estimator. The remaining sections of this chapter will compare the two approaches and examine how the biases are dependent on the number of samples and the number of variables. In general, Q-learning is slightly better in fully deterministic enviroments, but in Chapters 4 and 5 we will see that in noisy settings Double Q-learning often performs better. Because the double estimator approach that we discuss in this chapter is not unbiased, there also exist MDPs in which the related Double Q-learning algorithm performs less well because of the estimator bias. However, we will see in this chapter that the bias of the double estimator is often less than the bias of the single estimator approach that Q-learning is based on. As a result, we will see that in many settings Double Q-learning performs far better than Q-learning.

3.2 Preliminaries

Let $Y = \{Y_1, \ldots, Y_M\}$ denote a set of M random variables. Throughout this chapter, we will use M to stand for the size of a set of random variables. When applicable, we use N to denote the number of samples that are observed. The main general goal is to determine the maximal expected value of the set Y:

$$\max_i E\{Y_i\} \ .$$

Without knowledge of the functional form and parameters of the underlying distributions of the variables in Y, it is impossible to determine this value exactly. However, we can approximate this value by using sample estimate to construct approximations for $E\{Y_i\}$ for all i.

3.2.1 Samples and Estimators

Let $X = \bigcup_{i=1}^{M} X_i$ denote a set of samples, where X_i is the subset containing samples for the variable in Y_i. We assume that the samples in X_i are independent and identically distributed (iid) for all i. Furthermore, we assume that each sample in $x \in X_i$ is an unbiased estimate for Y_i, such that

$$E\{x | x \in X_i\} = E\{Y_i\} \ .$$

In the literature, the expected value is often called the mean and denoted with μ. We will use the same convention here, such that for all i

$$\mu_i = E\{Y_i\} \ .$$

Unbiased estimates for the expected values can be obtained, simply by computing the sample average for each variable:

$$\mu_i = E\{m_i\} \approx m_i(X) \stackrel{\text{def}}{=} \frac{1}{|X_i|} \sum_{x \in X_i} x \ .$$

Here m_i denotes an unbiased estimator for variable Y_i and $m_i(X)$ denotes the estimate resulting from this estimator for a given set of samples X. In other word, $m_i(X)$ is an unbiased noisy estimate for μ_i.

For Q-learning, the samples we will consider are the action values $Q_t(s,a)$. In a reinforcement learning setting, it may seem like a strong assumption that the samples are iid, but we stress that one should interpret $Q_t(s,a)$ as a sample for the action value at time t under the assumption of a certain algorithm. If the experiment is run multiple times, these action values can indeed be interpreted as iid samples for the expected value of the action value at this time step, given the problem and the algorithm that is used to update the values.

The approximation of m_i is unbiased since every sample $x \in X_i$ is itself an unbiased estimate for the value of μ_i. If for simplicity we assume a fixed sample size $N_i = |X_i|$, this follows easily from the linearity of the expectancy operator:

$$E\{m_i\} = E\left\{\frac{1}{N_i}\sum_{x\in X_i} x\right\} = \frac{1}{N_i}\sum_{x\in X_i} E\{x | x \in X_i\} = \frac{1}{N_i}\sum_{k=1}^{N_i} \mu_i = \mu_i \ .$$

Any error in the approximations therefore results solely from the variance in the estimators. As we obtain more samples the variance of the estimators will decrease and therefore the error will decrease as well. For simplicity, we will usually assume that X_i is non-empty for all i, such that $N_i > 0$.

As a special case, we might have a single estimation sample x_i for each element in Y. Then, we can simply use the value of each sample as an unbiased estimate for value of the corresponding element, such that $m_i(X) = x_i$ and:

$$E\{x_i\} = \mu_i \ .$$

We will consider this special case when we discuss the reinforcement learning setting in Chapter 4.

3.2.2 Optimal and Maximal Estimators

We define an *optimal* estimator as an estimator that corresponds to a random variable that has the highest expected value.

Definition 3.1. *[Optimal estimators] An estimator m_j and the corresponding random variable Y_j with expected value $\mu_j = E\{Y_j\}$ are called optimal for a given set of random variables Y if $\mu_j = \max_i \mu_i$. An index $j \in \{1,\ldots,M\}$ is called optimal if the corresponding random variable Y_j is optimal. The set of optimal indices is denoted by $\mathcal{O}$ and is thus defined by*

$$\mathcal{O} \stackrel{\text{def}}{=} \left\{ j \middle| \mu_j = \max_i \mu_i \right\} \ . \tag{3.1}$$

Note that there are potentially multiple optimal estimators and that the optimality of an estimator is independent on the set of samples X and the corresponding values of the estimates $m_i(X)$.

The problem of course is that it is usually unknown which estimators are optimal. Rather, we can observe which estimator has the highest value of all estimators. For a given set of samples, an estimator is called *maximal* if its estimate is at least as large as all the other estimates.

Definition 3.2. *[Maximal estimators] An estimator m_j is called maximal for a given set of samples X if the corresponding estimate $m_j(X)$ is at least as large as all the other estimates, i.e. if $m_j(X) \geq m_i(X)$ for all i. An index $j \in \{1,\ldots,M\}$*

is called maximal if the corresponding estimator is maximal and the set of maximal indices is denoted by

$$\mathscr{M}(X) \stackrel{\text{def}}{=} \left\{ j \,\middle|\, m_j(X) = \max_i m_i(X) \right\} \quad . \tag{3.2}$$

Note and that an optimal estimator is not necessarily a maximal estimator and a maximal estimator is not necessarily an optimal estimator. Furthermore, in contrast with the set of optimal indices, the set of maximal indices is dependent on the set of samples.

If any optimal index has a non-zero a priori probability of not being maximal, this property holds for all optimal estimators. Equivalently, there exists an optimal estimator that is maximal for any set of samples if and only if all optimal estimators are maximal for any set of samples. This result will be used later to show in which cases the maximal estimate is a biased estimator for the maximal expected value. We prove this in the following lemma.

Lemma 3.1. *Let $m = \{m_1, \ldots, m_M\}$ denote a set of unbiased independent estimators for a set of independent random variables $Y = \{Y_1, \ldots, Y_M\}$ with expected values $\mu_1, \ldots, \mu_M$, such that $E\{m_i\} = \mu_i = E\{Y_i\}$, for all $i \in \{1, \ldots, M\}$. Then, if at least one of the optimal estimators is not maximal for at least one possible sample set, then all optimal estimators are not maximal for at least one possible sample set, i.e.*

$$(\exists j \in \mathscr{O} : P(j \in \mathscr{M}) < 1) \leftrightarrow (\forall j : P(j \in \mathscr{M}) < 1) \quad ,$$

where the probabilities span over all possible sample sets.

Equivalently, an optimal estimator is maximal for all possible sample sets if and only if all optimal estimators are maximal for all possible sample sets.

$$(\exists j : P(j \in \mathscr{M}) = 1) \leftrightarrow (\forall j \in \mathscr{O} : P(j \in \mathscr{M}) = 1) \quad .$$

The proof for Lemma 3.1 is given in Section 3.10.1.

3.2.3 Probability Distributions

We will use the following notations, regarding the probability distributions of the random variables: $\mathscr{D}_i$ denotes the measurable domain of the random variable Y_i, $f_i : \mathscr{D}_i \to \mathbb{R}$ denotes the corresponding probability density function (PDF) and $F_i : \mathscr{D}_i \to [0,1]$ denotes the cumulative distribution function (CDF) of this PDF. Often, one can assume that the domain $\mathscr{D}_i$ is the set of real-values numbers $\mathbb{R}$. Then, the CDF of Y_i is given by

$$F_i(x) = \int_{-\infty}^{x} f_i(y)\, dy \quad , \tag{3.3}$$

and the expected value of Y_i is given by

$$\mu_i = \int_{-\infty}^{\infty} x\, f_i(x)\, dx \quad .$$

The maximum expected value that we are looking for can be expressed in closed form in terms of the underlying PDFs as

$$\max_i \mu_i = \max_i \int_{-\infty}^{\infty} x\, f_i(x)\, dx \quad , \tag{3.4}$$

where, as noted before, in general the problem is that the distributions f_i are not known.

Later in this chapter, we will consider some specific distributions such as the uniform distribution $u(x,y,z)$ and the Gaussian distribution $g(x,m,\sigma^2)$. These PDFs are defined by

$$u(x,y,z) = \begin{cases} \frac{1}{z-y} & \text{if } y \le x \le z \quad , \\ 0 & \text{otherwise} \quad , \end{cases} \tag{3.5}$$

$$g(x,m,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-m)^2/(2\sigma^2)} \quad . \tag{3.6}$$

The corresponding CDFs will be denoted $U(x,y,z)$ and $G(x,m,\sigma^2)$, respectively. Their definitions follow from (3.3) and the definitions of the corresponding PDFs.

The expected value of an estimator is defined over all possible sets of samples

$$E\{m_i\} = \int_{\mathscr{P}(D_i)} P(X) m_i(X)\, dX \quad ,$$

where $\mathscr{P}(\mathscr{D}_i)$ denotes the power set of the domain $\mathscr{D}_i$ and $P(X)$ is the probability on observing sample set X. We will often discuss the expected value of an estimator, given an unknown set of samples of a given size. Then, the integral that defines the expected value is assumed to span over the family of all possible sets of samples of this size. To avoid cluttering the notation, we use the same notation $E\{m_i\}$ in this case and assume the size of the sample set is clear from the context. As discussed, for any non-empty set of samples X_i, we have $E\{m_i\} = \mu_i$ where we assume the estimator $m_i(X)$ is the sample average of the set X_i. The PDF and CDF of the i^{th} estimator will be denoted f_i^m and F_i^m. These functions depend on the size of the sample set X_i, which will be assumed to be clear from the context. In general, f_i^m is equal to f_i only when only a single sample has been obtained for the random variable Y_i. As the number of samples increases, by the central limit theorem (de Moivre, 1718; Laplace, 1810; Liapunov, 1901) the variance of f_i^m will become smaller and the distribution will become more strongly peaked close to μ_i. We will investigate the effect of this for a specific setting in Section 3.7.

The next two sections give two methods that use estimators to approximate the maximum expected value.

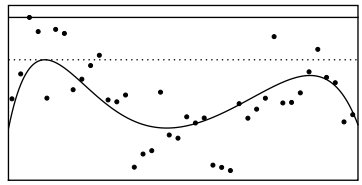

Figure 3.1: This figure depicts some arbitrary function and noisy unbiased samples of this function. The horizontal solid line runs through the largest noisy sample. The horizontal dotted line runs through the maximum of the function, which is the maximal expected value. For this sample, the maximal noisy sample is clearly much larger than the maximum expected value.

3.3 The Single Estimator

One fairly obvious way to approximate the maximum expected value $\max_i \mu_i$ is to simply use the value of the maximal estimator:

$$\max_i \mu_i = \max_i E\{m_i\} \approx \max_i m_i(X) \ . \tag{3.7}$$

Because we will contrast this method later with a method that uses two estimators for each variable, we call this method the *single estimator*.

The single estimator method is often used in practice. For instance, Q-learning uses this method to approximate the value of the next state by maximizing over the estimated action values in that state. In this case, the action values in the next state can be interpreted as random variables that are dependent on the MDP and the past experiences. For a given MDP and Monte Carlo chain of experiences, the action values may differ from one run of the experiment to the next if the MDP contains noise. Then, the action values can be interpreted as unbiased samples of random variables that represent the expected action values for Q-learning on the given MDP and time step. This setting and the resulting overestimation that Q-learning can experience is discussed at length in Chapter 4. In this section, we discuss the single estimator approach in general.

3.3.1 The Overestimation of the Single Estimator

We will formally discuss and prove the overestimation bias of the single estimator. To get an intuition, observe Figure 3.1. This figure shows some arbitrary function and noisy samples of many points of this function. These noisy samples were created simply by adding Gaussian noise to the function at all the sampled points. Even though each sample is unbiased, the maximum sample will often be larger than the maximum of the function.

The maximal estimator $\max_i m_i(X)$ is distributed according to some PDF $f^m_{\max}$ that is dependent on the PDFs of the estimators f^m_i. To determine this PDF, consider the CDF $F^m_{\max}(x)$, which gives the probability that the maximum estimate is lower or equal to x. This probability is equal to the probability that all the estimates are lower or equal to x:

$$F^m_{\max}(x) \stackrel{\text{def}}{=} P(\max_i m_i \leq x) = \prod_{i=1}^{M} P(m_i \leq x) \stackrel{\text{def}}{=} \prod_{i=1}^{M} F^m_i(x) \ .$$

This means that the PDF of the maximal estimate is equal to

$$f^m_{\max}(x) \stackrel{\text{def}}{=} \frac{d}{dx} F^m_{\max}(x) = \frac{d}{dx} \prod_{i=1}^{M} F^m_i(x) = \sum_{i=1}^{M} f^m_i(x) \prod_{j \neq i}^{M} F^m_j(x) \ .$$

It is straightforward to show that the maximal estimate $\max_i m_i(X)$ for a set of samples X is an unbiased estimate for the expected value of the maximal estimator $E\{\max_i m_i\}$. However, we are interested in the maximal expected value $\max_i E\{m_i\} = \max_i \mu_i$, where the order of the max operator and the expectancy operator is the other way around. Since the max operator is nonlinear, these quantities are in general not equal. Therefore, the maximal estimator $\max_i m_i(X)$ is a biased estimate for $\max_i \mu_i$. The expected value of the approximation by the single estimator is given by

$$E\{\max_i m_i\} = \int_{-\infty}^{\infty} x \, f^m_{\max}(x) \, dx = \sum_{i=1}^{M} \int_{-\infty}^{\infty} x \, f^m_i(x) \prod_{j \neq i}^{M} F^m_j(x) \, dx \ . \tag{3.8}$$

Indeed, this closed form expression is quite different from the maximal expected value, as shown in (3.4). We will prove formally that the value in (3.8) is larger or equal to the maximal expected value in (3.4) and we will give general conditions under which it is strictly larger, but first we give a small example.

Consider two standard dice. Each die yields an random integer between 1 and 6 with equal probabilities, resulting in an expected value of $3\frac{1}{2}$. We throw both dice and examine the value for the highest die. There is only a 1 in 36 probability that both dice are equal to 1, but in 11 out of the 36 possible combinations the maximum die will be 6. If we denote the outcome of both dice as random variables Y_1 and Y_2, this indicates that the expected maximum value $E\{\max\{Y_1, Y_2\}\}$ is larger than the maximum expected value $\max\{E\{Y_1\}, E\{Y_2\}\}$. Because the exact probabilities are known, we can easily calculate the difference:

$$\begin{aligned} E\{\max\{Y_1, Y_2\}\} = 1\frac{1}{36} + 2\frac{3}{36} + \ldots + 6\frac{11}{36} &= 4\frac{17}{36} \\ &> 3\frac{1}{2} = \max\{E\{Y_1\}, E\{Y_2\}\} \ . \end{aligned}$$

In the following lemma we prove in a more general sense that the approximation in (3.7) can result in an overestimation. This lemma is a generalization of Proposition 1 in Smith and Winkler (2006).

Lemma 3.2. *Let $Y = \{Y_1, \ldots, Y_M\}$ be a set of random variables with expected values $\mu_i, \ldots, \mu_M$ and let $m = \{m_1, \ldots, m_M\}$ be a set of unbiased estimators such that $E\{m_i\} = \mu_i$, for all i. Assume that a set of samples X contains at least one sample for each of the variables in Y. Let $\mathcal{O}$ be the set of optimal estimators as defined in Definition 3.1 and let $\mathcal{M}(X)$ be the set of maximal indices for X as defined in Definition 3.2. Then*

$$\forall j \in \mathcal{O}: \quad E\{\max_i m_i\} \geq E\{m_j\} = \mu_j = \max_i \mu_i \ . \tag{3.9}$$

Furthermore, the inequality is strict if and only if there is a non-zero probability that any optimal index j is not maximal, i.e.

$$E\{\max_i m_i\} > \max_i \mu_i \quad \leftrightarrow \quad \exists j \in \mathcal{O} : P(j \notin \mathcal{M}) > 0 \ . \tag{3.10}$$

We proof this lemma in Section 3.10.2. Equation (3.9) can also be interpreted as a consequence of Jensen's inequality (Jensen, 1906), that states that

$$E\{f(X)\} \geq f(E\{X\}) \ , \tag{3.11}$$

for any concave function. The max operator is linear when always the same estimators are maximal. This is only the case when there is only one estimator that is maximal for all possible sets of samples, or when all optimal estimators are constant and no suboptimal estimator can yield larger estimates than the optimal estimators. In all other cases, the max operator is strictly concave and the inequalities in (3.9) and (3.11) are strict.

Going back to the example with the two dice, we can conclude that both dice are optimal and that each die has a non-zero probability of not being the maximal die. And indeed we have seen that the inequality between the expected maximal value and the maximal expected value was strict. We will discuss other, more general examples later in this chapter.

3.3.2 An Upper Bound on the Size of the Overestimation

Uppers bounds for the expected overestimation have been established in previous work on order statistics (Clark, 1961; David and Nagaraja, 2003; Arnold et al., 2008). For instance, a relatively simply upper bound for M random variables with arbitrary expected values and variances is given by Aven (1985) as

$$E\{\max_i m_i\} \leq \max_i \mu_i + \sqrt{\frac{M-1}{M} \sum_i^M \sigma_i^2} \ ,$$

where σ_i^2 is the variance of the estimator m_i. A special case of this bound is found when the estimators are iid such that $E\{m_i\} = \max_i \mu_i$ and variances $\sigma_i = \sigma$ for all i. Then

$$E\{\max_i m_i\} \leq \max_i \mu_i + \sigma\sqrt{M-1} \ . \tag{3.12}$$

This bound is tight, since a distribution can be constructed that achieves it (Arnold and Groeneveld, 1979). The fact that the bound is tight when the variables have identical means indicates that this is a worst case setting for the single estimator. Others bounds have been formulated (e.g., Bertsimas et al., 2006), but we are mainly interested in the linear dependence of the bias on $\sigma\sqrt{M}$, which is the same in those bounds.

For specific distributions, the expected overestimation of the single estimator can be calculated or approximated numerically, using (3.8). A non-trivial general lower bound for (3.8) does not exist. General results such as Hoeffding's bound (Hoeffding, 1963) and Chebyshev's inequality can be used to find upper bounds on the probability that a random variable deviates more than a specified amount from its expected value. However, to get a general lower bound on the expected overestimation, we would additionally need lower bounds on these expected deviations. Non-trivial lower bounds are not available in general, since there exist combinations of PDFs with positive variance for which no overestimations occur. Trying to establish a tight non-trivial lower bound for the expected overestimation based on specific properties of the underlying distributions is beyond the scope of this dissertation. However, later in this chapter, we will investigate approximations for the expected overestimations for some specific distributions.

3.4 The Double Estimator

We will later see that the overestimation that results from the single estimator approach can have a large negative impact on algorithms that use this method, such as Q-learning. Therefore, we will look at an alternative method to approximate the value of $\max_i \mu_i$.

3.4.1 The Definition of the Double Estimator

We obtain this second approximation by using two sets of estimators $m^A = \{m_1^A, \ldots, m_M^A\}$ and $m^B = \{m_1^B, \ldots, m_M^B\}$. Therefore, we will refer to this method as the *double estimator*. We will use this approach in Chapter 4 to construct an alternative reinforcement learning algorithm that we will call Double Q-learning. Here we explain how this method approximates the value of the maximal expected value $\max_i E\{X_i\}$.

Both sets of estimators are updated with a subset of the samples we draw, such that $X = X^A \cup X^B$ and $X^A \cap X^B = \emptyset$ and

$$m_i^A(X) = \frac{1}{|X_i^A|} \sum_{x \in X_i^A} x \ , \qquad m_i^B(X) = \frac{1}{|X_i^B|} \sum_{x \in X_i^B} x \ .$$

Like the single estimator m_i, both m_i^A and m_i^B are unbiased if we assume that the samples themselves are unbiased and they are split in a proper manner, for instance randomly, over the two sets of estimators. Then

$$E\{m_i^A\} = E\{m_i^B\} = \mu_i \ .$$

The idea of the double estimator is to make use of the two unbiased sets of estimators as follows. We determine our best guess for the maximal index by selecting the maximal element from one set, but then we use the other set to determine the value of this element. This way, we avoid the overestimation of the single estimator. Even if we select an element because it by chance received a vastly overestimated value, the value for this element from the second set of estimators will be unbiased, although of course it can be noisy. Unfortunately, this does not mean the value we obtain is an unbiased approximation for the maximal expected value. The reason for the bias of the double estimator is that there is a probability that the index we select is not an optimal index. Then, although the obtained value is unbiased for this index, it can be an underestimation for the desired maximal expected value. We formalize and prove these statements in the remainder of this section.

3.4.2 The Underestimation of the Double Estimator

Before we prove formally that the double estimator is biased, we can observe the reason for this bias in Figure 3.1. The horizontal solid line goed through the maximal noisy sample. However, note that this sample is slightly to the left of the position of the maximum of function. If we then obtain an unbiased sample by using a second sample of the element that corresponds to the maximal sample, we will obtain an unbiased estimate of the function value at that position. This means we obtain an unbiased sample of a value that is slightly lower than the actual maximum of the function and therefore we can underestimation this value.

Let $\mathcal{M}^A(X)$ be the set of maximal indices in $m^A(X)$, where maximal indices are further defined as in Definition 3.2. By Lemma 3.2, we have $E\{m_j^A | j \in \mathcal{M}^A\} \geq \max_i \mu_i$. Since m^B is an independent unbiased set of estimators, we have $E\{m_j^B\} = \mu_j$ for all j, including all $j \in \mathcal{M}^A$. Let $a^* \in \mathcal{M}^A$ be maximal for m^A, such that

$$m_{a^*}^A(X) \stackrel{\text{def}}{=} \max_i m_i^A(X) \ . \tag{3.13}$$

Then we can use $m^B_{a^*}$ as an estimate for $\max_i E\{m^B_i\}$ and therefore also for $\max_i \mu_i$ and we obtain

$$\max_i \mu_i = \max_i E\{m^B_i\} \approx m^B_{a^*} \quad . \tag{3.14}$$

If there are multiple estimators that maximize m^A, we can for instance pick one at random, or we can determine the value of all maximal estimators and average these, to obtain

$$\max_i \mu_i = \max_i E\{m^B_i\} \approx \frac{1}{|\mathcal{M}^A|} \sum_{j \in \mathcal{M}^A} m^B_j \quad .$$

Compare the approximation in (3.14) to the approximation by the single estimator as given in (3.7). In the single estimator, the approximation is obtained by taking a maximal estimator and then using its estimate as an approximation. In Lemma 3.2, this approximation was shown to be positively biased. In the double estimator, the maximal estimator in m^A is selected, but then an unbiased estimate for the expected value of this estimator is obtained by using the corresponding estimate in m^B. Although $m^B_{a^*}$ is a unbiased estimate for $E\{X_{a^*}\}$, there is a non-zero probability that a^* is not optimal. Therefore, the double estimator can suffer from underestimations. We will prove this formally below in Lemma 3.3. Similar to the single estimator, as we gain more samples for each estimator the variance of the estimators will decrease. In the limit, $m^A_i(X)$ and $m^B_i(X)$ converge to μ_i for all i and therefore the approximation in (3.14) also converges to the correct result.

For simplicity, assume for a moment that the underlying probability distributions are continuous. Then the probability that two samples have the same value is zero. This implies that $\mathcal{M}^A(X)$ contains a single element with probability one for any X and thus a^* is uniquely defined and $P(j = a^*) = P(j \in \mathcal{M}^A)$ for all j. The probability that an arbitrary index j is maximal in m^A is equal to the probability that all other estimators in m^A give lower or equal estimates than m^A_j. Thus $m^A_j(X) = x$ is maximal for any value x with probability $\prod_{i \neq j}^M P(m^A_i \leq x)$. Integrating out all possible values for x gives

$$\begin{aligned} P(j = a^*) &= \int_{-\infty}^{\infty} P(m^A_j = x) \prod_{i \neq j}^{M} P(m^A_i \leq x)\, dx \\ &= \int_{-\infty}^{\infty} f^A_j(x) \prod_{i \neq j}^{M} F^A_i(x)\, dx \quad , \end{aligned}$$

where f^A_i and F^A_i are the PDF and CDF of estimator m^A_i. The expected value

of the approximation by the double estimator can then be given by

$$\sum_{j=1}^{M} P(j = a^*) E\left\{m_j^B\right\} = \sum_{j=1}^{M} P(j = a^*)\mu_j$$
$$= \sum_{j=1}^{M} \mu_j \int_{-\infty}^{\infty} f_j^A(x) \prod_{i \neq j}^{M} F_i^A(x)\, dx \quad . \tag{3.15}$$

Note that the difference to (3.8) consist mainly of the presence of μ_j outside the integral, in place of x within the integral. The single estimator can overestimate since x is in the integral and correlates with the monotonically increasing product $\prod_{i \neq j} F_i^m(x)$. The double estimator underestimates because its approximation is a weighted average of the expected values μ_j, where the weights correspond to the probabilities that j is maximal for all j. A weighted average of the values in a set can of course never be larger than the largest value in the set. If at least one of the weights for suboptimal estimators is positive, the inequality is strict and the double estimator can underestimate.

For discrete PDFs, the formulae are slightly different, since we would have to take into account the probability that two or more estimators have the same value. Also, the integrals should then be replaced with sums over all the possible values of x. These changes are relatively straightforward.

Returning to the two dice, the probability for each die to yield the highest estimate is equal at $\frac{1}{2}$ if we assume ties are broken randomly. Then, the value in (3.15) is equal to

$$P(j = 1)E\{Y_1\} + P(j = 2)E\{Y_2\} = \frac{1}{2}(3\frac{1}{2}) + \frac{1}{2}(3\frac{1}{2}) = 3\frac{1}{2} \quad .$$

In other words, in contrast to the approximation by the single estimator, the approximation by the double estimator is unbiased for the case of two standard dice. In fact, when the expected values of the random variables are equal, the double estimator will always yield an unbiased estimate, as we will prove below. However, in the following lemma, we show the estimate $E\{m_{a^*}^B\}$ is not an unbiased estimate of $\max_i \mu_i$ in general. This lemma holds both for continuous and discrete random variables.

Lemma 3.3. *Let $Y = \{Y_1, \ldots, Y_M\}$ be a set of random variables with expected values $\mu_1, \ldots, \mu_M$ and let $m^A = \{m_1^A, \ldots, m_M^A\}$ and $m^B = \{m_1^B, \ldots, m_M^B\}$ be two sets of unbiased estimators such that $E\{m_i^A\} = E\{m_i^B\} = \mu_i$ for all i. Let $a^* \in \mathcal{M}^A(X)$ denote a maximal element in $m^A(X)$ and let $\mathcal{O}$ be the set of optimal indices, as defined in Definition 3.1. Then*

$$E\{m_j^B | j \in \mathcal{M}^A\} = E\{Y_{a^*}\} \leq \max_i \mu_i \quad . \tag{3.16}$$

Furthermore, the inequality is strict if and only if there is a non-zero probability that a^* is suboptimal:

$$\left(E\{m_j^B | j \in \mathcal{M}^A\} < \max_i \mu_i\right) \quad \leftrightarrow \quad \left(\exists j \notin \mathcal{O} : P(j \in \mathcal{M}^A) > 0\right) \ . \tag{3.17}$$

The proof for Lemma 3.3 is given in Section 3.10.3. The lemma shows that the approximation by the double estimator gives an underestimation if and only if there is a non-zero probability that a suboptimal element $j \notin \mathcal{O}$ is maximal according to m^A.

Note that by symmetry of m^A and m^B, we can also use $m^A_{b^*}$, where b^* maximizes m^B. This means that we have access to two approximations instead of one. There are several ways these approximations can be combined, the simplest of which perhaps is to average them: $\max_i \mu_i \approx (m^B_{a^*} + m^A_{b^*})/2$. This improves the approximation by lowering the variance. Compared to the single estimator this is not an important advantage, since the variance of both m^A and m^B will normally be larger than that of the single estimator m, since both only use a subset of the samples.

3.4.3 An Upper Bound on the Size of the Underestimation

As with the single estimator, there exists no general non-trivial lower bound for the underestimation, since there exist non-trivial sets of random variables for which there is no underestimation. For instance, when the random variables are iid, no underestimation bias will occur. Additionally, we do not know of any general upper bounds in the literature. To get some idea of the worst case underestimation, we prove the following lemma.

Lemma 3.4. *Let $Y = \{Y_1, \ldots, Y_M\}$ be a set of random variables with expected values $\mu_1, \ldots, \mu_M$ and let $m^A = \{m^A_1, \ldots, m^A_M\}$ and $m^B = \{m^B_1, \ldots, m^B_M\}$ be two sets of unbiased estimators such that $E\{m^A_i\} = E\{m^B_i\} = \mu_i$ for all i. Let $j \in \mathcal{O}$ denote an optimal estimator and further assume that*

1. *all variances of all estimators are equal to σ, where $\sigma < \infty$,*
2. *all non-optimal estimators are iid with expected value $\mu_i < \mu_j$.*

The expected underestimation defined by $\mu_j - E\{m^B_i | i \in \mathcal{M}^A\}$ is then bounded by

$$\mu_j - E\{m_i^B | i \in \mathcal{M}^A\} < \frac{9}{4}\sigma\sqrt{M-1} \ .$$

The proof for Lemma 3.4 is given in Section 3.10.4. This lemma gives an upper bound for the magnitude of the underestimation that is a factor 9/4 larger than the bound we discussed for the overestimation of the single estimator. However, the additional factor is more likely due to the very rough

Table 3.1: The conditions for a bias to occur for the single and double estimator.

	condition for bias
single estimator	$\exists j \in \mathcal{O} : P(j \notin \mathcal{M}) > 0$
double estimator	$\exists j \notin \mathcal{O} : P(j \in \mathcal{M}^A) > 0$

approximations in the proof of the lemma than an actual worse bias for the double estimator. It is important to note that again we obtain a linear dependence on $\sigma\sqrt{M}$.

In the remainder of this chapter we will demonstrate that often the underestimation of the double estimator is smaller than the overestimation of the single estimator. Unfortunately, this does not hold in general and there even exist settings in which the single estimation experiences no bias, while the double estimator on average underestimates the maximal expected value.

3.5 Comparing the Single and Double Estimator

Comparing the conditions for a bias to occur for the single and the double estimator, one may get the impression that these methods are mirror images in a sense. In formula (3.10) it is stated that the single estimator overestimates when an optimal index that a non-zero probability of not being maximal. In the formula for the double estimator, given in (3.17), these conditions are reversed since it underestimates when there is a index that is not optimal that has a non-zero probability of being maximal. This is summarized in Table 3.1.

Although the conditions are related, there exist settings in which both, none, or only one of the estimators is biased. When the variables are iid with non-zero variance, only the condition of the single estimator holds. All the estimators are then optimal and the double estimator is therefore unbiased. An example of such an iid setting is the approximation of the maximum of the expected value of two normal dice, as we have seen before.

When there is a non-zero probability that a suboptimal estimator is maximal, the double estimator has a negative bias. If additionally all optimal estimators are always maximal, the single estimator is unbiased. An example includes estimating the maximal of a normal die and a die that always rolls six. Then, the single estimator always correctly gives six as its estimate for the maximal expected value. However, if we assume a random die is chosen when they both roll six, the double estimator can incorrectly pick the normal die with probability 1/12. Then, the second estimate, obtained by throwing the chosen die again, yields on average $3\frac{1}{2}$. Therefore, the expected estimate

by the double estimator in this case will be

$$\frac{11}{12}(6) + \frac{1}{12}\left(3\frac{1}{2}\right) = 6 - \frac{1}{12}\left(2\frac{1}{2}\right) \approx 5.79 \ .$$

The expected underestimation in this case is quite small and will be even smaller if we allow the choice of die to depend on more than a single throw. For instance, if we throw each die twice and we use the averages to determine which die to throw one final time to get our estimate there is only a probability of 1/72 that we pick the normal die. Then, the expected estimate by the double estimator already increases to $6 - (2\frac{1}{2})/72 \approx 5.97$.

A trivial setting in which both approaches are unbiased is when there is only one random variable. There are also many settings in which both approaches are biased. In the next sections, we will present some examples and intuitions about when to expect a bias.

3.6 A Comparison on Uniform Variables

The approximations resulting from the single and double estimator can both contain bias and will both become closer to the value $\max_i \mu_i$ as we obtain more samples. This raises the question when which approach works better. To shed some light on this issue, in the rest of this section we look at both methods in more depth in specific settings.

The rate of convergence is dependent on the distributions of the random variables. Usually convergence will be faster when the variances are smaller, which translates to less noise in the samples. Less variance can imply smaller probabilities that the optimal estimators are not maximal for the single estimator and smaller probabilities that the maximal estimators are not optimal for the double estimator. Similarly, for a given amount of variance, the approximations by the single estimator are usually better when the differences between the means of the estimators are large, again because then the probability that an optimal estimator is not maximal is usually smaller. The double estimator can experience two effects for large differences in the means. The larger the difference, the smaller the probability that the maximal index is suboptimal. This can result in less underestimation on average. However, when a suboptimal index is maximal the resulting underestimation will be larger. In other words, for the double estimator the probability of selecting a suboptimal estimator may decrease, but when it is selected the resulting approximation error is larger. We will illustrate these effects by examining an example in depth.

Assume we have two uniformly distributed variables Y_1 and Y_2 and we obtain a single sample for each of our estimators. This means that we assume we obtain one independent sample for each of the four estimators of the double estimator m_1^A, m_2^A, m_1^B and m_2^B as well as one for each of the two

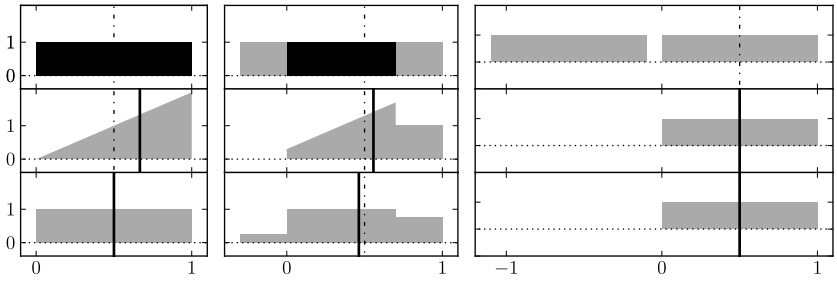

Figure 3.2: The first row shows three cases of PDFs of two uniformly distributed random variables. The parts where the PDFs overlap are shown in black. The second row shows the PDFs of the expected approximation by the single estimator. The third row shows the PDFs that result from the double estimator. In all cases, the dotted vertical line is the true maximum expected value $\max_i \mu_i = 1/2$, and the solid vertical lines indicate the expected values for each approximation of this value.

estimators m_1 and m_2 in the single estimator. In this example the double estimator uses twice the number of samples, but we nullify this advantage by only considering a single approximation $m^B_{a^*}$ instead of the average of $m^A_{b^*}$ and $m^B_{a^*}$ that was discussed at the end of the former section.[1]

We analyze the overestimation of the single estimator and the underestimation of the double estimator. The first row of Figure 3.2 shows the PDFs of two otherwise identical uniform random variables Y_1 and Y_2 with different separations between their expected values. In the left column, the two PDFs overlap fully. In the middle column there is some overlap and in the right column, the PDFs are separated completely, which implies that samples for one of the variables will always be larger than those for the other variable. The second and third row respectively show the PDFs for the approximations when using a single estimator and when using a double estimator. These PDFs were constructed analytically. The solid vertical lines in the plots show the corresponding expected approximation of the maximum expected value for the two methods.

The second plot in the left column shows that the expected approximation of the single estimator is an overestimation when the PDFs fully overlap. The intuitive explanation is that on average in three out of four cases at least one of the estimates will be larger than the expected value of 1/2, while in only one out of four cases both estimates will be smaller than 1/2. This indicates

[1]Note that in this case the double estimator only uses three of the four samples. For instance, if $a^* = 1$, we will only need m^B_1 and not m^B_2. All four samples are needed if both $m^B_{a^*}$ and $m^A_{b^*}$ are used, but the estimate by the double estimator will then be better because its variance will be lower.

that the maximum of the two values will be an overestimation more often than it will be an underestimation. The formal reason there is a positive bias is that there is a non-zero probability that one of the optimal estimators is not maximal. In fact, in this case the probability is close to one. The third plot in the left column shows the double estimator is unbiased and on average returns the true value. This happens because both variables are optimal and if there exist no suboptimal values, the probability that a suboptimal value is maximal is of course equal to zero.

The second plot in the central column shows that when the overlap between the distributions becomes less, the expected overestimation by the single estimator decreases. This results from a decrease in the probability that the optimal estimator is not maximal. The third plot in the central column shows that the double estimator now gives an underestimation of the true maximal expected value. This happens because the probability of selecting a suboptimal estimator is no longer zero, since in contrast with the overlapping PDFs in the left column now there exists a suboptimal estimator. The step-wise shape of the PDF is the result of a weighted addition of the two PDFs.

The right column shows that when the PDFs do not overlap, both approaches yield the unbiased, correct result. In this scenario, both the single estimator and the double estimator exploit the fact that the optimal estimator is always maximal.

It is interesting to note that there is no underestimation by the double estimator when the PDFs fully overlap. Conversely, the single estimator approach has a large positive bias in this case.

In a simple case as this, we can analytically determine the expected over- and underestimations for various amounts of separation between the expected values of the two distributions. Let $Y_1 \sim u(x,0,1)$ and $Y_2 \sim u(x,-D,1-D)$, with $D \in [0,1]$, be two uniformly distributed random variables. Note that $\max_i \mu_i = E\{Y_1\} = \frac{1}{2}$ for all $D \in [0,1]$. We can then use (3.8) and (3.15) to ob-

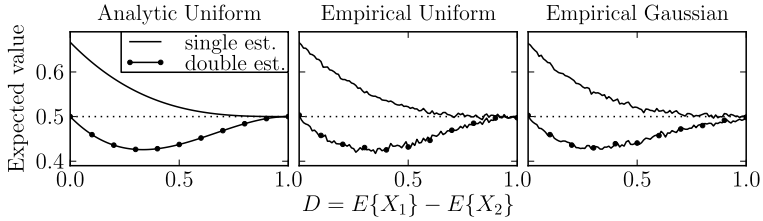

Figure 3.3: Approximations by the single and double estimator to the value of $\max_i E\{Y_1, Y_2\} = 1/2$. The left plot shows analytical expected values where $Y_1 \sim u(x, 0, 1)$ and $Y_2 \sim u(x, -D, 1-D)$, for $0 \leq D \leq 1$, the center plot shows empirical results for these variables. The right plot shows empirical results for Gaussian PDFs with the same mean and variance: $Y_1 \sim g(x, 0.5, \frac{1}{12})$ and $Y_2 \sim g(x, 0.5-D, \frac{1}{12})$. The center and right plots are averaged over 10,000 sets of samples.

tain

$$
\begin{aligned}
E\{\max_i m_i\} &= \int_{-\infty}^{\infty} x \left(\frac{d}{dx} U_1(x) U_2(x) \right) dx \\
&= \int_0^{1-D} x \left(\frac{d}{dx} x(x+D) \right) dx + \int_{1-D}^{1} x \left(\frac{d}{dx} x \right) dx \\
&= \frac{1}{2} + \frac{(1-D)^3}{6} \quad , \\
\sum_{i=1}^{2} P(i = a^*) E\{m_i^B\} &= \int_{-\infty}^{\infty} u_1(x) U_2(x) E\{Y_1\} + U_1(x) u_2(x) E\{Y_2\} \, dx \\
&= \int_{-\infty}^{\infty} u_1(x) U_2(x) \left(\frac{1}{2} \right) + U_1(x) u_2(x) \left(\frac{1}{2} - D \right) dx \\
&= \int_0^{1-D} (x+D) \left(\frac{1}{2} \right) + x \left(\frac{1}{2} - D \right) dx + \int_{1-D}^{1} \left(\frac{1}{2} \right) dx \\
&= \frac{1}{2} - \frac{D(1-D)^2}{2} \quad .
\end{aligned}
$$

These values are plotted in the leftmost plot in Figure 3.3 for separations D of 0 (full overlap) to 1 (no overlap). The overestimation by the single estimator is clearly visible when there is little separation between the expected values of the random variables. The larger the separation, the lower this overestimation. The magnitude of the underestimation related to the double estimator approach first grows, because selecting the suboptimal estimator becomes more harmful. For larger D this underestimation shrinks again, since selecting the suboptimal estimator becomes less likely. For $D \geq 1$, the PDFs no longer overlap and the single estimator and the double estimator are

both unbiased.

The worst biases can be found analytically for this simple case: these occur at $D = 0$ for the single estimator, for an expected overestimation of $\frac{1}{6}$ and at $D = \frac{1}{3}$ for the double estimator for an expected underestimation of $-\frac{2}{27}$. The upper bounds we found earlier in this case reduce to $\sigma\sqrt{M-1} = 1/\sqrt{12} = (1/6)\sqrt{3}$ for the single estimator and $-(9/4)\sigma\sqrt{M-1} = -(3/8)\sqrt{3}$ for the double estimator. We see that in this case the general bound for the single estimator is fairly tight, at $\sqrt{3}$ times the actual bound for the uniform distributions. The bound for the double estimator is much looser, at more than 8 times the actual worst underestimation. This confirms our suspicion that the general bound for the double estimator is quite loose.

Also shown in the figure are empirical results for both the uniform case and for two Gaussian variables with the same means and variances. Besides showing that our analysis so far is correct, the fact that the center and right plot are almost identical shows that the estimates for Gaussian variables are very similar to the estimates for corresponding uniform variables. The main difference is that for a separation of $D \geq 1$, the Gaussian PDFs will still slightly overlap, resulting in a slight over- and underestimation.

3.7 The Effect of More Samples

The former section assumes fixed uniform PDFs and looks at different separations of the mean. In practice, for a given problem the difference between the expected values of the random variables is fixed. This does not mean that the approximation biases of the single and double estimator are stable, since the variance of each estimator decreases as more samples are observed. How this affects the biases of the single and double estimator is important, since this tells us something about the data efficiency in general of both methods.

A decreasing variance has two relevant effects. First, estimators with different means will have PDFs with less and less overlap and thus the probability that the maximal estimator is also optimal increases. Second, for all estimators the expected deviation from their true mean becomes less. The effects improve the estimation for both the single and the double estimator but it is not immediately clear how large these improvements are.

Considering the worst case settings we discussed in Sections 3.3 and 3.4, we can simply state that the upper bound on the bias decreases with $1/\sqrt{N}$. This is the results of the central limit theorem, that implies that the random variable that represents an estimator that is the average of N samples for a random variable with variance σ^2 will become more similar to a Gaussian with variance σ^2/N for increasing N. We will not redo the analysis in full, but the upper bounds for the over- and underestimations can then be given as $\sigma\sqrt{(M-1)/N}$ and $(9/4)\sigma\sqrt{(M-1)/N}$, respectively.

Suppose we increase the used number of samples in the case of two uniform variables, as considered in the previous section. Then, the resulting PDFs for the estimators are no longer uniformly distributed and due to the central limit theorem they become more and more similar to Gaussian distributions. Given that a uniform distribution $u(x,y,z)$ has variance $\frac{(z-y)^2}{12}$, we know that an estimator for a variable $u(x,-\frac{1}{2},\frac{1}{2})$ will be approximately distributed according to a Gaussian $g_N(x) = g\left(x,0,\frac{1}{12N}\right)$ after N samples have been obtained for that estimator. Similarly, the PDF for an estimator for a variable with PDF $u(x,-\frac{1}{2}-D,\frac{1}{2}-D)$ will be approximately $g_N(x+D) = g\left(x,-D,\frac{1}{12N}\right)$.

Assume all estimators have received N samples for the corresponding variable and that N is sufficiently large that the PDFs of the estimators become approximately Gaussian in shape. Again, to avoid unfairly advantaging the double estimator approach, we only look at a single approximation due to $m^B_{a^*}$ and ignore $m^A_{b^*}$. Figure 3.3 indicates that the uniform expressions we derived earlier give a fairly good approximation of these Gaussian values. So instead of calculating the expected over- and underestimation directly for the Gaussians, we give the uniform approximations. The corresponding uniform PDFs are defined as $u_N = u\left(x,\frac{-1}{2\sqrt{N}},\frac{1}{2\sqrt{N}}\right)$ and $u_N(x+D) = u\left(x,\frac{-1}{2\sqrt{N}}-D,\frac{1}{2\sqrt{N}}-D\right)$. It is easily verified that the variance of these distributions is equal to $\frac{1}{12N}$, as required. Note that $\max_i \mu_i = E\{u_{0,N}\} = 0$. The oveestimation bias due to the single estimator can then be approximated by

$$\begin{aligned} E\{\max_i Y_i\} &= \int_{-\infty}^{\infty} x\left(\frac{d}{dx}G_N(x)G_N(x+D)\right)dx \\ &\approx \int_{-\infty}^{\infty} x\left(\frac{d}{dx}U_N(x)U_N(x+D)\right)dx \\ &= \frac{\left(\frac{1}{\sqrt{N}}-D\right)^3}{\frac{6}{N}}\ , \end{aligned} \tag{3.18}$$

Similarly, the underestimation bias of the double estimator can be approximated with

$$\begin{aligned} \sum_{i=1}^{2} P(i=a^*)E\{m^B_i\} &= \int_{-\infty}^{\infty} g_N(x)G_N(x+D)\mu_1 + G_N(x)g_N(x+D)\mu_2\,dx \\ &= \int_{-\infty}^{\infty} -DG_N(x)g_N(x+D)\,dx \\ &\approx \int_{-\infty}^{\infty} -DU_N(x)u_N(x+D)\,dx \\ &= -\frac{D\left(\frac{1}{\sqrt{N}}-D\right)^2}{\frac{2}{N}}\ . \end{aligned} \tag{3.19}$$

Uniform approximation							
	single estimator				double estimator		
D	0	0.01	0.03	0.1	0.01	0.03	0.1
$N = 1$	0.167	0.162	0.152	0.121	−0.005	−0.014	−0.040
$N = 10$	0.053	0.048	0.039	0.017	−0.005	−0.012	−0.023
$N = 10^2$	0.017	0.012	0.006	0	−0.004	−0.007	0
$N = 10^3$	0.005	0.002	0.000	0	−0.002	−0.000	0
$N = 10^4$	0.002	0	0	0	0	0	0
Empirical results							
	single estimator				double estimator		
D	0	0.01	0.03	0.1	0.01	0.03	0.1
$N = 1$	0.173	0.164	0.150	0.122	−0.007	−0.010	−0.041
$N = 10$	0.053	0.047	0.040	0.017	−0.004	−0.011	−0.022
$N = 10^2$	0.016	0.012	0.006	0.001	−0.004	−0.007	−0.001
$N = 10^3$	0.005	0.002	0.000	−0.000	−0.002	−0.000	0.000
$N = 10^4$	0.002	0.000	0.000	0.000	−0.000	0.000	−0.000

Table 3.2: The under- and overestimations according to the single and double estimator for different separations between the means of two uniform random variables after finding N samples for each variable. The upper half shows the results obtained by the approximations in (3.18) and (3.19). The lower half shows the average over- and underestimations obtained by using each method to approximate the maximal expected value using actual random data. The results in the lower half were obtained by averaging over 10,000 sample sets for each combination of D and N.

Note that these expressions only hold for $D \leq \frac{1}{\sqrt{N}}$ since for larger D both approaches will be unbiased because the distributions then no longer overlap. Although it is an approximation of the actual bias, it is interesting to have these expressions for higher numbers of samples for each estimator. For $N = 1$, these results reduce to those given in the former section. For general $N \in \mathbb{N}$, $N \geq 1$, the worst approximation for the single estimator continues to occur at $D = 0$, for an overestimation of $\frac{1}{6\sqrt{N}}$. For the double estimator, the worst separation depends on N and occurs at $D = \frac{1}{3\sqrt{N}}$ for an underestimation of $-\frac{2}{27\sqrt{N}}$.

Table 3.2 gives some values for the under- and overestimations when there are two uniform distributions with equal variance. We see that for most shown combinations of differences in the mean D and numbers of samples N the bias of the double estimator is closer to zero, while for some combinations the bias of the single estimator is closer to zero. However, in general the bias of the double estimator often is smaller than that of the single estimator,

sometimes by a large margin.

The bottom half of Table 3.2 contains an empirical verification of the approximation we used. For this, each method was used 10,000 times to approximate the maximal expected value for each combination of D and N. These 10,000 results were then averaged to get the results in the table. Results for the double estimator when $D = 0$ are not shown, since then the double estimator has no bias.

Recall that the average over multiple samples of a uniform distribution will tend towards a Gaussian and we have used (3.18) and (3.19) to approximate this Gaussian with a uniform PDF with the same variance. The table shows that this approximation is quite accurate. Differences between the upper and lower half of the table are mostly due to the variance in the empirical results that is not present in the approximations. This explains why the differences are the largest for $N = 1$, although then formulas (3.18) and (3.19) are exact. For larger N, the variance is smaller and this has a larger effect than any approximation error that is introduced by using the uniform approximation. For any single sample, the variance will often obscure the biases, which is probably why these biases are often not noticed.

It is useful that the uniform approximations from (3.18) and (3.19) apparently are good approximations for Gaussian variable, since the distribution of any estimator will tend to a Gaussian if more samples are obtained. Undoubtedly better approximations for any probability distribution can be obtained, but an analysis of the accuracy of the present approximations and possible improvements lie beyond the scope of this chapter. Rather, we attempt to give a general idea of the size of the biases and in which settings each bias can be expected to occur.

3.8 The Effect of More Variables

Sections 3.6 and 3.7 specifically assume two random variables. In this section we will find fairly general approximations for the worst case bias of the single estimator and double estimator when there are more than two random variables.

The worst expected overestimation for the single estimator occurs when the variables are iid. For M iid uniform PDFs with parameters y and z, the

expected maximal sample is

$$\begin{aligned} E\{\max_i m_i\} &= \int_{-\infty}^{\infty} x \left(\frac{d}{dx} \prod_i^M U(x,y,z) \right) dx \\ &= \int_y^z x \left(\frac{d}{dx} \left(\frac{x-y}{z-y} \right)^M \right) dx \\ &= \frac{Mz+y}{M+1} \quad , \end{aligned} \tag{3.20}$$

which is an overestimation of

$$E\{\max_i m_i\} - \max_i E\{m_i\} = \frac{Mz+y}{M+1} - \frac{z+y}{2} = \frac{M-1}{M+1}\frac{z-y}{2} \quad .$$

The iid uniform distributions have a standard deviation equal to $\sigma = (z - y)/\sqrt{12}$. Therefore, the overestimation for multiple uniform distributions with identical means and standard deviations of σ can be rewritten as

$$E\{\max_i m_i\} - \max_i E\{m_i\} = \frac{M-1}{M+1}\sigma\sqrt{3} \quad .$$

This precise overestimation for the uniform is quite a lot smaller than the upper bound of $\sigma\sqrt{M-1}$, especially for large M.

Similar to the former section, we can use the uniform case as an approximation for the overestimation for M Gaussian distributions. Due to the central limit theorem, we can assume that any estimator with N samples from a random variable with variance σ^2 will tend to a Gaussian distribution with variance $\frac{\sigma^2}{N}$ for large enough N. Further, from the last section we know that the overestimation for the Gaussian distribution is very similar to the overestimation for the uniform distribution. Therefore, we hypothesize that the maximal overestimation for M iid variables with any distribution will be approximately

$$E\{\max_i m_i\} - \max_i E\{m_i\} \approx \frac{M-1}{M+1}\sigma\sqrt{\frac{3}{N}} \quad , \tag{3.21}$$

where N is the number of samples obtained for each of the variables. We will not test the accuracy of this approximation, but preliminary tests show that it gives a good general idea of the size of the overestimation.

The double estimator in the iid case gives no underestimation since then $\mu_j = \max_i \mu_i$, for all j and therefore

$$\sum_{j=1}^{M} P(j = a^*) E\{m_j^B\} = \sum_{j=1}^{M} \frac{1}{M}\mu_j = \sum_{j=1}^{M} \frac{1}{M} \max_i \mu_i = \max_i \mu_i \quad .$$

The worst underestimation for multiple PDFs which only differ in their means occurs when there is a single optimal variable and all other variables have

	single estimator		**double estimator**	
	U	G	U	G
$M = 2$	0.577σ	0.569σ	-0.257σ	-0.241σ
$M = 3$	0.866σ	0.854σ	-0.402σ	-0.375σ
$M = 4$	1.039σ	1.030σ	-0.494σ	-0.450σ
$M = 10$	1.417σ	1.546σ	-0.702σ	-0.736σ
$M = 100$	1.698σ	2.505σ	-0.849σ	-1.322σ
$M = 1000$	1.728σ	3.243σ	-0.864σ	-1.594σ
$M \to \infty$	$\sqrt{3}\sigma$	∞	$-\sqrt{3}\sigma/2$	$-\infty$

Table 3.3: The maximal under- and overestimations according to the single and double estimator for different numbers of uniform random variables after obtaining 1 sample for each variable. Analytical results for M uniform variables with variance σ^2 (columns with label U) and empirical results for M Gaussian variables with variance σ^2 (columns with label G) are shown. The positive and negative infinity in the last row only hold when $\sigma > 0$. If $\sigma = 0$, all values are equal to zero.

equal lower expected values. The reason is that if the expected values of the other variables are at a suitable distance from the optimal expected value, there will be a reasonable high probability that they are maximal, while they still ensure a relatively large underestimation. For instance, for M uniform PDFs with a variance of σ^2, the underestimation can then be calculated with use of (3.15) to get

$$E\{m_j^B | j \in \mathcal{M}^A\} - \max_i E\{m_i\} = -D\left(\frac{1}{M}\left(\frac{\sqrt{12}D}{\sigma}\right)^M - \frac{\sqrt{12}D}{\sigma} + \frac{M-1}{M}\right) .$$

It is non-trivial to find a closed form expression for the distance D that minimizes this expression and therefore gives the largest underestimation. Like the single estimator, the underestimation will decrease linearly as a function of $\sqrt{N}$ as more samples are obtained. Furthermore, for specific values of M, it is easy to calculated the worst underestimation.

In Table 3.3, we compare the worst over- and underestimations for the single and double estimator for different numbers of uniform PDFs and $N = 1$. Both methods were also used to estimate the maximal expected value for M Gaussian variables. The averages of these estimates over 10,000 trials are shown in the Table.

Table 3.3 shows that the maximal bias for the single estimator is about twice that of the double estimator. Of course, in many cases the variables will not be distributed in a worst case manner and the actual over- and underestimations will be less severe. However, if there are sufficiently many variables, the bias of each approach can be significant.

Finally, we note that although the bias for the Gaussian variables continues to increase with larger M, the bias is far smaller than the upper bounds we have formulated before. For instance, for $M = 1000$, the upper bound we formulated for the single estimator is approximately 31.6σ, while that of the double estimator is approximately -71.1σ. For smaller values of M, the bounds are sharper. More specifically, the uniform bias grows with a factor of $(M-1)/(M+1)$ rather than $\sqrt{M}$ and the Gaussian bound seems to grow with a logarithmic rate.

3.9 Conclusion

In this section we summarize our results and point out directions for future research.

3.9.1 Summary and Discussion

We have shown that there exist at least two approaches to determine an estimate for the maximal expected value of a set of random variables. We have proven that both approaches are biased. The single estimators approach, which simply estimates the maximal expected value by taking the maximal value from a set of unbiased estimators, on average overestimates the maximal expected value. Conversely, the double estimator uses the set of estimators only to determine the likely identity of the maximal random variable. Then, the value of an independent unbiased estimate for this element is used as an estimate for the maximal expected value. This approach can underestimate this value, if there is a chance we have selected the wrong element in the first step.

We have formulated worst-case upper bounds for the single and double estimator and we have shown that for both approaches these upper bounds increase at most linearly in $\sigma\sqrt{M}$, where σ is the variance of the variables and M is the number of variables. If the shape of the distributions are known, it becomes possible to calculate the bias more accurately and we have done so for the uniform distribution. This way we could show in which manner the bias depends on the mean of the distributions, the number of samples and the number of random variables. Additionally, we have shown that the biases as calculated when assuming uniform distributions are a fairly good estimate for the biases when the random variables are distributed according to a Gaussian distribution in some cases. When the number of variables increases, this approximation because less accurate. For any given set of distributions, one can obtain better estimates of the expected bias by numerical integration, as for instance proposed by Ross (2010) in a setting with Gaussian variables. This is not always particularly useful, since the distributions in general will not be known.

Our analysis shows that in many cases the double estimator has a lower bias than the single estimator. Additionally, both approaches obtain their worst behavior in somewhat different settings. The single estimator has the largest overestimation in settings with many iid variables. The double estimator has the largest underestimation when there is a single optimal variable and many suboptimal variables with values that are neither too close nor too far from the optimal variable.

Since the direction of the bias for the approaches is opposite, one could envision averaging the results of both to get better estimates. However, because the approaches are not fully mirror images of each other, such an approach would either overestimate or underestimate, depending on the precise distributions of the random variables.

We have seen that the bias of the double estimator is often smaller for small sample sizes. For instance, see the first row of Table 3.2 for an example with two uniform or Gaussian variables. Therefore, in the next chapter we will investigate whether a form of Q-learning that uses a double estimator approach is also has a lower bias than normal Q-learning in Chapter 4.

The overestimation bias of the single estimator is similar to the overfitting that can occur when fitting a model to data. When one optimizes the parameters of a model and then uses the prediction of this same model as an estimator of the performance of the model, the model will be overly optimistic. The bias here is essentially the same as for the single estimator. The double estimator is then similar to a 2-fold cross validator. This indicates a possible direction of improvement for the double estimator approach to a N-fold cross validator. This is discussed at the end of the future work section below.

3.9.2 Future Work

Interesting future work can include trying to find an unbiased estimator for the maximal expected value of a set. Probably, this goal is not reachable in general, since we assume the distributions of the random variables are not known. Sometimes, rather than trying to determine the maximal expected value directly one could try to estimate the distributions. For instance, if the samples do not seem to contradict this, we can assume normal distributions for the random variables. Then, we can find unbiased estimates for the means and variances of these distributions. Using the obtained unbiased estimates for the actual distributions, we can find the expected maximal value by numerical integration. Then we can construct an estimate of the overestimation and deduct this from the found maximal value. Naturally, the obtained result will not be unbiased in general since we made the explicit choice for the shape of the distribution. The same holds if we admit different choices of distribution shapes. Additionally, one would want to use different sets of samples to determine the distributions and to determine the maximal estimator, since

the estimate will gain bias if these values are not independent. However, for large enough numbers of samples this may give accurate results.

A limitation of such an approach is that in many applications the number of samples may be too small to obtain a reliable estimation of the distribution. For instance, if a reinforcement learning agent that uses Q-learning visits a state, it must make an estimate for the maximal action value in that state based only on a single sample for each action. One cannot even reuse these samples when the state is visited again, since the action values may have changed based on later information. It is important to stress this point: the action values in a state should be interpreted as samples from an unknown distribution, where this distribution depends on the whole experience of the agent, on the learning algorithm and on the MDP. Even if the MDP and the learning algorithm are assumed to be stationary, at the next visit to a state the experience of the agent will be different and therefore the unknown distributions of the action values may have changed. In general it is even impossible to obtain more unbiased samples if we assume we have more than one learning agent, since stochastic state transitions may make it impossible to reach the same state with the same experience.

Another interesting topic for future work is the extension of the double estimator approach. The double estimator is one way to avoid the overestimation of the single estimator, but perhaps it is better not to use more than two sets of samples. For instance, if we have N samples for each variable, we can average $N-1$ of these samples to obtain an estimator that we can use to obtain the maximal element. In the double estimator approach as we described it in this chapter, only $N/2$ samples are used for this. This means that the probability decreases that we choose a suboptimal estimator and when we do choose a suboptimal estimator the probability that its value is fairly close to the optimal value is larger. Of course, we then have only a single sample left to get an unbiased estimate for the variable that we select. Therefore, its variance will be larger. However, we can then repeat the procedure for all N samples to obtain N different estimates. These can then be combined to obtain a sample with the same low variance as the estimate by the single estimator. The estimate for the N-fold estimator will still have a negative bias, but the magnitude of this bias will be lower than the underestimation by the double estimator. However, this approach does assume that multiple samples are available for each variable. In reinforcement learning, this is normally not the case as most methods only store a single action value function. We will discuss the application of the double estimator to Q-learning in the next chapter.

3.10 Proofs

This section contains the proofs for the lemmas contained in this chapter.

3.10.1 Proof for Lemma 3.1

Proof. Assume

$$P(j \in \mathcal{M}) < 1 \ , \tag{3.22}$$

for some $j \in \mathcal{O}$. If m_j is the only optimal estimator, we immediately reach our desired conclusion that $\forall j \in \mathcal{O} : P(j \in \mathcal{M}) < 1$. Therefore, we will focus on the case where m_j is one of multiple optimal estimators.

Assumption (3.22) implies that there is at least one estimator that yields a higher estimate than m_j for some sample sets. In other words, at least one estimator has a non-zero probability of yielding a higher estimate than m_j:

$$\exists i : P(m_i > m_j) > 0 \ . \tag{3.23}$$

Then, since m_j is optimal, m_j must also sometimes yield higher estimates than this estimator. Otherwise, the expected value of m_i is larger than that of m_j, which contradicts the optimality of m_j. Therefore

$$\forall j \in \mathcal{O} \forall i : \big(P(m_i > m_j) > 0 \ \rightarrow \ P(m_i < m_j) > 0\big) \ . \tag{3.24}$$

Furthermore, in general no estimator can yield estimates that are larger than the maximal expected value for all sample sets. Otherwise, this estimator would have a larger expected value than the maximal expected value, which contradicts our general assumption that the individual estimators are unbiased. This implies that

$$\forall j \in \mathcal{O} \forall k : P(m_k \leq m_j) > 0 \ . \tag{3.25}$$

We will now consider two different cases.

First, in addition to assumption (3.22), assume that m_j never overestimates the maximal expected value:

$$P(m_j \leq \max_i \mu_i) = 1 \ . \tag{3.26}$$

Since j is optimal and unbiased, this implies that m_j also never underestimates the maximal expected value and therefore $P(m_j = \max_i \mu_i) = 1$. In other words, assumption (3.26) implies that m_j is a constant estimator that yields the same estimate for any set of samples. In this case, equation (3.23) can be rewritten as

$$\exists i : P(m_i > \max_i \mu_i) > 0 \ . \tag{3.27}$$

Now, assume m_i is some specific estimator for which (3.27) holds and consider an arbitrary other index $k \neq i$. The probability that k is not maximal is at

least as high as the probability that m_i gives a larger estimate than m_k. This implies

$$P(m_k \notin \mathcal{M}) \geq P(m_i > m_k) \geq \left(P(m_i > \max_i \mu_i) P(\max_i \mu_i \geq m_k) \right) \ , \qquad (3.28)$$

where the second inequality uses our general assumption that the estimators are independent. By equation (3.25) and our assumption that (3.27) holds for m_i, the elements of the product on the right hand side or inequality (3.28) are strictly positive. Since k was chosen arbitrarily, this then holds for all $k \neq i$. Finally, by equation (3.24) we know that $P(m_i \notin \mathcal{M}) > 0$. Together, this implies that assumptions (3.22) and (3.26) lead to the conclusion that

$$\forall k : P(m_k \notin \mathcal{M}) > 0 \ .$$

Since this holds for all k, it naturally holds for all optimal estimators and therefore assumption (3.26) that m_j never overestimates the maximal expected values leads to the desired result.

Next, we assume the negation of assumption (3.26). In other words, we assume that m_j sometimes overestimates the maximal expected value:

$$P(m_j > \max_i \mu_i) > 0 \ . \qquad (3.29)$$

This immediately implies that m_j must also sometimes underestimate the maximal expected value, because otherwise m_j would not be unbiased:

$$P(m_j < \max_i \mu_i) > 0 \ . \qquad (3.30)$$

For any $k \neq j$ we have

$$P(m_k \notin \mathcal{M}) \geq P(m_j > m_k) \geq \left(P(m_j > \max_i \mu_i) P(\max_i \mu_i \geq m_k) \right) \ ,$$

where the second inequality again holds by independence of the estimators. By equations (3.29) and (3.25), both elements in the product in the last equation must be strictly positive. Add to this the assumption (3.25) itself and we obtain

$$\forall k : P(m_k \notin \mathcal{M}) > 0 \ .$$

We have now shown that assumption (3.25) leads to the conclusion that $P(m_k \notin \mathcal{M}) > 0$ for all k both when we assume (3.26) and when we assume its negation, (3.29). By the law of the excluded middle (Aristotle, 350BC), this concludes our proof and shows that if $P(j \notin \mathcal{M}) > 0$ for any $j \in \mathcal{O}$, then $P(k \notin \mathcal{M}) > 0$, for all $k \in \mathcal{O}$. □

3.10.2 Proof for Lemma 3.2

Proof. Assume $j \in \mathcal{O}$, such that by Definition 3.1 m_j is an optimal estimator. Then

$$\begin{aligned} E\left\{\max_i m_i\right\} &= P(j \in \mathcal{M}) E\left\{\max_i m_i \middle| j \in \mathcal{M}\right\} + P(j \notin \mathcal{M}) E\left\{\max_i m_i \middle| j \notin \mathcal{M}\right\} \\ &= P(j \in \mathcal{M}) E\left\{m_j \middle| j \in \mathcal{M}\right\} + P(j \notin \mathcal{M}) E\left\{\max_i m_i \middle| j \notin \mathcal{M}\right\} \\ &\geq P(j \in \mathcal{M}) E\left\{m_j \middle| j \in \mathcal{M}\right\} + P(j \notin \mathcal{M}) E\left\{m_j \middle| j \notin \mathcal{M}\right\} \\ &= E\left\{m_j\right\} = \mu_j = \max_i \mu_i \ . \end{aligned}$$

By definition of $\mathcal{M}$ we have $E\{\max_i m_i | j \notin \mathcal{M}\} > E\{m_j | j \notin \mathcal{M}\}$, for any j. Therefore, the inequality is strict if and only if $P(j \notin \mathcal{M}) > 0$, for some $j \in \mathcal{O}$. By Lemma 3.1, we know that if we find one $j \in \mathcal{O}$ such that $P(j \notin \mathcal{M}) > 0$ then this holds for all $j \in \mathcal{O}$. If we can not find one such optimal estimator, then we do not know if the inequality in is strict and therefore in general we write $E\{\max_i m_i\} \geq \max_i \mu_i$. □

3.10.3 Proof for Lemma 3.3

Proof. Assume a^* is optimal, such that $a^* \in \mathcal{O}$. Then, because $m^B_{a^*}$ is an unbiased estimator for μ_{a^*} and by definition of $\mathcal{O}$

$$E\left\{m^B_{a^*} \middle| a^* \in \mathcal{O}\right\} = E\left\{\mu_{a^*} \middle| a^* \in \mathcal{O}\right\} = \max_i \mu_i \ .$$

Now assume a^* is not optimal, such that $a^* \notin \mathcal{O}$. Then

$$E\left\{m^B_{a^*} \middle| a^* \notin \mathcal{O}\right\} = E\left\{\mu_{a^*} \middle| a^* \notin \mathcal{O}\right\} < \max_i \mu_i \ .$$

Naturally, one of these cases must hold, so in general

$$E\left\{m^B_{a^*}\right\} \leq \max_i \mu_i \ .$$

The inequality is strict when there is a non-zero probability that a^* is not optimal. This can occur when there is a non-zero probability that an estimator that is not optimal yields a maximal estimate, i.e.

$$\left(\exists j \notin \mathcal{O} : P(j \in \mathcal{M}^A)\right) \quad \rightarrow \quad \left(E\left\{m^B_{a^*}\right\} < \max_i \mu_i\right) \ .$$

If $E\{m^B_j\}$ is lower than $\max_i \mu_i$ for some j, this immediately implies that j is not optimal. Therefore, if the maximal element a^* has a lower expected value than the maximal expected value, then

$$E\left\{m^B_{a^*}\right\} < \max_i \mu_i \quad \leftrightarrow \quad \exists j \notin \mathcal{O} : P(j \in \mathcal{M}^A) \ .$$

□

3.10.4 Proof for Lemma 3.4

In a worst case setting, there can only be one optimal estimator, since if there are multiple optimal estimators, the probability of selecting an optimal estimator must decrease if we remove one. The probability of selecting a non-optimal estimator is equal to the probability that the maximum of all non-optimal estimators yields a larger estimate than the optimal estimator. In other words, if $j \in \mathcal{O}$ is the only optimal estimator, then

$$P(a^* \in \mathcal{O}) = P(j = a^*) = P(m_j > \max_{i \neq j} m_i) \ .$$

We define a new estimator

$$\hat{m}(X) = \max_{i \neq j} m_i(X)$$

In other words, the value of this estimator is the maximum value of all non-optimal estimators. Since there are M estimators in total and we assume that all $M-1$ non-optimal estimators are iid with mean μ_i and variance σ, we know from inequality (3.12) that the expected value of this new estimator is bounded by

$$\hat{\mu} \leq \mu_i + \sigma\sqrt{M-2} \ .$$

Furthermore, it can be shown that the variance $\hat{\sigma}^2$ of this estimator is bounded by $\hat{\sigma^2} < (M-1)\sigma^2$ (Papadatos, 1995).

Assume now for a moment that $\hat{\mu}$ is smaller than μ_j. Without loss of generality, we can then assume that the difference between $\hat{\mu}$ and μ_j is given by

$$\mu_j - \hat{\mu} = k\hat{\sigma} < k\sigma\sqrt{M-1} \ .$$

There is now a trade-off. When k increases, the probability decreases that a suboptimal estimator is maximal, but the underestimation when this happens increases. We will determine the worst case value of k and quantify an upper bound for the resulting underestimation.

We will use Chebyshev's inequality, that states that

$$P(X \geq \mu + k\sigma) \leq \frac{1}{1+k^2} \ ,$$

for any random variable X with expected value μ and variance σ. Specifically, we will use the equivalent reversed inequality:

$$P(X \leq \mu + k\sigma) \geq \frac{k^2}{1+k^2} \ ,$$

Then, the probability that $\hat{m}$ is smaller than or equal to m_j is at least

$$\begin{aligned} P(\hat{m} \leq m_j) &\geq P(\hat{m} \leq \hat{\mu} + k\hat{\sigma}/2)P(m_j \geq \mu_j - k\hat{\sigma}/2) \\ &> P(\hat{m} \leq \hat{\mu} + k\hat{\sigma}/2)P(m_j \geq \mu_j - k\sigma\sqrt{M-1}/2) \ . \end{aligned}$$

In other words, the probability that $\hat{m}$ is smaller than m_j is larger than the probability that both estimate do not deviate past some given value in between the two expected values. We can then apply Chebyshev's inequality twice to obtain

$$P(\hat{m} \leq m_j) \geq \left(\frac{k^2}{4+k^2}\right)\left(\frac{k^2(M-1)}{4+k^2(M-1)}\right) \geq \left(\frac{k^2}{4+k^2}\right)^2 \ ,$$

where the second inequality holds since $M \geq 1$. The probability that $\hat{m}$ is larger than m_j is then bounded by

$$P(\hat{m} > m_j) < 1 - \left(\frac{k^2}{4+k^2}\right)^2 \ . \tag{3.31}$$

If the maximal non-optimal estimate is larger than the optimal estimate, the underestimation is bounded by

$$\mu_j - \mu_i \leq \mu_j - \hat{\mu} + \sigma\sqrt{M-2} < k\sigma\sqrt{M-1} + \sigma\sqrt{M-2} < (k+1)\sigma\sqrt{M-1} \ .$$

The probability that this happens in bounded in (3.31), giving an upper bound for the expected underestimation of

$$\mu_j - E\{m_{a^*}\} < \left(1 - \left(\frac{k^2}{4+k^2}\right)^2\right)(k+1)\sigma\sqrt{M-1} \ .$$

If we maximize this for k we get

$$\max_k \left(1 - \left(\frac{k^2}{4+k^2}\right)^2\right)(k+1) = \frac{9}{4} \ ,$$

and therefore

$$\mu_j - E\{m_{a^*}\} < \frac{9}{4}\sigma\sqrt{M-1} \ .$$

For the former bound we assumed $\hat{\mu}$ is smaller than μ_j. If we assume that $\hat{\mu}$ is larger than μ_j we obtain

$$\mu_j - \mu_i \leq \mu_j - \hat{\mu} + \sigma\sqrt{M-2} \leq \sigma\sqrt{M-2} \ .$$

Since this bound is tighter than when $\hat{\mu}$ is smaller than μ_j, we can adopt the looser bound as a general result and our proof is concluded.

CHAPTER 4

THE OVERESTIMATION OF Q-LEARNING

In Chapter 2, we discussed methods to solve the prediction problem of finding the policy action value function Q^π and the control problem of finding the optimal action value function Q^*. The definitions of V^* and Q^* are given in equations (2.5) and (2.8) and are repeated here in expectancy notation for clarity:

$$V^*(s) = \max_a E\left\{r_t + \gamma V^*(s_{t+1}) | s = s_t, a = a_t\right\} \quad , \tag{4.1}$$

$$Q^*(s,a) = E\left\{r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') | s = s_t, a = a_t\right\} \quad . \tag{4.2}$$

The equation for the optimal action value function can alternatively be written in operator notation as

$$Q^* = \mathcal{T}^* Q^* \quad , \tag{4.3}$$

where the operator $\mathcal{T}^* : \mathbb{R}^{S \times A} \to \mathbb{R}^{S \times A}$ is defined as in (2.21), which we also repeat here:

$$\forall s \in S, a \in A(s): \quad (\mathcal{T}^* Q)(s,a) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma \max_{a'} Q(s', a')\right) \quad . \tag{4.4}$$

For any starting value function, repeated application of the operator $\mathcal{T}^*$ will bring the value of the value function closer to Q^*, until the fixed point in (4.3) is reached. However, direct application of $\mathcal{T}^*$ requires a known model for P and R, which we do not want to assume in general. Therefore, many reinforcement learning techniques use samples to approximate this mapping.

It is not possible to directly sample a value to approximate the mapping used for the state value, for which the fixed point is shown in (4.1), because of the max operator. However, it is possible to sample Q^* as given in (4.2). This results in the Q-learning algorithm which was first proposed by Watkins (1989):

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)\left(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)\right) \quad , \tag{4.5}$$

where $\alpha_t(s,a) \in [0,1]$ is a step-size parameter that averages over the randomness in the rewards and state transitions. In Chapter 2, we discussed Q-learning and its on-policy variant Sarsa.

In Chapter 3, we have seen that choosing the maximal element from a set of random variables can result in an overestimation of the maximal expected

value. In this chapter, we apply that analysis to Q-learning and show that Q-learning can suffer from large overestimations. We present a new algorithm based on the double estimator approach that was introduced in Section 3.4 and we show that it performs much better on some problems, even though it is shown to underestimate the action values in some cases.

4.1 Context and Contributions

Q-learning and Sarsa have been used to find solutions on many problems (Crites and Barto, 1996; Randløv and Alstrøm, 1998; Smart and Kaelbling, 2002; Naghibi-Sistani et al., 2006; Wiering and van Hasselt, 2008) and especially Q-learning was an inspiration to similar algorithms, such as Phased Q-learning (Kearns and Singh, 1999), Fitted Q-iteration (Ernst et al., 2005) and Delayed Q-learning (Strehl et al., 2006), to name some. These variations have mostly been proposed in order to speed up theoretical or empirical convergence rates compared to the original algorithm. Szepesvári (1998) has shown that the convergence rate of Q-learning can be exponential in the number of experiences, while Even-Dar and Mansour (2003) have shown this to be dependent on the learning rates that are used and that with a proper choice of learning rates convergence in polynomial time can be obtained. All the variations named above can claim polynomial time convergence.

4.1.1 Contributions

Our work extends the aforementioned previous work by showing that the performance of Q-learning can suffer from a large overestimation bias caused by the single estimator approach to determine the maximal action value in the next state. Previous work seems to have presumed this bias either does not exist, or is too small to be of relevance. We will show that this bias alone is enough to explain the upper bounds on the convergence rates proposed in earlier work. This implies that these upper bounds on the convergence rates are not merely a theoretical curiosity and that they are tight. Additionally, we link the poor performance to specific types of MDPs, where the worst case settings include MDPs with noisy recurrent connections in the state space. In some settings the bias is severe enough to remove all practical use for the Q-learning algorithm, because it takes too long before reasonable action values are learned. We demonstrate this on some small problems to show that Q-learning can be extremely slow to converge even on problems with just a single state.

We will see that if we remove the overestimation from the algorithm the performance is greatly improved on some of these MDPs. We propose to use an alternative method to find an estimate for the maximum value of a set of stochastic values which we call the double estimator. This methods was dis-

cussed in Chapter 3 where it was shown that this method sometimes underestimates rather than overestimates the maximum expected value. Based on this double estimator method, we present a new model-free off-policy temporal-difference algorithm called Double Q-learning (van Hasselt, 2010). To the best of our knowledge, this is the first off-policy reinforcement learning algorithm that approximates Q^*, but does not have a positive bias in estimating the action values in stochastic environments. We show on some examples that it does not on average overestimate the action values, but that it can suffer from underestimations. Additionally, we conduct some experiments to highlight the strengths and weaknesses of the different approaches and we give pointers to future research.

In our analysis, we consider the average case convergence rates and not the often used probably approximately correct (PAC) convergence rates (Even-Dar and Mansour, 2003; Strehl et al., 2009). Both approaches have advantages. The advantage of an average case analysis is that some problems are not approximately correct to a given degree within a given amount of time, while they still have reasonably good expected values. An example of such a case is any reinforcement learning algorithm with fixed learning rates. For any fixed learning rate, there exists an accuracy that can not be attained because of the sustained noise in the update, although the finite time performance may be quite good.

4.1.2 Q-learning as Biased Value Iteration

In Chapter 2, we discussed the dynamic programming operator $\mathcal{T}^\pi$ as given in equation (2.9). This operator is defined as

$$\forall s \in S: \quad (\mathcal{T}^\pi V)(s) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma V(s') \right) \ . \tag{4.6}$$

Asynchronous policy evaluation works by applying this operator on one state at a time. As discussed in Section 2.3.4, we can sample this mapping each time step and obtain the TD learning update

$$V_{t+1}(s_t) = V_t(s_t) + \beta_t(s_t) \left(r_{t+1} + \gamma V(s_{t+1}) - V_t(s_t) \right) \ . \tag{4.7}$$

If we assume we apply operator $\mathcal{T}^\pi$ to V_t, the target used in update (4.7) is an unbiased sample for the value in (4.6) for state s_t, although it may contain noise due to the stochastic rewards and state transitions. In other words, TD-learning is an unbiased sample based alternative for policy evaluation.

Similarly, Q-learning can be interpreted as a sample based alternative for asynchronous action value iteration, as obtained by applying operator $\mathcal{T}^*$ as defined in (4.4) to some action value function for one state action pair at a time. This asynchronous value iteration update is defined as

$$Q_{t+1}(s,a) = \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} + \gamma \max_{a'} Q_t(s',a') \right) \ . \tag{4.8}$$

This update can be shown to converge to Q^* under mild conditions on the selection of s and a. If we assume s and a are selected deterministically, the action function Q_t is deterministic as well.

However, the Q-learning update in (4.5) is not an unbiased sample for the asynchronous value iteration update in (4.8). More specifically, Q-learning can experience a positive bias in estimating the maximal action values. The reason is that the action values as obtained by Q-learning can be interpreted as noisy samples of the action values that would be obtained by asynchronous value iteration. As such, the expected value of the maximal action value, as used in update (4.5) can be an overestimation for the maximal expected value as used in (4.8). This is a direct result of the overestimation bias of the single estimator that we discussed in Chapter 3. Moreover, the bias can be cumulative, since the biased updated bootstraps on other action values that may already have accrued a positive bias.

In Chapter 3, we showed that overestimations occur when the values in a set that is maximized are noisy estimators for the expected values of the set.[1] In our settings, this translates to the fact that $\max_a Q_t(s,a)$ can contain a bias when $Q_t(s,a)$ is not deterministically determined by the algorithm, the time step t and the state and action. In practice, this means we can expect overestimations when we use Q-learning on a stochastic MDP. Other reasons for noisy action values can also result in overestimations, such as when the initial action values are sampled from some random distribution or when the action values are stored in a noisy manner. In this chapter, for simplicity we will only focus on MDPs with stochastic rewards as a reason for noise in the action values.

In this chapter we will look at the overestimation bias of Q-learning in detail. The main ideas also apply to similar algorithms, such as Sarsa. When Sarsa is used with a fixed policy that is not dependent on the action values, no implicit maximization occurs and there will be no overestimation. However, Sarsa often implicitly uses a max operator, for instance when it is used in combination with an ϵ-greedy policy or even with Boltzmann exploration, since these exploration procedures give the highest selection probability to the highest valued action. For instance, we can say that in stochastic MDPs Q-learning overestimates the optimal action values, while Sarsa with ϵ-greedy exploration overestimates the ϵ-optimal values, where these are defined as the optimal values under an ϵ-greedy policy. We will not consider Sarsa explicitly in the rest of this Chapter.

[1] Although intuitive, this statement is a bit loose. Formally, overestimations occur when there is a non-zero probability that an optimal estimate is not maximal, as proven in Lemma 3.2.

4.1.3 Organization of the Chapter

The chapter is organized as follows. We use the results from Chapter 3 in Section 4.2 to find approximations for the average case overestimation of the value of the maximum arm in multi-armed bandit problems. We compare this with the proposed upper bounds from Chapter 3 and discuss the accuracy of some approximations of the overestimation. This analysis is extended to simple MDPs in Section 4.3 to show that the convergence of Q-learning can suffer from this overestimation. In Section 4.4 we present the Double Q-learning algorithm that extends our analysis in Section 3 and avoids this overestimation. The new algorithm is proven to converge to the optimal solution in the limit, although it can experience underestimations of the action values before converging. In Section 4.5 we show the results on some experiments to compare these algorithms. Some general discussion and related work is presented in Section 4.6.1 and Section 4.6 concludes the chapter with some pointers to future work.

4.2 Overestimations in Bandit Problems

In Chapter 3 we have looked at the problem of estimating the maximum expected value of a set of random variables. In this section we look at a very similar problem: that of determining the value of the maximal arm in a multi-armed bandit problem (Robbins, 1952; Berry and Fristedt, 1985).

The purpose of this section is to connect the discussion in the previous chapter to the reinforcement learning setting. We do this by first considering the simpler multi-armed bandit setting that will be explained in the next subsection. Then we will discuss the bias caused by the single estimator approach that is commonly used for these types of problems. We will find a usable approximation for this bias and use this to determine how many samples are needed before the estimator values converge with a satisfactory accuracy. These results will be used in the next section to show why Q-learning can overestimate the action values in certain MDPs by a wide margin. In Section 4.4 we will reconsider the double estimator in a reinforcement learning context, but in this section we focus only on the single estimator.

Earlier work has focused on the convergence rates in Bandit problems, for instance giving the complexity in a probably approximate correct (PAC) manner (Even-Dar et al., 2002) or focusing on exploration (Auer et al., 2002; Mannor and Tsitsiklis, 2004). This previous work shows a quadratic dependence on relevant parameters of the model, demonstrating that convergence rates are in the order of $\Omega(\frac{1}{\epsilon^2})$, when we only consider the dependence on a given approximation error ϵ to the true value. We will rederive this quadratic dependence and additionally give analytic approximations of the average distance as a function on the number of trials. Rather than just considering

the abstract convergence rate, this analysis gives a direction to this distance, since we show it will in general be an overestimation.

4.2.1 Multi-Armed Bandits

In a multi-armed bandit problem there is a set of M arms $A = \{a_1, \ldots, a_M\}$. Each arm a_i results in some stochastic immediate reward $r(a_i)$, where the expectancy of this reward is some constant $R_i = E\{r(a_i)\}$. The aim for a learning agent is to find the arm j that maximizes the expected reward, such that

$$R_j = \max_i R_i \ . \tag{4.9}$$

Additionally, we want the agent to estimate the expected value R_j for this optimal arm. This value will be important when we consider value-based reinforcement learning in Markov decision processes in Section 4.3. For simplicity, we will assume that the rewards have a binomial distribution and can only be 0 (failure) or 1 (success) as a running example. We will also consider the general case, although the general bounds necessarily will be quite loose. The task is then to find the arm with the highest chance of success.

In the following, in order to be consistent with the reinforcement learning terminology we will talk about actions rather than arms. Furthermore, we will use the shorter term bandits to refer to multi-armed bandits.

To get estimates of the rewards, the agent tries different actions. The obtained rewards are then used to update estimators. We will denote the estimate for the reward of action a by $Q_t(a)$, where $Q_t : A \to \mathbb{R}$ is an adaptable action value function. The estimator is then updated by

$$Q_{t+1}(a_t) = Q_t(a_t) + \alpha_{t+1}(a_t)(r_{t+1}(a_t) - Q_t(a_t)) \ , \tag{4.10}$$

where $r_t(a)$ is the stochastic reward for action a_t in trial t.

Often, exploration is an important topic in the bandit settings. In this section, we will abstract over the exploration issues and will simply talk about trials, where in each trial every action gets selected exactly once. In other words, we assume synchronous updates to the action values. Note that the number of trials needed to converge to a certain solution is a lower bound to the number of actions needed to obtain similar convergence. We use the letter N to denote the number of trials. For now, we assume a learning rate parameter of $\alpha_N(a) = \frac{1}{N}$, where N is the number of times the action a has been selected. Such a learning rate is optimal for stationary bandit problems. Then, we obtain

$$\forall a : Q_N(a) = \frac{1}{N} \sum_{i=1}^{N} r_N(a) \ ,$$

such that $Q_N(a)$ is simply the average of N rewards obtained for arm a.

4.2.2 Overestimations for Binomial Bandits

In this subsection we will analyze the convergence rate as a function of the number of actions and the number of trials for a bandit problem where the rewards are Boolean. We will consider the worst case performance of the single estimator approach. As noted in the former section, the bias of the single estimator is the largest when the different estimators are iid and the bias is related to the size of the variance. We will compute the expected maximal action value in our running example and compare it to the upper bound of $\sigma\sqrt{(M-1)/N}$ that we obtained in Sections 3.3 and 3.7. According to this upper bound, the expected number of trials before the overestimation falls below some value ϵ is

$$N < \sigma^2 \frac{(M-1)^2}{\epsilon} \quad . \tag{4.11}$$

In our running example we assume that each action yields a reward of -1 or 1 according to a Bernoulli distribution with equal probabilities $p = (1-p) = 1/2$ for both success and failure. Then the rewards are iid and the variance is maximal for the given reward range. The values of -1 and 1 were chosen rather then the more common values of 0 and 1 because of convenience, since then the expected value is equal to zero and the variance is equal to one. Then the expected value of the maximal arm corresponds one to one to the overestimation bias. The probability distribution of total number of successes x after N trials then behaves as a binomial distribution $b(x,N,p)$, where $p = 1/2$ is the probability of success on each trial.

The expected maximal value, also known as the maximal order statistic, of a set of random variables Y distributed according to a binomial distribution is equal to

$$E\left\{\max_i Y_i \middle| N,M,p\right\} = N - \sum_{x=0}^{N-1} B(x,N,p)^M \quad , \tag{4.12}$$

where $B(x,N,p)$ is the CDF of the binomial distribution, N is the number of trials, M is the number of actions and p is the probability of success (Gupta and Panchapakesan, 1967; David and Nagaraja, 2003; Arnold et al., 2008). The expected maximum action value in our example then is

$$\begin{aligned} E\left\{\max_a Q_N(a) \middle| M\right\} &= \frac{2E\left\{\max_i Y_i | N,M,\frac{1}{2}\right\} - N}{N} \\ &= 1 - \frac{2}{N}\sum_{x=0}^{N-1} B\left(x,N,\frac{1}{2}\right)^M \quad , \end{aligned} \tag{4.13}$$

which is simply a rescaling of (4.12) to the interval $[-1,1]$ under the assumption $p = 1/2$. Since we will assume these rewards and success probabilities throughout our example, the expected maximal action value is conditioned only on the number of actions M and the number of sample N. We will now

examine the size of the overestimation and give two useful approximations to this value.

Using (4.13), we obtain for respectively 2 and 3 actions

$$E\left\{\max_a Q_N(a)\middle|2\right\} = b\left(N, 2N, \frac{1}{2}\right) ,$$
$$E\left\{\max_a Q_N(a)\middle|3\right\} = \frac{3}{2}b\left(N, 2N, \frac{1}{2}\right) ,$$

both of which can be proven algebraically (Calkin, 1994). Expressions for $M > 3$ are more complex and to the best of our knowledge no closed form expressions simpler than (4.13) exist. However, we can give a simple approximate expression for the overestimation. We can rewrite the values as

$$E\left\{\max_a Q_N(a)\middle|M\right\} = C_M(N)\binom{2N}{N}4^{-N} , \tag{4.14}$$

where $C_1(N) = 0$, $C_2(N) = 1$ and $C_3(N) = 3/2$. For $M \geq 4$, $C_M(N)$ is an unknown function that increases in N. We can approximate (4.14) with Stirling's formula to obtain

$$E\left\{\max_a Q_N(a)\middle|M\right\} \approx \frac{C_M(N)}{\sqrt{\pi N}} . \tag{4.15}$$

Although $C_M(N)$ increases in N, the approximation in (4.15) is a decreasing function of N, for any $M \geq 2$.

For $M \in \{2,3\}$, the approximation in (4.15) is a good approximation for the overestimation of M Gaussian random variables with variance $1/N$. In that case the known expected overestimations are $1/\sqrt{\pi N}$ for $M = 2$ and $3/(2\sqrt{\pi N})$ for $M = 3$ (David and Nagaraja, 2003; Arnold et al., 2008). A closed form expression for $C_M(N)$ for general N and M seems to be an open question in order statistics. However, it can be verified numerically that $C_M(N)$ increases roughly logarithmically in N for $M \geq 4$.

We will now construct an approximate lower bound to how many trials it takes for a given ϵ before the overestimation on average falls below this value. Because we have no general closed form expressions for $C_M(N)$, we will use $C_M(1)$:

$$C_M(1) = \frac{4\left(1 - 2B\left(0, 1, \frac{1}{2}\right)^M\right)}{\binom{2}{1}} = \frac{4\left(1 - 2^{1-M}\right)}{2} = 2 - 2^{2-M} .$$

Since $C_M(N)$ increases in N, we known that in general $C_M(1) \leq C_M(N)$ for all M and N. Using $C_M(1)$ in place of $C_M(N)$, we get

$$E\left\{\max_a Q_N(a)\middle|M\right\} \gtrapprox \frac{2 - 2^{2-M}}{\sqrt{\pi N}} . \tag{4.16}$$

Exact (with equation (4.13))							
M	2	3	4	8	16	32	64
N	3183	7161	10596	20267	31187	42835	54931

Approximations								
	M	2	3	4	8	16	32	64
(4.17)	N	3183	7161	9748	12534	12731	12732	12732
(4.19)	N	3333	7500	10800	18148	23356	26473	28182
$\frac{3}{\pi}$ (4.19)	N	3183	7161	10313	17330	22303	25280	26912

Upper bound (4.11)							
M	2	3	4	8	16	32	64
N	10000	20000	30000	70000	150000	310000	630000

Table 4.1: Average number of trials before the maximum of M actions is within $\epsilon = 0.01$ of the actual value in a bandit problem with Bernoulli ($p = 1/2$) distributions with reward of -1 or 1 for each action. Shown are exact numerical results using (4.13), approximations using (4.17), (4.19) and (4.19) multiplied with a factor $3/\pi$. Also shown is the upper bound obtained with $\epsilon = \sigma\sqrt{(M-1)/N}$.

We can restrict the overestimation to ϵ and then solve (4.16) for N to get what we could call an approximate underestimation of the number of required trials

$$N \gtrapprox \frac{1}{\pi}\left(\frac{2-2^{2-M}}{\epsilon}\right)^2 \quad . \tag{4.17}$$

Alternatively, we can use the uniform approximation from (3.21) to obtain

$$E\left\{\max_a Q_N(a)\middle| M\right\} \approx \frac{M-1}{M+1}\sqrt{\frac{3}{N}} \quad , \tag{4.18}$$

where we have used $\sigma = 1$, as is the case in our example. For $M = 2$ and $M = 3$, (4.16) and (4.18) are very similar: $1/\sqrt{\pi N}$ and $3/(2\sqrt{\pi N})$ for (4.16) compared to $1/\sqrt{3N}$ and $3/(2\sqrt{3N})$ for (4.18). The only difference is a factor of $\sqrt{3}$ instead of $\sqrt{\pi}$ in the denominator. We can solve (4.18) for N to find how long it takes before the expected overestimation falls below ϵ according to the uniform approximation. This gives our second approximation

$$N \approx 3\left(\frac{M-1}{\epsilon(M+1)}\right)^2 \quad . \tag{4.19}$$

To get an idea of the impact of the overestimation bias and to check our approximations we compare our results with the analytical numerical results, as can be obtained by use of (4.17). Table 4.1 gives approximations for the number of trials needed to limit the overestimation to $\epsilon = 0.01$ for various M

according to (4.17) and (4.19). The actual numerical values obtained with use of (4.13) are also shown. Additionally, the values of the second approximation are shown if we multiply the approximation in (4.19) with a factor $3/\pi$.

In Table 4.1, we see that the approximations are accurate for small values of M. For larger M the numerical results and the approximations start to differ. However, for a lower bound the approximations due to (4.19) are very useful. The general upper bound, which was not restricted to binomial distributions, is clearly quite loose in this case.

The approximation that gives the closest result without overestimating the number of trials for any M is

$$E\left\{\max_a Q_N(a)\middle| M\right\} \geq \frac{3\sigma}{\sqrt{\pi N}} \frac{M-1}{M+1} \ , \tag{4.20}$$

as used for the last approximation in Table 4.1. This approximation underestimates the exact numerical results for all $M \geq 4$, which implies that we can use these results as a lower bound for the setting multiple iid actions with equiprobable Boolean rewards, without any risk of exaggerating the problems associated with the found overestimation. We will use this in the next section to analyze the average convergence rate of Q-learning and examine how it is affected by the different relevant parameters.

We have seen that in the iid binomial and iid uniform case the lower bound on the overestimation is linear in $\sigma/\sqrt{N}$. The same holds for the upper bound, so in general we can assume that for a set of M iid values with an arbitrary distribution with variance σ^2, the overestimation is equal to

$$E\left\{\max_a Q_N(a)\middle| M\right\} = x_M \frac{\sigma}{\sqrt{N}} \ , \tag{4.21}$$

where $x_M \in o(M)$ is an unknown value that increases in M and that is strictly positive for $M > 1$. The important thing to note is that x_M does not depend on N. Now, if we want a lower or an upper bound, we can simply replace x_M with the relevant quantity.

4.3 Convergence Rates of Q-learning

In the previous section we have analyzed the overestimation of a single estimator approach in multi-armed bandit problems with only direct rewards. In this section, we will use this analysis to look at the convergence rates of Q-learning.

We will construct MDPs where Q-learning suffers a big performance penalty because of this bias. An analysis of these problems is more involved, because of the possible dependence of state-action values on each other, but the main principle is similar. However, we will see that the quadratic dependence of

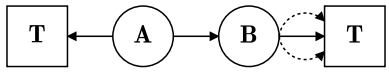

Figure 4.1: Episodic MDP.

the number of required trials to get below a certain overestimation transforms into an exponential dependence when there are recurrent connections.

We will extend our analysis on the bandit problems to episodic and recurrent MDPs. This allows us to give examples of how many steps it can take in some very simple settings before Q-learning converges to any meaningful approximation of the true action values.

4.3.1 Episodic Bandit MDP

In this subsection we look at an MDP that is fairly easy to analyze, since it is a simple extension of a multi-armed bandit. The MDP is shown in figure 4.1. In order to keep the figures simple, we do not show the actions explicitly as we did before. Rather, each line depicts both an action and a state transition. State A is the starting state and the squares represent terminal states. State B is in essence a bandit problem, with multiple actions all leading to a terminal state T. All actions in state B yield a stochastic reward with expected value of zero and a variance of σ^2. The action a_{B} leads from state A to B and yields a reward of zero. The action a_{T} leads from A to T and yields some positive reward and is therefore the optimal action. We will discuss this action later, but we will first focus on the estimation of the value of a_{B}.

For simplicity of the analysis, we will consider the case in which the value of this action is updated with ordinary Q-learning with a learning rate of $1/N$, where N is the number of times the relevant action has been selected. Furthermore, we assume synchronous Q-learning, such that all actions are updated at the same time. By doing this we abstract over any exploration issues.

The expected value of the maximum of the actions in state B as a function of the number of trials N is given in equation (4.21), where $M = |A(\mathrm{B})|$ is the number of actions that lead from state B to state T. If we assume a learning rate of $1/N$, the value of $Q_N(\mathrm{A}, a_{\mathrm{B}})$ is:

$$Q_N(\mathrm{A}, a_{\mathrm{B}}) = \gamma \frac{1}{N} \sum_{n=0}^{N-1} \max_a Q_n(\mathrm{B}, a) \ . \tag{4.22}$$

Assume all values are initialized at zero. Then, using (4.21) we can obtain

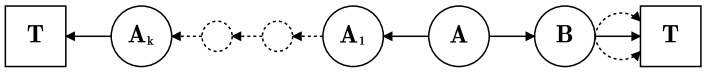

Figure 4.2: Episodic MDP.

the following general overestimation:

$$\begin{aligned} E\left\{Q_N(A,a_{\mathrm{B}})\right\} &= \gamma \frac{1}{N} \sum_{n=0}^{N-1} E\left\{\max_a Q_n(\mathrm{B},a)\right\} \\ &= \frac{\gamma}{N} \sum_{n=1}^{N-1} \frac{x_M \sigma}{\sqrt{n}} \\ &\geq \frac{\gamma x_M \sigma}{\sqrt{N}} \quad , \end{aligned}$$

where the inequality holds for all $N > 3$. We are interested how long it takes on average before this value is within ϵ of the actual value, so we assert $\epsilon = E\{Q_N(A,a_{\mathrm{B}})\}$ and solve for N to get

$$N \geq \left(\frac{\gamma x_M \sigma}{\epsilon}\right)^2 \quad . \tag{4.23}$$

Although this is a lower bound, the quadratic dependence on the relevant parameters seems manageable and scalable.

Equation (4.23) seems to indicate that lower discount factors lead to faster convergence to the optimal policy. This is not necessarily the case in general, as the next example will demonstrate. Consider the MDP in Figure 4.1 and assume that the optimal action in state A is a_{T}, for some positive average reward of $Q(A,a_{\mathrm{T}}) = \epsilon$. Equation (4.23) then gives the approximate average number of trials until the value of the 'right' action is less than this value and the optimal policy is found. Indeed, this value decreases when the discount factor γ decreases. Now consider the similar MDP in Figure 4.2, where the a_1 yields a reward of zero and leads to a different state A_1. This state only has a single action that leads to yet another state A_2, again with a reward of zero. This continues until eventually state A_k is reached, where a single action yields some positive reward ϵ and a terminal state is reached. This implies that the value of a_1 from the original state A is $\gamma^k \epsilon$. Then, the amount of trials before it can be determined on average that a_1 is preferred to a_{B} in state A then is at least

$$N \geq \left(\frac{\gamma x_M \sigma}{\gamma^k \epsilon}\right)^2 = \left(\frac{x_M \sigma}{\gamma^{k-1} \epsilon}\right)^2 \quad ,$$

which increases when γ decreases for all $k \geq 2$. This shows that it is by no means trivial when modeling a problem as an MDP to select a discount factor that in general aids the convergence of an algorithm such as Q-learning.

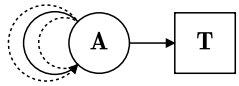

Figure 4.3: An MDP with reflexive actions that yield stochastic rewards.

Based only on the γ-contraction in the proof of convergence, one would assume that lower γ always results in faster convergence. Although this is true when only considering the action values, the policy may converge faster when γ is larger, as in our example in Figure 4.2. Also note that the average convergence rate is no longer necessarily quadratic in all the relevant parameters, as the exponent of γ now depends on the length of the sequence $\mathrm{A}_1, \ldots, \mathrm{A}_k$.

4.3.2 Loop Bandit MDP

In our next example, we are again dealing with a bandit-type state, but this time multiple actions lead back to the same state. For completeness, there is also a terminal state that can be reached, but we will defer discussion about this option until later, so for now we ignore the corresponding action. The MDP is shown in Figure 4.3. We will demonstrate that in this MDP Q-learning obtains the theoretical worst-case bound of $O(1/N^{1-\gamma})$ as found in the asymptotic case by Szepesvári (1998) and as found for finite sets of experiences by Even-Dar and Mansour (2003). Although simple in structure, the MDP in Figure 4.3 can therefore be considered a difficult problem for Q-learning.

After N trials, we have the following value for any arbitrary action a:

$$\begin{aligned} Q_N(A,a) &= \frac{N-1}{N} Q_{N-1}(A,a) + \frac{1}{N}\left(r_N(a) + \gamma \max_b Q_{N-1}(A,b)\right) \\ &= \ldots \\ &= \frac{1}{N}\sum_{n=1}^{N}\left(r_n(a) + \gamma \max_b Q_{n-1}(A,b)\right) \ , \end{aligned}$$

where $r_n(a)$ is the reward received by action a on trial n. In this section, M denotes the number of actions that lead from state A back to itself.

Since all actions loop back to the same state, the expected maximum over these values can be found as follows:

$$\begin{aligned} & E\left\{\max_a Q_N(\mathrm{A},a)\right\} \\ &= E\left\{\max_a\left(\frac{1}{N}\sum_{n=1}^{N}\left(r_n(a) + \gamma \max_b Q_{n-1}(\mathrm{A},b)\right)\right)\right\} \\ &= E\left\{\max_a \frac{1}{N}\sum_{n=1}^{N} r_n(a)\right\} + E\left\{\frac{\gamma}{N}\sum_{n=1}^{N}\max_b Q_{n-1}(\mathrm{A},b)\right\} \ , \end{aligned} \tag{4.24}$$

where the first term is equal to the expected average value of the maximal action in a bandit problem

$$E\left\{\max_a \frac{1}{N}\sum_{n=1}^{N} r_n(a)\right\} = \frac{x_M\sigma}{\sqrt{N}} \ .$$

The derivation in (4.24) shows that the expected maximum value is related over different time steps in the same way as the following process:

$$q(N) = \frac{x_M\sigma}{\sqrt{N}} + \frac{\gamma}{N}\sum_{n=0}^{N-1} q(n)$$

We construct an approximation to the derivative of $q(N)$:

$$\begin{aligned}
\frac{d}{dN}q(N) &= \left(\frac{d}{dN}\frac{x_M\sigma}{2\sqrt{N}}\right) + \left(\frac{d}{dN}\frac{\gamma}{N}\right)\sum_{n=0}^{N-1} q(n) + \frac{\gamma}{N}\left(\frac{d}{dN}\sum_{n=0}^{N-1} q(n)\right) \\
&\approx -\frac{x_M\sigma}{2N\sqrt{N}} - \frac{\gamma}{N^2}\sum_{n=0}^{N-1} q(n) + \frac{\gamma}{N}q(N) \\
&= -\frac{x_M\sigma}{2N\sqrt{N}} - \frac{1}{N}\left(q(N) - \frac{x_M\sigma}{\sqrt{N}}\right) + \frac{\gamma}{N}q(N) \\
&= \frac{x_M\sigma}{2N\sqrt{N}} - \frac{1-\gamma}{N}q(N)
\end{aligned}$$

where we used the discrete approximation

$$\frac{d}{dN}\sum_{n=0}^{N-1} q(n) \approx (\sum_{n=0}^{N} q(n)) - (\sum_{n=0}^{N-1} q(n)) = q(N) \ .$$

This shows that the derivative of the expected maximal action value approximately has the structure of the ordinary differential equation (ODE)

$$q'(N) = \frac{x_M\sigma}{2N\sqrt{N}} - \frac{1-\gamma}{N}q(N) \ .$$

We solve the ODE to get the approximation

$$E\left\{\max_a Q_N(\mathrm{A},a)\right\} \approx KN^{\gamma-1} - \frac{x_M\sigma}{\sqrt{N}}\frac{1}{2(\gamma-1/2)} \ ,$$

where K is a constant due to the integration needed to find the solution of the ODE. The overestimation for $N = 1$ is equal to $x_M\sigma$ and we use this to find $K = \gamma x_M\sigma/(\gamma - 1/2)$. This means we have the approximation

$$E\left\{\max_a Q_N(\mathrm{A},a)\right\} \approx \frac{x_M\sigma}{\gamma-1/2}\left(\gamma N^{\gamma-1} - \frac{1}{2\sqrt{N}}\right) \ . \tag{4.25}$$

We cannot easily find an analytical function of N that tells us how long it takes before the overestimation falls below some ϵ, but we can investigate the different parameters and observe their influence on the time before convergence. We start by noting that the values for $\gamma = 0$ and $\gamma = 1$ are respectively

$$E\left\{\max_a Q_N(\mathrm{A},a)\middle|\gamma=0\right\} \approx \frac{x_M\sigma}{\sqrt{N}} \ ,$$
$$E\left\{\max_a Q_N(\mathrm{A},a)\middle|\gamma=1\right\} \approx x_M\sigma\left(2-\frac{1}{\sqrt{N}}\right) \ .$$

The first result shows that if the discount factor is zero, the approximation reduces to the same value as in the direct reward case described in the former section, as one would expect. The second result implies that when $\gamma = 1$, the overestimation strictly increases from $x_M\sigma$ to $2x_M\sigma$ for increasing N. Therefore, with $\gamma = 1$ we do not have convergence. This is not an artifact of the approximation; the true value of all actions is zero, so we can deduce that any overestimation must be positive. Because none of these overestimations get discounted when $\gamma = 1$, they continue to be represented in the action values, resulting in a lack of convergence. Approximation (4.25) is not defined for $\gamma = 1/2$, although it is continuous in the region. Solving the ODE directly for $\gamma = 1/2$, we get:

$$E\left\{\max_a Q_N(\mathrm{A},a)\middle|\gamma=1/2\right\} \approx \frac{x_M\sigma}{\sqrt{N}}\left(1+\log\sqrt{N}\right) \tag{4.26}$$

Another way to interpret the approximation in equation (4.25) is by noting that for $\gamma \in (1/2,1)$ the term $N^{\gamma-1}$ decreases with a slower rate than $1/\sqrt{N}$. That implies that if N is large enough, the approximation behaves roughly as:

$$E\left\{\max_a Q_N(\mathrm{A},a)\middle|\gamma\in\left(\frac{1}{2},1\right),N\gg 1\right\} \approx x_M\sigma\frac{\gamma N^{\gamma-1}}{\gamma-1/2} \ .$$

If we want to know how long it approximately takes before the value of the maximum action falls below some other value ϵ, we can solve this to get

$$N \approx \left(x_M\sigma\frac{\gamma}{\epsilon(\gamma-1/2)}\right)^{1/(1-\gamma)} \ .$$

This can be an overestimation of the true number of trials since we ignored the negative second term in equation (4.25). However, N becomes sufficiently large for small values of ϵ that it does give an accurate idea of the rate of convergence. This value can become very large when γ is close to one.

The slow rate of convergence is also apparent when using the previous, more accurate approximation from (4.25). Table 4.2 shows the approximations for N for a few combinations of M,γ and ϵ. We used the uniform lower bound of $x_M = (3/\sqrt{\pi})(M-1)/(M+1)$ and a variance of $\sigma^2 = 1$. Consider the

Value of N, according to approximations (4.25) and (4.26)				
ϵ		0.5	0.1	0.01
$\gamma = 0$	$M = 2$	2	32	3,183
	$M = 10$	8	192	19,178
$\gamma = 0.5$	$M = 2$	4	550	154,867
	$M = 10$	78	5,377	1,230,936
$\gamma = 0.9$	$M = 2$	9,649	1.09×10^{10}	1.09×10^{20}
	$M = 10$	88,000,282	8.63×10^{13}	8.63×10^{23}
$\gamma = 0.95$	$M = 2$	34,472,155	3.30×10^{20}	3.30×10^{40}
	$M = 10$	2.18×10^{14}	2.08×10^{28}	2.08×10^{48}

Table 4.2: Approximations for the number of steps N until Q-learning converges within ϵ of the actual value in a reflexive stochastic MDP (without terminal state) for different numbers of reflexive actions M and discount factors γ.

values of N for $\epsilon = 0.1$ and $\epsilon = 0.01$ for the higher discount factors. When ϵ decreases with a factor 10, the value of N increases with a factor $10^{1/(1-\gamma)}$. Furthermore, when the number of actions becomes fairly large, the number of trials to get within a meaningful region around the actual value becomes too high to be of practical use. A setting with 10 actions and a discount factor of 0.9 is by no means uncommon. However, if the MDP contains a loop, such as the one in Figure 4.3 with iid stochastic rewards, we see it can take millions of visits to this state before the overestimation by Q-learning of the highest valued action in this state reduces below 0.5. This will rarely be accurate enough.

As a verification of the approximations used to construct Table 4.2 we conduct a small experiment. We ran Q-learning on the loop MDP for $N = 1,000$, $N = 2,000$ and $N = 4,000$ trials with a discount factor of $\gamma = 0.9$ and a uniform random reward between $-1/\sqrt{3}$ and $1/\sqrt{3}$ on each transition. For $M = 2$ actions, this resulted in overestimations of respectively $\epsilon = 0.611$, $\epsilon = 0.575$ and $\epsilon = 0.540$, showing slow convergence towards 0.5 at more or less the same rate as predicted. For $M = 10$ actions the overestimations were $\epsilon = 1.585$, $\epsilon = 1.492$ and $\epsilon = 1.402$. These values are averages over 1,000 experiments. The decrease for $M = 10$ is approximately constant for each doubling in the number of trials. We can extrapolate these values to find that for $M = 10$ it would take approximately 4 million trials to reach 0.5. This is faster than the value of 88 million shown in the Table 4.2—which may be due to the extrapolation—but it is much too long to be of practical use.

The analysis and the resulting approximations for the number of steps indicate that one should be careful when using Q-learning on problems with

a somewhat large discount factor, even if these problems look deceptively simple. However, as we saw before some problems need models with a high discount factor to be able to reach the desired optimal policy. For instance, consider the case where the optimal action does not lead to a high reward immediately, but rather only after k steps, similar to the episodic MDP in Figure 4.2. Suppose that a state has reflexive connections, similar to state A in Figure 4.3, but that the intended behavior is to select an action that leads to a reward of 1 in k steps. Now assume there is also a distractor action which leads to a reward of $\frac{1}{2}$ in m steps. If $m < k$ the discount factor should be at least $1/(2^{1/k-m})$ for the optimal policy in the MDP to correspond to the intended behavior that leads to the reward of 1. This translates to a required discount factor larger than 0.9 for $k - m \geq 7$ and a required discount larger than 0.95 when $k - m \geq 22$. If the discount factor is chosen to be lower, the intended behavior will no longer be optimal in the resulting MDP and will therefore not be learned.

These results imply that there exist problems in which Q-learning takes a prohibitive long time to converge to any meaningful policy. This result is similar to the asymptotic bound of $O(1/N^{1-\gamma})$ found by Szepesvári (1998) and the result that there exist MDPs such that after $N = \Omega((1/\epsilon)^{1/(1-\gamma)})$ steps the maximum Q value is further than ϵ from the optimal value as described by Even-Dar and Mansour (2003). However, we have shown that the overestimation bias can cause Q-learning to attain these upper bound convergence rates in very simple MDPs, causing unacceptable slow convergence.

Since we have analyzed the overestimation as a cause for the poor convergence rates, we can devise of a way to solve these issues. In the next section we present a version of Q-learning that is inspired by the double estimator we presented in Chapter 3.

4.4 Double Q-learning

Q-learning uses the single estimator to estimate the value of the next state: $\max_a Q_t(s_{t+1}, a)$ is an estimate for $E\{\max_a Q_t(s_{t+1}, a)\}$, which in turn approximates $\max_a E\{Q_t(s_{t+1}, a)\}$.[2] In the last section we have seen this can cause overestimations that result in slow convergence. In the next section we will verify empirically on some simple problems that the overestimations can be substantial. Therefore, in this section we present an algorithm to avoid these overestimation issues. The algorithm is based on the double estimator approach and therefore it is called Double Q-learning (van Hasselt, 2010).

[2]The expectation should be understood as averaging over all possible runs of the same experiment and not—as it is often used in a reinforcement learning context—as the expectation over the next state, which we will encounter in the next subsection as $E\{\cdot|P_t\}$.

Algorithm 5 Double Q-learning

1: Given an initial state distribution I and an MDP.
2: Initialize Q^A,Q^B,$s \sim I$
3: **repeat**
4: Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$
5: Observe r, s'
6: Choose (e.g. random) either UPDATE(A) or UPDATE(B)
7: **if** UPDATE(A) **then**
8: Define $a^* = \arg\max_a Q^A(s',a)$
9: $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
10: **else if** UPDATE(B) **then**
11: Define $b^* = \arg\max_a Q^B(s',a)$
12: $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
13: **end if**
14: $s \leftarrow s'$
15: **until** end

Double Q-learning is shown in Algorithm 5. Two action value functions are stored: Q^A and Q^B. Each action value function is updated with a value from the other action value function for the next state. To be more precise, the action a^* in line 8 is the maximal valued action in state s', according to the action value function Q^A. However, instead of using the value $Q^A(s',a^*) = \max_a Q^A(s',a)$ to update Q^A, as Q-learning would do, we use the value $Q^B(s',a^*)$. Since Q^B was updated on the same problem, but with a different set of experience samples, this can be considered an unbiased estimate for the value of this action. A similar update is used for Q^B, using b^* and Q^A. It is important that both Q functions learn from separate sets of experiences, so is why we suggest choosing one of the values functions to update at random on each time step. One could also update the action values functions in turn, such that if $Q^A(s,a)$ is updated, $Q^B(s,a)$ will be updated the next time the agent selects action a in state s. However, this would require extra bookkeeping, which is why we prefer the random approach.

To select an action to perform, one can use both value functions, for instance by simple averaging $Q_t(s,a) = (Q_t^A(s,a) + Q_t^B(s,a))/2$. Then, the obtained value function Q_t uses all the available experiences up to time t. In this way, Double Q-learning is as data-efficient as Q-learning, which would not be the case if we would only use Q^A or Q^B for the action selection since each of these only uses approximately half of the experiences. In our experiments, we calculate the average of the two value functions for each action and then perform ϵ-greedy exploration with the resulting averaged action values.

Double Q-learning is not a full solution to the problem of finding the maximum of the expected values of the actions. Similar to the double estimator

in Chapter 3, action a^* may not be the action that maximizes the expected Q function $\max_a E\{Q^A(s',a)\}$. In general $E\{Q^B(s',a^*)\} \leq \max_a E\{Q^B(s,a)\}$, and underestimations of the action values can occur. We will see an example MDP in which this causes some underestimation of the action values later in this chapter. However, in some of the settings in which Q-learning performs very poorly, Double Q-learning shows fast convergence, making it an interesting algorithm for further consideration.

Similar to the action selection, one could imagine using both Q functions to determine the value of the next state for each of the updates. However, using the average of $Q^A(s',b^*)$ and $Q^B(s',a^*)$ to update both of the values functions would not be a good idea. The value-functions would become equal, and the resulting algorithm would be similar to Q-learning with the associated overestimation problems. Possibly, one could use a weighted average of $Q^A(s',a^*)$ and $Q^B(s',a^*)$ to update Q^A and a weighted average of $Q^A(s',b^*)$ and $Q^B(s',b^*)$ to update Q^B. This would result in a hybrid algorithm between Q-learning and Double Q-learning, which in some cases overestimates and in some cases underestimates the action values. Although interesting, we do not consider this variant further. For completeness, we note that using $Q^A(s',b^*)$ for Q^A and $Q^B(s',a^*)$ for Q^B yields a very similar algorithm to the one described in Algorithm 5, with similar performance.

4.4.1 Convergence in the Limit

In this section we show that in the limit Double Q-learning converges to the optimal policy. Intuitively, this is what one would expect: Q-learning is based on the single estimator and Double Q-learning is based on the double estimator and in Chapter 3 we showed that the estimates by the single and double estimator both converge to the same answer in the limit. However, this argument does not transfer immediately to bootstrapping action values, so we will prove the convergence of Double Q-learning formally, making use of Lemma 2.1 that was introduced in Section 2.4.4.

The conditions for convergences for Double Q-learning are similar to those for Q-learning. This includes some conditions on the learning rates and the exploration, but these conditions are minimal in the sense that any stochastic approximation process must have similar conditions to ensure convergence with probability one. Our theorem is as follows:

Theorem 4.1. *Assume the conditions below are fulfilled. Then, in a given ergodic MDP, both Q^A and Q^B as updated by Double Q-learning as described in Algorithm 5 will converge to the optimal value function Q^* as given in the Bellman optimality equation (4.2) with probability one if an infinite number of experiences in the form of rewards and state transitions for each state action pair are given by a proper learning policy. The additional conditions are:*

1. *The MDP is finite, i.e.* $|S \times A| < \infty$,
2. $\gamma \in [0,1)$,
3. *the action values are stored in a lookup table,*
4. *both* Q^A *and* Q^B *receive an infinite number of updates,*
5. $\alpha_t(s,a) \in [0,1]$, $\sum_t \alpha_t(s,a) = \infty$, $\sum_t (\alpha_t(s,a))^2 < \infty$ *w.p.1,* $\forall s,a \neq s_t,a_t$: $\alpha_t(s,a) = 0$,
6. $\forall s,a,s' : Var\{R^{s'}_{sa}\} < \infty$.

A 'proper' learning policy ensures that each state action pair is visited an infinite number of times. For instance, in a communicating MDP proper policies include a random policy. It is easy to extend the proof to non-ergodic MDPs, as long as we have some way to obtain an infinite number of experiences for each state-action pair. Also, the proof can be extended to episodic undiscounted MDPs. For conciseness, we do not explicitly discuss the adaptations needed to prove these additional cases, but the changes to the proof are straightforward. The proof is shown in Section 4.7.1.

4.4.2 Variants of Double Q-learning

Although we will not consider them here in any detail, any variant of Q-learning can be transformed into a Double Q-learning variant. This includes algorithms such as Fitted Q-iteration (Ernst et al., 2005), Delayed Q-learning (Strehl et al., 2006) and Greedy-GQ (Maei et al., 2010). Additionally, other standard extensions to Q-learning such as function approximation in large state spaces are straightforward to implement. In short, using the double estimator in place of the single estimator does not cause significant limitations in the use of the algorithm. Therefore, we focus the rest of our analysis on the potential difference in performance that may result from this adaptation.

4.5 Experiments

This section will show results on three small experimental settings. This will be useful as a check that our analyses are valid, as an illustration of the bias of Q-learning and as a first practical comparison with Double Q-learning. The settings are purposefully kept simple, because in simple settings it is easier to analyze the precise behavior and the reasons for this behavior for the algorithms. In a large complex setting other issues than the ones we are focusing on may weigh more heavily on the performance. Furthermore, since Double Q-learning shares the same scalability issues and remedies as Q-learning does, we do not feel complex experiments would aid much to our

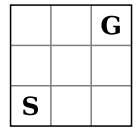

Figure 4.4: A small grid MDP.

comparison of the virtues of both algorithms. As mentioned, any variant of Q-learning can also be transformed into a similar variant of Double Q-learning. In our view, this stresses the importance of a good understanding of the basic algorithm, and these experiments are intended to help build some intuitions.

The settings are the gambling game of Roulette, a small grid world and a small episodic task. In the first two tasks, there is considerable randomness in the rewards, and as a result we see that indeed Q-learning performs quite poorly. The third task serves as an illustration of an MDP in which Double Q-learning is outperformed by Q-learning.

4.5.1 Grid World

We start with a very small grid world MDP as shown in Figure 4.4. In each state there are 4 actions, corresponding to the directions the agent can go. The starting state is in the lower left position and the goal state is in the upper right. There are no obstacles and each time the agent selects an action that would walk off the grid, the agent stays in the same state. The agent receives a random reward of -11 or $+9$ with equal probability. In the goal state every action always yields $+5$ and ends an episode. The optimal policy ends an episode after five actions, so the optimal average reward per step is $+0.2$. The discount factor was set to 0.95.

We conducted two experiments on this MDP, varying the learning rates. The learning rate was either linear: $\alpha(s,a) = 1/n(s,a)$, or polynomial $\alpha(s,a) = 1/n(s,a)^{0.8}$. The learning rates for Double Q-learning were dependent on the number of times the relevant action values function had been updated. For instance, if in the linear case Q^A was to be updated, the learning rate was $\alpha(s,a) = 1/n^A(s,a)$, where $n^A(s,a)$ is the number of updates to $Q^A(s,a)$ up to and including the present update.

Exploration was ϵ-greedy, selecting the greedy action with probability $(1-\epsilon)$ and a random action otherwise. The exploration parameter was $\epsilon(s) = 1/\sqrt{n(s)}$, assuring infinite exploration in the limit which is a theoretical requirement for the convergence of Q-learning and Double Q-learning. Figure 4.5 shows the average rewards in the first row and the maximum action value in the starting state of the MDP in the second row.

The performance of Double Q-learning as measured in average rewards was clearly much better than that of Q-learning in this setting. However,

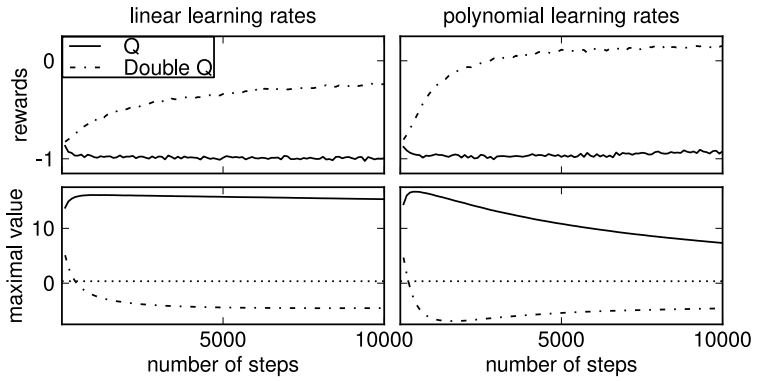

Figure 4.5: Results in the grid world for Q-learning and Double Q-learning. The first row shows average rewards for the stochastic setting with rewards of -12 or $+10$. The second row shows the maximal action value in the starting state S for the stochastic setting. Averaged over 10,000 experiments.

the fact that Double Q-learning obtains better average rewards than normal Q-learning does not necessarily imply that their estimations of the action values are more accurate. We examined the value of the maximally valued action in the starting state, which was almost always the up or right action, as expected. Note that the optimal value of this action would be $5\gamma^4 - \sum_{k=0}^{3} \gamma^k \approx 0.36$, which is depicted in the second row of Figure 4.5 with a dotted line. We see that despite its better average rewards, Double Q-learning gets only slightly closer to this value in $10,000$ learning steps than Q-learning.

The obvious question is why Double Q-learning performs so much better when its action values are not necessarily much more accurate. The answer lies in the direction of the bias. If actions that are selected often suffer from overestimations, these actions will only get selected more and more. On the other hand, if actions that are selected suffer from underestimations, this will give an incentive to the algorithm to try other actions. In this simple grid world, this translated to good policies for Double Q-learning, that may underestimate the values of its actions, but still quickly favors the actions that result in high rewards above actions that do not. For Q-learning, the behavior was typical for the overestimation. Often, the Q-learning agent would walk to a corner and there is would bump into the corner. The state then does not change, making the setting similar to the single state loop MDP we discussed earlier. Eventually, Q-learning may learn that the corner state is not as highly valued as it thought it was, but then there are still plenty of other places to bump into the wall, resulting in very slow convergence to the desired behavior, even when polynomial learning rates are used.

4.5.2 Roulette

In the game of roulette, a small ball is thrown into a rotating wheel in which there are 38 compartments. Each compartment has a color and a number associated with it. The numbers 00 and 0 are green, the numbers 1-36 are each either black or red. The player can choose between 170 different betting actions, including placing a bet on any of the numbers, on either of the colors black or red, on either of the odd or even numbers, and so on. Although the winning probabilities differ between certain bets, the payoff for winning each of these bets is chosen such that almost all bets have an expected payout of $\frac{1}{38}\$36 = \$0.947$ per dollar bet, resulting in an expected loss of -$0.053 per play if we assume the player bets $1 every time.[3] In addition to the betting actions, we introduce an action that stops playing. This action yields a reward of $0 and ends the episode. Naturally, the optimal policy is to stop playing as soon as possible.

We modeled roulette as an MDP with one state and 171 actions, of which one terminates the episode, similar to the MDP in Figure 4.3. The discount factor was 0.95. We ignore the available funds of the player as a factor and assume he bets $1 each turn.

One may rightfully note that roulette is perhaps better modeled as a bandit problem, without any state transitions and discount factor. However, we still feel the problem is interesting to discuss as a discounted MDP, since such settings may occurs as part of a larger MDP. As we will see, Q-learning will have difficulty walking away from the table. In a larger MDP this may lead to unexpected behavior. If a roulette-type setting is part of the larger MDP, one may observe that the performance of Q-learning on the larger MDP is very poor. Without the analysis of the subproblems, it could then be hard to find out where this poor performance stems from. Perhaps this has even occurred in the application of Q-learning to real-world problems, resulting in people thinking unfavorably about the algorithm, although in fact the poor behavior is only the result of a rarely noticed bias that actually can be prevented.

Figure 4.6 shows the mean action values over all actions, as found by Q-learning and Double Q-learning as a function of the number of trials. Each trial consisted of a synchronous update of all 171 actions, using the previous action values. After 1 million trials, Q-learning with a linear learning rate values all actions at more than $20 and there is little progress in the convergence towards the real value. Considering that the agent only bets $1 per turn and only a limited set of the actions have a maximal possible reward which is larger than $20, the overestimation is huge. When we use polynomial learning rates, the performance increases, but Double Q-learning converges close to the actual average action values much more quickly. Fur-

[3] Only the so called 'top line' which pays $6 per dollar when 00, 0, 1, 2 or 3 is hit has a slightly lower expected value of -$0.079 per dollar.

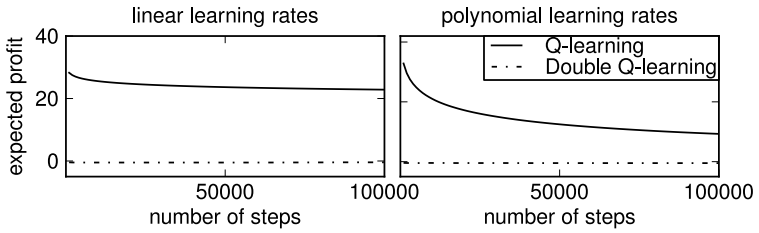

Figure 4.6: The average action values according to Q-learning and Double Q-learning when playing Roulette with a 'walk-away' action worth \$0. Averaged over 10 experiments.

thermore, the overestimation after 100,000 steps is still approximately \$10 on average, which is almost 200 times the actual difference between the value of the optimal action and the values of the suboptimal actions.

4.5.3 Episodic Bandit MDP

As a last experiment, we look at a setting where Q-learning performs better than Double Q-learning. We revisit a version of the episodic MDP in Figure 4.1. Assume there is one 'right' action in state A that yields zero reward, and leads to state B. In state B there are two actions leading to the terminal state T. In our first setting, one of the actions in B has a deterministic reward of -1, the other a deterministic reward of 1. In our second setting, we construct a somewhat worst-case setting for Double Q-learning. In Section 3.6 we established that when we want to obtain the maximum of two uniform distributions, the double estimator has the largest underestimation if the means of these distributions lie one third of the width of the PDFs apart. Therefore, we construct an MDP with the same average rewards as before, but with exactly this overlap of the distributions. One action then yields a uniform random value between -4 and 2, while the other action yields a uniform random value between -2 and 4. In both cases the discount factor is 0.95 and exploration is ϵ-greedy with $\epsilon(s) = 1/\sqrt{n(s)}$.

The results of experiments on the two MDPs described above are shown in Figure 4.7. The figure shows the average value of the action 'right' in state A. The first row shows the results in the deterministic MDP. The second row shows the results for the stochastic MDP. As predicted, Q-learning performs best in this setting and even in the stochastic setting only experiences mild overestimations during learning.

Although Double Q-learning performs worse than Q-learning, the differences are not as large as in the previous two settings and Double Q-learning does not perform terribly. This is largely due to the smaller number of actions. However, if we increase the number of suboptimal actions the performance of

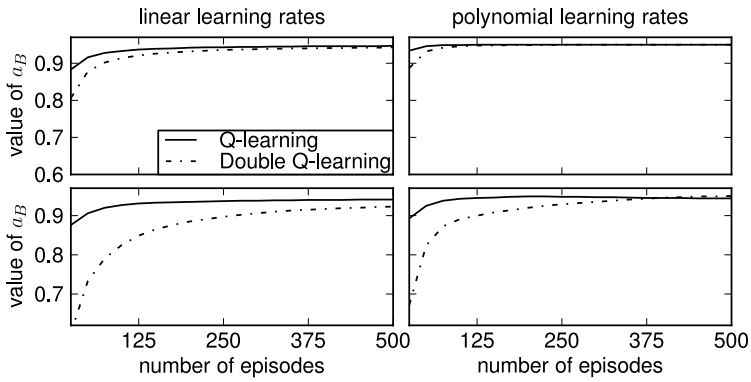

Figure 4.7: Average action values for the 'right' action in state A according to Q-learning and Double Q-learning for the episodic MDP. The first row has deterministic reward of -1 and 1 in state B, the second has stochastic rewards with PDFs $u(x,-4,2)$ and $u(x,-2,4)$. Optimal value is equal to the discount factor of 0.95. Averaged over 1,000 experiments.

Q-learning also deteriorates. We did not find a setting in which the advantage for Q-learning was as striking as the advantage for Double Q-learning in the grid world and in the roulette task.

4.6 Conclusion

The first aim of this chapter is to show that the popular reinforcement learning algorithm Q-learning can sometimes perform very poorly and why this is the case. Our analysis shows that an important reason for this poor performance lies in the overestimation that Q-learning can experience on each update, when the value of the maximum action in the next state is considered. This is especially relevant in problems in which multiple actions yield stochastic, overlapping rewards. The second aim of this chapter is to present, analyze and empirically test the new algorithm called Double Q-learning, that results from applying the double estimator from Chapter 3 to Q-learning. Double Q-learning can be used as an alternative for Q-learning, since it can perform quite well in settings in which Q-learning suffers from prohibitively large overestimations. However, our analysis shows that Double Q-learning also has a bias that can affect performance, so more research is still in order.

4.6.1 Discussion

There is an important difference between the well-known heuristic exploration technique of optimism in the face of uncertainty (Kaelbling et al., 1996;

Sutton and Barto, 1998) and the overestimation bias that is discussed in this chapter. In short, overestimation of action values can be a good idea if it represents an optimistic view on uncertain events. However, as is for instance clearly visible in the roulette task that we discussed, Q-learning can easily overestimate values on which already much information is gathered. Additionally, the overestimated value can be far larger than is practical useful to encourage exploration. In contrast, although Double Q-learning can underestimate the value of experienced actions, it is in principle quite feasible to set the initial action values optimistically enough to ensure optimism for actions that have experienced limited updates. Therefore, the use of the technique of optimism in the face of uncertainty can be thought of as an orthogonal concept to the over- and underestimation that is the topic of this chapter.

Although we have not focused on this in depth, a similar overestimation is experienced by Sarsa when the exploration policy becomes more greedy. However, in contrast to Q-learning, eligibility traces for Sarsa are well defined and unambiguously implementable (Rummery and Niranjan, 1994). Suppose a greedy selection of an action yields a certain, positively biased value. This value will then probably be decreased through the eligibility traces of that action in the next few steps, if indeed the value was an overestimation. When the traces parameter is set close to one, the procedure resembles a Monte Carlo method, which indeed is unbiased. However, the variance in the updates may increase, which may result in poor behavior. This in part explains why intermediate eligibility trace parameters work best in practice: in a sense this is a trade-off between bias and variance (Sutton and Singh, 1994).

We have not mentioned in detail many of the other approaches to polynomial time reinforcement learning. Some interesting model-based algorithms with known, polynomial bounds include E^3 (Kearns and Singh, 2002), R-max (Brafman and Tennenholtz, 2003) and MBIE (Strehl and Littman, 2005). The model-free, related algorithm of Delayed Q-learning (Strehl et al., 2006) has already been mentioned above. For an overview of these algorithms see the work by Strehl et al. (2009).

The analysis in this chapter is not only applicable to Q-learning. For instance, in a recent paper on multi-armed bandit problems, methods were proposed to exploit structure in the form of the presence of clusters of correlated arms in order to speed up convergence and reduce total regret (Pandey et al., 2007). The value of such a cluster in itself is an estimation task and the proposed methods included taking the mean value, which would result in an underestimation of the actual value, and taking the maximum value, which is a case of the single estimator and results in an overestimation as can be shown using the results in Section 4.2. If for instance the PDFs of the random variables are known to be binomial, the performance of the resulting hierarchical algorithm can potentially greatly be improved by being aware of

the overestimation and trying to analytically determine an estimate of said overestimation in order to get a more unbiased estimate of the value of such a cluster of bandits.

As a final remark, we would like to mention that our analysis on the convergence rates for Q-learning is not limited to MDPs with stochastic reward functions. When the rewards are deterministic, but the state transitions are stochastic, the same pattern of overestimations due to this noise can occur and the same analysis as for stochastic rewards holds, although the MDPs on which Q-learning is most affected will look somewhat different. This extension is relatively straightforward and we leave it for future work.

4.6.2 Future Work

There are a couple of pointers to future work. We believe the most interesting of these is the extension of the discussion in this chapter to a fully unbiased form of Q-learning. For this, the analysis in Chapter 3 can be useful, since it gives some indications on how large the overestimation and underestimation biases of the single and double estimator are. For instance, we have given a precise value of the overestimation bias for independent and identically uniformly distributed random variables with known parameters. In principle, one could estimate, for instance in a Bayesian manner (Smith and Winkler, 2006), the PDFs of the different random variables and then make an unbiased estimate on the overestimation itself. When the parameters and distribution are not known, one can for instance assume Gaussian distributions and then determine the expected means and variances for the given data. The maximum of a large number of Gaussians is hard to find analytically, but it can be approximated by eliminating pairs of Gaussians and assuming the resulting distribution of the maximum of such a pair is again Gaussian in shape (Clark, 1961). More sophisticated methods can also be used, although care must be taken in the reinforcement learning setting that the PDFs of the random variables that are to be modeled are not stationary. This complicates the analysis and it is not clear if there is a reliable way to estimate the bias in general.

Perhaps the best option to reduce the bias further is to use an N-fold version of Double Q-learning, which stores N Q functions, rather than just two. Then, similarly to cross-validation, the bias can be reduced at the cost of some more computations per step. These variants of Double Q-learning still needs to be investigated to see if it is a real improvement over the standard Double Q-learning algorithm.

Furthermore, we believe more analysis on the performance of Q-learning and related algorithms such as Fitted Q-iteration (Ernst et al., 2005) and Delayed Q-learning (Strehl et al., 2006) is relevant. For instance, we believe Delayed Q-learning can suffer from the same overestimation, although it does attain polynomial convergence bounds. We believe this is similar to the poly-

nomial learning rates: although performance is improved from an exponential to a polynomial dependence on the relevant parameters, the algorithm still suffers from the inherent overestimation bias due to the single estimator approach.

Finally, more research is desirable on the Double Q-learning algorithm and its potential variants, such as Fitted Double Q-learning and Delayed Double Q-learning. Thus far, Double Q-learning's performance seems promising, but we have seen that in some cases Q-learning performs better. In the next chapter, we will introduce some other alternatives to Q-learning and we will see that Double Q-learning sometimes performs better than these alternatives and sometimes it does not. However, as far as we know, Double Q-learning is the only off-policy reinforcement learning algorithm without a positive bias in its action values, which makes it an interesting topic for further research.

4.7 Proofs

This section contains the proof for Theorem 4.1.

4.7.1 Proof for Theorem 4.1

Proof. Because of the symmetry in the updates on the functions Q^A and Q^B it suffices to show convergence for either of these. We will apply Lemma 2.1 with $P_t = \{Q_0^A, Q_0^B, s_0, a_0, \alpha_0, r_1, s_1, \ldots, s_t, a_t\}$, $X = S \times A$, $\Delta_t = Q_t^A - Q^*$, $\zeta^A = \alpha$ and

$$F_t(s_t, a_t) = r_t + \gamma Q_t^B(s_{t+1}, a^*) - Q_t^*(s_t, a_t) \ ,$$

where $a^* = \arg\max_a Q^A(s_{t+1}, a)$. It is straightforward to show the first two conditions of the lemma hold. The fourth condition of the lemma holds as a consequence of the boundedness condition on the variance of the rewards in the theorem. This, together with the condition that the discount factor is lower than 1, ensures that the Q values are bounded. The theorem can be extended to undiscounted MDPs that have a non-zero probability of eventually terminating for all states and actions, but we do not consider this extension here.

This leaves to show that the third condition on the expected contraction of F_t holds. We can write

$$F_t(s_t, a_t) = F_t^Q(s_t, a_t) + \gamma \left(Q_t^B(s_{t+1}, a^*) - Q_t^A(s_{t+1}, a^*) \right) \ ,$$

where

$$F_t^Q = r_t + \gamma Q_t^A(s_{t+1}, a^*) - Q_t^*(s_t, a_t) \ ,$$

is the value of F_t if normal Q-learning would be under consideration. It is well-known that $E\{F_t^Q|P_t\} \leq \gamma||\Delta_t||$, so to apply the lemma we identify

$$c_t = \gamma E\left\{ Q_t^B(s_{t+1}, a^*) - Q_t^A(s_{t+1}, a^*) \middle| P_t \right\} .$$

We define a new function

$$\Delta_t^{BA}(s,a) = Q_t^B(s,a) - Q_t^A(s,a) .$$

Then to prove convergence of c_t to zero, it is sufficient to show that Δ_t^{BA} converges to zero.

Depending on whether Q^B or Q^A is updated, the update of Δ_t^{BA} at time t is either

$$\begin{aligned}
\Delta_{t+1}^{BA}(s_t,a_t) &= \Delta_t^{BA}(s_t,a_t) + \alpha_t(s_t,a_t) F_t^B(s_t,a_t) \ , \text{ or} \\
\Delta_{t+1}^{BA}(s_t,a_t) &= \Delta_t^{BA}(s_t,a_t) - \alpha_t(s_t,a_t) F_t^A(s_t,a_t) \ ,
\end{aligned}$$

where

$$\begin{aligned}
F_t^A(s_t,a_t) &= r_t + \gamma Q_t^B(s_{t+1}, a^*) - Q_t^A(s_t,a_t) \text{ and} \\
F_t^B(s_t,a_t) &= r_t + \gamma Q_t^A(s_{t+1}, b^*) - Q_t^B(s_t,a_t) .
\end{aligned}$$

Then

$$\begin{aligned}
&E\left\{ \Delta_{t+1}^{BA}(s_t,a_t) \middle| P_t \right\} \\
&= \Delta_t^{BA}(s_t,a_t) + \frac{\alpha_t(s_t,a_t)}{2} E\left\{ F_t^B(s_t,a_t) - F_t^A(s_t,a_t) \middle| P_t \right\} \\
&= (1 - \zeta_t^{BA}(s_t,a_t)) \Delta_t^{BA}(s_t,a_t) + \zeta_t^{BA}(s_t,a_t) E\left\{ F_t^{BA}(s_t,a_t) \middle| P_t \right\} \ ,
\end{aligned}$$

where $\zeta_t^{BA}(s,a) = \alpha_t(s,a)/2$ and

$$E\{F_t^{BA}(s_t,a_t)|P_t\} = \gamma E\left\{ Q_t^A(s_{t+1}, b^*) - Q_t^B(s_{t+1}, a^*) | P_t \right\} .$$

This means we are done if we can show that $\|E\left\{F_t^{BA}\middle|P_t\right\}\| \leq \kappa \|\Delta_t^{BA}\|$ for $\kappa \in [0,1)$. To show this, we will consider two mutually exclusive cases.

Assume $E\{Q_t^A(s_{t+1}, b^*)|P_t\} \geq E\{Q_t^B(s_{t+1}, a^*)|P_t\}$. By definition of a^* as given in line 6 of Algorithm 5 we have $Q_t^A(s_{t+1}, a^*) = \max_a Q_t^A(s_{t+1}, a) \geq Q_t^A(s_{t+1}, b^*)$ and therefore

$$\begin{aligned}
\left| E\{F_t^{BA}(s_t,a_t)|P_t\} \right| &= \gamma E\left\{ Q_t^A(s_{t+1}, b^*) - Q_t^B(s_{t+1}, a^*) | P_t \right\} \\
&\leq \gamma E\left\{ Q_t^A(s_{t+1}, a^*) - Q_t^B(s_{t+1}, a^*) | P_t \right\} \leq \gamma \left\| \Delta_t^{BA} \right\| .
\end{aligned}$$

Now assume $E\{Q_t^B(s_{t+1},a^*)|P_t\} > E\{Q_t^A(s_{t+1},b^*)|P_t\}$ and note that by definition of b^* we have $Q_t^B(s_{t+1},b^*) \geq Q_t^B(s_{t+1},a^*)$. Then

$$\begin{aligned}\left|E\{F_t^{BA}(s_t,a_t)|P_t\}\right| &= \gamma E\left\{Q_t^B(s_{t+1},a^*) - Q_t^A(s_{t+1},b^*)|P_t\right\} \\ &\leq \gamma E\left\{Q_t^B(s_{t+1},b^*) - Q_t^A(s_{t+1},b^*)|P_t\right\} \leq \gamma\left\|\Delta_t^{BA}\right\| \quad .\end{aligned}$$

Clearly, one of the two assumptions must hold at each time step. Since $F_t^{BA}(s,a) = 0$ for all $(s,a) \neq (s_t,a_t)$ we obtain the desired result that $\|E\{F_t^{BA}|P_t\}\| \leq \gamma\|\Delta_t^{BA}\|$. Applying the lemma yields convergence of Δ_t^{BA} to zero, which in turn ensures that the original process also converges in the limit. □

CHAPTER 5

ACTION VALUE ALGORITHMS

In this chapter, we will dive deeper into the subfield of reinforcement learning that considers model-free temporal-difference algorithms. These algorithms have the advantage that they can learn through simulation, without the need of models for the reward function and the transition probabilities. Additionally, the algorithms are computationally efficient and can learn online, which implies that they can be used in non-stationary environments.

In the former chapters, we have discussed Q-learning (Watkins, 1989) and Sarsa (Rummery and Niranjan, 1994), which are two prominent examples of such reinforcement learning algorithms. We also introduced Double Q-learning as an alternative for Q-learning. However, these are by no means the only possible algorithms. In this chapter, we discuss some possible variants.

5.1 Introduction

In Chapter 2, we noted that Q-learning and Sarsa can be thought of as averaging sampled updates of Bellman equations. We repeat those equations here for clarity.

$$Q^\pi(s,a) = E\left\{r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \middle| s_t = s, a_t = a, \pi\right\} \quad , \tag{5.1}$$

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \middle| s_t = s, a_t = a, \pi\right\} \quad . \tag{5.2}$$

Sarsa uses a sampled version of the first update, yielding:

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma Q_t(s_{t+1},a_{t+1})\right) \quad , \tag{5.3}$$

whereas Q-learning uses a sampled version of the second update:

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma \max_a Q_t(s_{t+1},a)\right) \quad . \tag{5.4}$$

The main difference is that Q-learning is off-policy, which means it can learn about the optimal policy while following some exploratory policy. Sarsa is on-policy, which implies that it learns the value of the current policy. For a fixed policy π, the action values updated by Sarsa converge to Q^π, while those updated by Q-learning converge to Q^*, under some conditions on the learning rates, the policy and the MDP. Sarsa converges to Q^* rather than to Q^π if the policy is slowly made greedy during the learning phase (Singh et al., 2000).

Note that Sarsa does not converge if the policy is simply made greedy right away, since then no exploration occurs.

Both Q-learning and Sarsa converge very slowly in some settings. We analyzed why this happens and presented a partial solution in Chapter 4. In this chapter, we discuss some other model-free temporal-difference algorithms for control. We discuss Expected Sarsa in Section 5.3, General Q-learning in Section 5.4, QV-learning in Section 5.5, Actor Critic methods in Section 5.6 and Actor Critic Learning Automata in Section 5.7. These are not all possible or even all known model-free action value algorithms and we will give some pointers to other variants where relevant. In Section 5.8 we show the results for some experiments to give an indication on how the algorithms compare to each other. These experiments also include the Double Q-learning algorithm that was introduced in the former chapter.

5.2 A Different Perspective: Gradients and Norms

Most of the algorithms we will discuss can be interpreted as minimizing quadratic one-step errors through gradient descent (Sutton, 1988). We will use TD learning as an example of how this works. For clarity, we repeat the TD update here:

$$V_{t+1}(s_t) = V_t(s_t) + \beta_t(s_t)\left(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\right) \ , \tag{5.5}$$

where $\beta_t(s_t) \in [0,1]$ is a step size parameter. We use β to differentiate from α which we use as a step size parameter for action values.

Consider a parametrized function $f_t : \mathbb{R}^N \to \mathbb{R}$ that is a linear weighted sum of its input, such that $f_t(\vec{x}) = \vec{w}_t^T \vec{x}$, where $\vec{x} \in \mathbb{R}^N$ is an input vector and $\vec{w}_t \in \mathbb{R}^N$ is a parameter vector. The gradient of a function to its input is defined as follows:

Definition 5.1 (Gradient)**.** *The gradient of a function $f : \mathbb{R}^N \to \mathbb{R}$ at a point $\vec{x} = (x_1, \ldots, x_N)$ is denoted by $\nabla f(\vec{x})$ and is defined as a row vector with N partial derivatives of f for each of the elements of $\vec{x}$, such that*

$$\nabla f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_N}\right) \ .$$

If $\vec{x}$ is a one-dimensional vector, or a scalar rather than a vector, the gradient is equal to the derivative, such that

$$\nabla f(\vec{x}) = f'(x) \ ,$$

where $\vec{x} = (x)$.

The gradient to the parameters is defined equally, with $\vec{w}$ in the place of $\vec{x}$ in the definition.

Suppose we receive a noisy sample for a desired combination of an input $\vec{x}$ and an output $y \in \mathbb{R}$ for this function. We can use gradient descent to update the parameter vector toward this target with the update

$$\vec{w}_{t+1} = \vec{w}_t - \zeta_t \nabla_{\vec{w}_t} \mathbf{E}_t \ , \tag{5.6}$$

where ζ_t is some learning rate and $\mathbf{E}_t = \frac{1}{2}(f(\vec{x}) - y)^2$ is an error. Other error measures than the squared difference are possible too, but a discussion of these falls outside the scope of this section. The gradient effectively gives us the direction of steepest ascent for the error relative to the parameter vector. In other words, for a point $\vec{x}$ in the input space, the gradient tells us in which direction we should move the parameter vector in order to increase the error the fastest, given a first order approximation. Since we want to decrease the error, rather than increase it, we step in the opposite direction: the direction of steepest descent. Hence, the name gradient descent. Since f was assumed to be linear in $\vec{x}$, update (5.6) then reduces to

$$\vec{w}_{t+1} = \vec{w}_t + \zeta_t (y - \vec{w}_t^T \vec{x}) \vec{x} \ , \tag{5.7}$$

an update which is known as the Widrow-Hoff rule (Widrow and Hoff, 1960).

Since in this chapter we consider tables that store the individual values, we can equivalently write the value function as $V(s_t) = \vec{w}_t^T x_t$, where $\vec{w}_t$ is a vector of size $|S|$ containing all the state values and x_t is a vector of size $|S|$ with all elements zero, except the element corresponding to the current state which will be equal to one. It is easily verified that then indeed $\vec{w}_t^T x_t = V(s_t)$. In TD learning, we want to reach the fixed point $V = \mathcal{T} V$, so it makes sense to use the error

$$\mathbf{E} = \frac{1}{2} \| V_t - \mathcal{T} V_t \|_{2,\pi} = \frac{1}{2} \sum_{s \in S} P(s = s_t)(V_t(s) - (\mathcal{T} V_t)(s))^2 \ ,$$

where the norm $\| \cdots \|_{2,\pi}$ denotes a quadratic norm weighted by the steady state probabilities $P(s = s_t | \pi)$ that the agent is in each state, which are dependent on the policy π. This error can be sampled as

$$\mathbf{E}_t = \frac{1}{2} \big(r_t + \gamma V_t(s_{t+1}) - V_t(s_t) \big)^2 = \frac{1}{2} (\delta_t)^2 \ ,$$

which is simply the squared TD error. Using the Widrow-Hoff update with $y = r_t + \gamma V_t(s_{t+1})$ and $\zeta_t = \beta_t(s_t)$ one directly obtains the TD learning update in (5.5).

In the tabular case we consider here, the TD error can in principle be reduced fully to zero and the exact fixed point $V = \mathcal{T} V$ can be obtained in the limit. However, if one uses less parameters than states, for instance when the

state space is large or continuous, this no longer holds in general. Then, one could in principle do one of two things: minimize the normal TD error as far as possible, given the limitation of the function representation that is used, or minimize a different error function. We will discuss these possibilities in Chapter 7.

5.3 Expected Sarsa

The first novel algorithm we discuss is a variant on Sarsa. We noted that Sarsa can be viewed as an averaging sample of Bellman equation (5.1). Update (5.3) samples the reward and the transition that actually occur as well as the action that is chosen in the next state. This sets the algorithm apart from Q-learning that uses the maximal value in this next state for its update.

Closer inspection shows the sampling of the policy is in fact unnecessary. The reason we sample the reward and transition is because we do not want to assume or store a model for the reward and transition function. In any case, a known policy is required since it is used to select actions. In other words, the policy π_t at time t should always be known. This means there is no need to sample the action that is actually taken and we can instead use the following update:

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma \sum_a \pi_t(s_{t+1},a) Q_t(s_{t+1},a)\right) . \quad (5.8)$$

The first mention of this algorithm seems to be in an exercise in the book by Sutton and Barto (1998). It was more recently studied in detail and found in general to be an improvement over Sarsa (van Seijen et al., 2009). It can also be interpreted as a special case of the General Q-learning algorithm that we will discuss in Section 5.4. We will discuss the connection between Expected Sarsa, General Q-learning and Sarsa in more detail when we discuss General Q-learning.

We will prove that in general Expected Sarsa has the same bias, but lower variance than the original Sarsa algorithm. As such, it is a superior algorithm, although not in all cases by a large margin. Lower variance means that in practice α can often be increased in order to speed up the learning process. This has empirically been demonstrated (van Seijen et al., 2009). When the environment is deterministic, Q-learning and Expected Sarsa can use $\alpha = 1$, while Sarsa still requires $\alpha < 1$ to average over stochasticity in the policy.

5.3.1 Variance Analysis

In this section, we analyze the difference between Sarsa and Expected Sarsa. We show under which conditions Expected Sarsa will perform better than Sarsa. Specifically, we show that both algorithms have the same bias and that the variance of Expected-Sarsa is lower. Finally, we describe which factors affect this difference in variance. In this section, we use $v_t^E = \sum_a \pi_t(s_{t+1},a)Q_t(s_{t+1},a)$ and $v_t^S = Q_t(s_{t+1},a_{t+1})$ to denote the values used for the updates of Expected Sarsa and Sarsa, respectively. Further, we use the shorthands $E_t\{\cdot\} = E\{\cdot|(s,a) = (s_t,a_t)\}$, $\text{Bias}_t(\cdot) = \text{Bias}(\cdot|(s,a) = (s_t,a_t))$ and $\text{Var}_t(\cdot) = \text{Var}(\cdot|(s,a) = (s_t,a_t))$ for the expected value, bias and variance on time t.

The bias of the targets for the updates of both algorithms compared to the value of any policy π is given by

$$\text{Bias}_t(v_t) = E_t\{Q^\pi(s_{t+1},a_{t+1}) - v_t\} \ ,$$

where v_t is either v_t^E or v_t^S. Regardless of π, both algorithms have the same bias, since $E_t\{v_t^E\} = E_t\{v_t^S\}$.

The variance of the targets is given by

$$\text{Var}_t(v_t) = E_t\{(v_t)^2\} - (E_t\{v_t\})^2 \ . \tag{5.9}$$

The squared expected target in (5.9) is the same for both Sarsa and Expected Sarsa. Therefore the difference between the two variances is equal to the difference between expected squared targets: $E_t\{(v_t^S)^2\} - E_t\{(v_t^E)^2\}$. For Sarsa, the expected squared target is

$$\begin{aligned} E_t\{(v_t^S)^2\} &= E_t\left\{(Q_t(s_{t+1},a_{t+1}))^2\right\} \\ &= \sum_s P^s_{s_t a_t}\left(\sum_a \pi_t(s,a)(Q_t(s,a))^2\right) \ . \end{aligned}$$

Similarly, for Expected Sarsa we get

$$\begin{aligned} E_t\{(v_t^E)^2\} &= E_t\left\{\left(\sum_a \pi_t(s_{t+1},a)Q_t(s_{t+1},a)\right)^2\right\} \\ &= \sum_s P^s_{s_t a_t}\left(\sum_a \pi_t(s,a)Q_t(s,a)\right)^2 \ . \end{aligned}$$

Therefore, the difference in variances is equal to

$$\begin{aligned} &\text{Var}_t(v_t^S) - \text{Var}_t(v_t^E) = \\ &\qquad \sum_s P^s_{s_t a_t}\left(\sum_a \pi_t(s,a)(Q_t(s,a))^2 - \left(\sum_a \pi_t(s,a)Q_t(s,a)\right)^2\right) \ . \end{aligned}$$

The inner term of the differences in variance between Sarsa and Expected Sarsa is of the form

$$\sum_i^n w_i x_i^2 - (\sum_i^n w_i x_i)^2 \quad , \tag{5.10}$$

where the $\vec{w}$ corresponds to the policy and $\vec{x}$ corresponds to the action values. When $w_i \geq 0$ for all i and $\sum_i^n w_i = 1$, an unbiased estimate of the variance of the weighed values $w_i x_i$ is equal to

$$\frac{\sum_i^n w_i (x_i - \mu_w)^2}{1 - \sum_i^n w_i^2} \quad ,$$

where $\mu_w = \sum_i^n w_i x_i$ is the weighted average. This term is uniformly positive. We rewrite the numerator of this fraction to get

$$\begin{aligned} \sum_i^n w_i (x_i - \mu_w)^2 &= \sum_i^n w_i x_i^2 - 2\mu_w \sum_i^n w_i x_i + \mu_w^2 \sum_i^n w_i \\ &= \sum_i^n w_i x_i^2 - 2\mu_w^2 + \mu_w^2 \\ &= \sum_i^n w_i x_i^2 - \mu_w^2 \quad , \end{aligned}$$

which is exactly the same quantity as given in (5.10). This implies that the difference in variance is closely related to the weighted variance of the action values. In other words, a larger weighted variance of the action values, weighted according to the current policy, results in a larger difference in variance between Sarsa and Expected Sarsa. This means that Expected Sarsa has an expected improved performance in problems with high weighted variance in the action values.

More concretely, our analysis shows that we can expect Expected-Sarsa to perform better than Sarsa especially in problems in which the action values are separated far from each other and in which much exploration is used. This makes intuitive sense, since then the stochastic sample $Q_t(s_{t+1}, a_{t+1})$ that Sarsa uses is likely to contain more noise than the sample $\sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a)$ that Expected-Sarsa uses.

5.4 General Q-learning

Expected Sarsa was coined as an on-policy algorithm with the intention of using it as an improved replacement for Sarsa. However, the update is more flexible than Sarsa since we are not restricted to use the behavior policy π_t for our update, but rather, we can specify another estimation policy π_t^e of which we want to learn the value. We will refer to this algorithm as General

Q-learning and its update is

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma \sum_a \pi_t^e(s_{t+1},a)Q_t(s_{t+1},a)\right) . \quad (5.11)$$

This algorithm was proposed in earlier work by Precup et al. (2000), where it was called the tree backup algorithm. Q-learning, Sarsa and Expected Sarsa can be seen as special instances of General Q-learning. If the estimation policy π_t^e is equal to some fixed policy π that is also used as the behavior policy, the algorithm reduces to on-policy Expected Sarsa and can be shown to converge to Q^π. If $\pi_t^e \neq \pi_t$ the algorithm is called off-policy.

In previous work, the General Q-learning algorithm was shown to converge to Q^e for any fixed estimation policy π^e, where Q^e is the value of this policy (Precup et al., 2000; Precup and Sutton, 2001). The proof of convergence was also extended to include eligibility traces. The only restriction is that the behavior policy is non-starving, which is similar to requiring infinite exploration in an ergodic MDP. Both statements imply that every state-action pair is visited an infinite number of times almost surely in the limit.

We will prove that General Q-learning converges to the optimal policy for any combination of behavior and estimation policies as long as the behavior policy results in infinite exploration and the estimation policy is greedy in the limit. Although this can be shown with a relatively straightforward application of Lemma 2.1, we do not know of earlier work that discusses this result. Furthermore, we show that General Q-learning is a strict improvement over per-decision importance sampling, a similar algorithm for general off-policy reinforcement learning that was proposed concurrently (Precup et al., 2000). This confirms the experimental results in that work and the resulting analysis that General Q-learning is to be preferred over per-decision importance sampling. Similar to the analysis for Expected Sarsa, we can show that both algorithms have the same bias, but General Q-learning has a lower variance.

5.4.1 Convergence

The idea is to apply Lemma 2.1 with $X = S \times A$, $P_t = \{Q_0, s_0, a_0, \alpha_0, r_1, s_1, a_1, \ldots, s_t, a_t\}$, $v_t = (s_t, a_t)$, $\zeta_t(v_t) = \alpha_t(s_t,a_t)$ and $\Delta_t(v_t) = Q_t(s_t,a_t) - Q^*(s_t,a_t)$. If we can prove that Δ_t converges to zero with probability one, we have convergence of the Q values to the optimal values.

Theorem 5.1. *[Convergence of General Q-learning] General Q-learning as defined by update* (5.11) *converges to the optimal value function whenever the following assumptions hold:*

1. *S and A are finite,*

2. $\alpha_t(s_t,a_t) \in [0,1]$, $\sum_t \alpha_t(s_t,a_t) = \infty$, $\sum_t (\alpha_t(s_t,a_t))^2 < \infty$ *w.p.1 and* $\forall (s,a) \neq (s_t,a_t) : \alpha_t(s,a) = 0$,

3. *The estimation policy* π_t^e *is greedy in the limit and the behavior policy* π_t *ensures infinite exploration,*

4. $Var\{r_{t+1}|P_t\} < \infty$.

The proof of the theorem is shown in Section 5.10.1. Theorem 5.1 shows that General Q-learning converges to the optimal policy under similar conditions as Sarsa and Q-learning. The difference with Q-learning is the added restriction that the estimation policy must become greedy, whereas Q-learning has no explicit notion of an estimation policy. The difference with Sarsa is that the behavior policy need not become greedy and the estimation policy need not ensure infinite exploration, since the estimation and behavior policies are separate.

Many algorithms are a special case of General Q-learning. We could use the greedy policy as the estimation policy. Trivially, an estimation policy that is greedy everywhere is also greedy in the limit. Then, General Q-learning reduces to Q-learning and converges to Q^*. If we let π_t^e be a stochastic policy such that $\pi_t^e(s,a) = 1$ with probability $\pi_t(s,a)$, General Q-learning reduces to Sarsa. Therefore, Theorem 5.1 implies the convergence of Q-learning and of Sarsa and Expected Sarsa for behavior policies that are greedy in the limit.

5.4.2 Eligibility Traces

Like Q-learning and Sarsa, General Q-learning can be extended with eligibility traces. We want the algorithm to approximate Q^e for any fixed π^e and any λ. This implies that we want the summed TD errors to telescope when $\lambda = 1$ is used, such that the resulting algorithm reduces to a Monte Carlo method. Therefore, we propose to use a variant similar to the Q(λ) algorithm proposed by Peng and Williams (1996). The General Q(λ) algorithm is shown in Algorithm 6, where we have applied the delayed eligibility trace updates from Wiering and Schmidhuber (1998).

5.4.3 Related Work

A similar algorithm is the per-decision importance sampling (PDIS) algorithm by Precup et al. (2000). The TD error used by this algorithm is

$$\delta_t = r_{t+1} + \gamma \frac{\pi_t^e(s_{t+1},a_{t+1})}{\pi_t(s_{t+1},a_{t+1})} Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t) \ . \tag{5.12}$$

It is straightforward to show that this TD error has the same expected value as the TD error used by General Q-learning with a behavior policy π_t and

Algorithm 6 Fast General Q(λ)

Given γ, π, π^e, I and an MDP to act on.
Initialize Q, $s \sim I$, $D = 0$, $\phi = 1$, $\forall s : e(s) = 0$.
repeat
  if s is terminal **or** $\lambda\gamma\phi < \epsilon_M$ **then**
    for all $s \in S$, $a \in A(s)$ **do**
      $Q(s,a) = Q(s,a) + \alpha(s,a)e(s,a)\big(D - d(s,a)\big)$
      $d(s,a) = 0$
      $e(s,a) = 0$
    end for
    $\phi = 1$
    $D = 0$
    $s \sim I$
  else
    Choose $a \sim \pi(s)$
    Preform a, observe r and s'
    $V' = \sum_{a' \in A(s')} \pi^e(s',a') \big[Q(s',a') + \alpha(s')e(s')\big(D - d(s',a')\big)\big]$
    $\delta = r + \gamma V - Q(s,a)$
    $D = D + \phi\delta$
    $e(s,a) = 1/\phi$
    $\phi = \lambda\gamma\phi\,\pi^e(s,a)/\pi(s,a)$
    $s = s', a = a'$
    for all $a \in A(s)$ **do**
      $Q(s,a) = Q(s,a) + \alpha(s)e(s)\big(D - d(s)\big)$
      $d(s,a) = D$
    end for
    Determine π^e, π (if these are not fixed)
  end if
until sufficient convergence

an estimation policy π^e_t. In the next subsection, we show that the variance of General Q-learning is lower while both algorithms share the same bias. Therefore, General Q-learning can also be viewed as an improved version of PDIS. This is in line with the experiments in Precup et al. (2000) that show that General Q-learning has better performance than PDIS. As a special case, if $\pi^e_t = \pi_t$ the TD error in (5.12) reduces to the same TD error as used by Sarsa.

Another similar algorithm is the very recent GQ algorithm (Maei and Sutton, 2010; Maei et al., 2010). GQ uses the TD error

$$\delta_t = r_{t+1} + \beta_{t+1} z_t + (1 - \beta_{t+1}) \sum_a \pi^e_t(s_{t+1}, a) Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \ , \qquad (5.13)$$

where β_t is the probability that an option terminates (Sutton et al., 1998). We will not explain options in any detail here, but the probability that the option

does not terminate is a parameter comparable to the discount factor, such that we can take $\gamma = (1 - \beta_t)$, for all t. If we further ignore the target outcome function z_t which is related to the options framework, such that $z_t = 0$ for all t, the GQ algorithm reduces to General Q-learning. GQ was extended to arbitrary smooth function approximation (Maei et al., 2010) and linear function approximation in combination with eligibility traces (Maei and Sutton, 2010). It seems the step to arbitrary function approximation in combination with eligibility traces has not yet been made, possibly because the focus was mainly on convergence guarantees and these are harder to give for that case.

The similarities between GQ and General Q-learning imply that the convergence guarantees that are given for GQ also hold for General Q-learning. This implies that General Q-learning converges to the value Q^e for any fixed estimation policy π^e not only in the single-step tabular case outlined above, but also when eligibility traces and linear function approximation are used. Also, General Q-learning can be combined with arbitrary smooth function approximation as indicated in Maei et al. (2010) without diverging. However, we note that this result does not guarantee convergence of Q_t to Q^* or even to a small region around Q^*. The guaranteed convergence of an algorithm to a region around Q^* on any MDP when non-linear function approximation is used is still largely open, due to the inherent possibility of reaching local optima. We will discuss function approximation in reinforcement learning in more detail in Chapter 7.

Finally, we note that although General Q-learning converges to Q^* when suitable estimation policies are used and General Q-learning converges to Q^e for any fixed π^e when eligibility traces are used, we know of no proofs of convergence for any eligibility trace method to Q^* for general λ.

5.4.4 Variance Analysis

In this section, we show under which conditions General Q-learning will perform better than PDIS. Specifically, we show that both algorithms have the same bias and that the variance of General Q-learning is lower. We describe which factors affect this difference in variance.

In this section, we use the notation

$$v_t^{GQ} = \sum_a \pi_t^e(s_{t+1}, a) Q_t(s_{t+1}, a) \ ,$$

for the value of the next state used by General Q-learning and

$$v_t^{PDIS} = \frac{\pi_t^e(s_{t+1}, a_{t+1})}{\pi_t(s_{t+1}, a_{t+1})} Q_t(s_{t+1}, a_{t+1}) \ ,$$

for the value used by PDIS. Again, P_t is the past up to time t as defined in Section 5.4.1. For simplicity, we will assume π^e is fixed, but the argument below holds with small adaptations for time-dependent estimation policies.

Both algorithms have the same bias, since

$$\begin{aligned} E\left\{ v_t^{PDIS} \middle| P_t \right\} &= E\left\{ \frac{\pi_t^e(s_{t+1},a_{t+1})}{\pi_t(s_{t+1},a_{t+1})} Q_t(s_{t+1},a_{t+1}) \middle| P_t \right\} \\ &= \sum_s P^s_{s_t a_t} \left(\sum_a \pi_t(s,a) \frac{\pi_t^e(s,a)}{\pi_t(s,a)} Q_t(s,a) \right) \\ &= \sum_s P^s_{s_t a_t} \sum_a \pi_t^e(s,a) Q_t(s,a) \\ &= E\left\{ v_t^{GQ} \middle| P_t \right\} \quad , \end{aligned}$$

where we assume that the action that is selected in state s_{t+1} is selected with the behavior policy as it was on time t. There may be a subtle difference if the behavior policy is not fixed and therefore it is possible that $\pi_{t+1}(s_{t+1},\cdot)$ is not equal to $\pi_t(s_{t+1},\cdot)$. In this case the update by General Q-learning is to be preferred, since by design it implicitly uses the actual behavior policy at time $t+1$. For simplicity, in the rest of this analysis we assume the behavior policy does not change, where again the argument below continues to hold with small adaptations if we would relax this requirement.

Similar to the analysis for Expected Sarsa, the variance is given by (5.9). Since $E\{v_t^{PDIS}|P_t\} = E\{v_t^{GQ}|P_t\}$, the last term in (5.9) is equal for PDIS and General Q-learning and therefore the difference between the two variances is equal to the differences between the first terms:

$$\mathrm{Var}\left(v_t^{PDIS} \middle| P_t \right) - \mathrm{Var}\left(v_t^{GQ} \middle| P_t \right) = E\left\{ (v_t^{PDIS})^2 \middle| P_t \right\} - E\left\{ (v_t^{GQ})^2 \middle| P_t \right\} \quad .$$

For PDIS, this term is equal to

$$\begin{aligned} E\left\{ (v_t^{PDIS})^2 \middle| P_t \right\} &= E\left\{ \left(\frac{\pi_t^e(s,a)}{\pi_t(s,a)} Q_t(s_{t+1},a_{t+1}) \right)^2 \middle| P_t \right\} \\ &= \sum_s P^s_{s_t a_t} \left(\sum_a \frac{(\pi_t^e(s,a))^2}{\pi_t(s,a)} (Q_t(s,a))^2 \right) \quad . \end{aligned}$$

Similarly, for General Q-learning we get

$$\begin{aligned} E\left\{ (v_t^{GQ})^2 \middle| P_t \right\} &= E\left\{ \left(\sum_a \pi_t^e(s_{t+1},a) Q_t(s_{t+1},a) \right)^2 \middle| P_t \right\} \\ &= \sum_s P^s_{s_t a_t} \left(\sum_a \pi_t^e(s,a) Q_t(s,a) \right)^2 \quad . \end{aligned}$$

Therefore, the difference in variance is equal to

$$\begin{aligned} &\mathrm{Var}(v_t^{PDIS}) - \mathrm{Var}(v_t^{GQ}) = \\ &\qquad \sum_s P^s_{s_t a_t} \left(\sum_a \frac{(\pi_t^e(s,a))^2}{\pi_t(s,a)} (Q_t(s,a))^2 - \left(\sum_a \pi_t^e(s,a) Q_t(s,a) \right)^2 \right) \quad . \end{aligned}$$

The inner term is of the form

$$\sum_i^n \frac{1}{w_i}(w_i^e x_i)^2 - (\sum_i^n w_i^e x_i)^2 \quad , \tag{5.14}$$

where the $\vec{w}$, $\vec{w}^e$ and $\vec{x}$ correspond to the π, π^e and the action values, respectively. Note that $w_i \geq 0$, $w_i^e \geq 0$ for all i and $\sum_i^n w_i = \sum_i^n w_i^e = 1$.

If we minimize the first term, subject to the constraints $\sum_i^n w_i = 1$ and $\forall j : w_j \in [0,1]$, we find that it is minimal if the following holds for all j:

$$w_j = \frac{|w_j^e x_j|}{\sum_i^n |w_i^e x_i|} \quad . \tag{5.15}$$

If we resubstitute this in (5.14), we get zero if and only if all action values have the same sign. In other words, the quantity in (5.14) is non-negative and therefore General Q-learning can not have a larger variance than per-decision importance sampling. Of course, this does not come as a surprise, since the value used by General Q-learning is the expected value for the one used by per-decision importance sampling. If equation (5.15) does not hold or when not all action values have the same sign, General Q-learning will have a strictly lower variance. Even when we assume that all action values are positive, equation (5.15) implies that in order to have the same variance we should choose the behavior policy for PDIS as $\pi_t(s,a) = \pi_t^e(s,a)Q_t(s,a)/(\sum_b \pi_t^e(s,b)Q_t(s,b))$. Then, the update for PDIS reduces precisely to that of General Q-learning, but at the cost of a very specific behavior policy, whereas General Q-learning can choose its behavior policy freely.

We can compare this difference in variance to the difference in variance between Sarsa and Expected Sarsa. That difference has the form given in (5.10) and we showed it is related to the weighted variance of the action values. There are two differences between (5.10) and (5.14). The first difference is that for Expected Sarsa the weights for the squared action values in the first term are equal to the estimation policy—which is equal to the behavior policy—while the weights are equal to the estimation policy times $\pi_t^e(s,a)/\pi_t(s,a)$ for General Q-learning. The second difference is that in practice the estimation policy itself will often be different in the two cases. For Sarsa and Expected Sarsa the estimation policy will usually contain exploration, since it is equal to the behavior policy. For General Q-learning the estimation policy will more often be deterministic, for instance because it is the greedy policy.

Suppose for a moment that indeed the estimation policy is deterministic. We can reflect this in (5.14) by setting w_i^e to zero for all elements except one. Then (5.14) reduces to

$$\frac{1}{w_i}(x_i)^2 - (x_i)^2 \quad , \tag{5.16}$$

where i corresponds to the deterministically chosen element according to the estimation weights w_i^e. For instance, this may be the greedy element. Note that w_i is equal to one only when the behavior policy is greedy, which will usually not be the case as it would defeat the purpose of using an off-policy update procedure. Since $w_i \leq 1$ the quantity in (5.16) is clearly non-negative, and will be large especially when the probability that the behavior policy selects the deterministic choice of the estimation policy is small. This is especially the case in settings with much exploration and when there are many actions to choose from.

This analysis indicates that General Q-learning has an expected improved performance because of a lower variance in its updates, especially in problems with high weighted variance in the action values. In practice, this means we expect General Q-learning to perform better than PDIS especially in problems in which the action values are separated far from each other and in which much exploration is used. Of course, this makes intuitive sense, since then the stochastic sample that PDIS uses is likely to contain more noise than the sample that General Q-learning uses.

5.5 QV-learning

Another on-policy algorithm is QV-learning. It too can be considered a variation on Sarsa, although it differs more from Sarsa in its update rule than Expected Sarsa does. Equivalent to equation (5.1), the function Q^π can be defined as follows:

$$Q^\pi(s,a) = E\left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) \middle| s_t = s, a_t = a, \pi \right\} \quad , \tag{5.17}$$

since by definition $V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s,a)$. For Expected Sarsa, we used the sum $\sum_a \pi(s,a) Q_t(s,a)$ as an approximation for this value. However, we can also directly approximate the value of V^π with the TD learning update (5.5), which ensure that V_t converges to V^π in the limit as t goes to infinity. Then, using V_t as an increasing good approximation for V^π, we can sample (5.17) to obtain:

$$Q_{t+1}(s_t,a_t) = (1 - \alpha_t(s_t,a_t)) Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma V_t(s_{t+1})\right) \quad . \tag{5.18}$$

This algorithm is called QV-learning (Wiering, 2005; Wiering and van Hasselt, 2007). A potential advantage compared to Q-learning and Sarsa is that the state space is smaller than the combined state-action space and therefore, the approximation of the state value might improve faster than the approximation of the action values. Similar to Expected Sarsa, QV-learning has the advantage of low variance in its updates. Indeed, QV-learning has been shown to sometimes outperform Q-learning and Sarsa (Wiering, 2005; Wiering and van Hasselt, 2007, 2008, 2009). Additionally, QV-learning has been

used to select appropriate interaction behaviors in a human-robot interaction task with children with autism spectrum disorder (Liu et al., 2008; Conn et al., 2008).

Other variations of QV-learning can be constructed. For instance, we note that $V^*(s) = \max_a Q^*(s,a)$. Then, we can update V_t as an approximation of a one-step on-policy update that picks the maximal action afterward. The resulting algorithm is not fully off-policy since it uses one step of the current policy, but it is also not fully on-policy because of the use of the maximal action value. The action value is then updated as equation (5.18), but the state value is updated as

$$V_{t+1}(s_t) = (1-\beta_t(s_t))V_t(s_t) + \beta_t(s_t)\left(r_{t+1} + \gamma \max_a Q_t(s_{t+1},a)\right) \quad .$$

This algorithm and other variations on QV-learning have been investigated in Wiering and van Hasselt (2009), but we will not consider them further in this dissertation.

5.5.1 Empirical Comparison

In previous work it was shown that in the control case, when policies are typically non-stationary, QV-learning sometimes performs better and sometimes worse than both Q-learning and Sarsa, when these algorithms are all optimized in their parameters (Wiering, 2005; Wiering and van Hasselt, 2007, 2008). As of yet, no clear indication was given in which type of control setting one algorithm should be preferred over another. In this section we will show an example of an MDPs where the explicit use of state values yields an advantage. Additionally, an example is given where it is a disadvantage to use state values and Q-learning is to be preferred. On purpose, these examples are kept simple to allow for a good understanding of what is happening for each of the algorithms. However, the findings tell us something about the behavior of these algorithms in more general problems.

Figure 5.1 depicts a simple MDP with only four states: X, Y, Z and the terminal state T. The task is episodic; X is the starting state and the terminal state T is reached after two steps in each episode. The state transitions are deterministic. In state X there are two actions: one action leads to state Y and one action leads to state Z. Both actions yield zero reward. From state Y, there is only one action which leads to state T and yields a reward of +10. From state Z there can be multiple actions, each of which leads to state T. One such action is depicted in the figure. Two of the potential extra actions are also shown, with dotted lines. The reward for each of the actions in Z is +100 or −100 with equal probability. This is equivalently depicted as a stochastic state transition with deterministic reward in Figure 5.1, where each action from Z transitions to T with a reward of 100 or with a reward of −100 with equal probability.

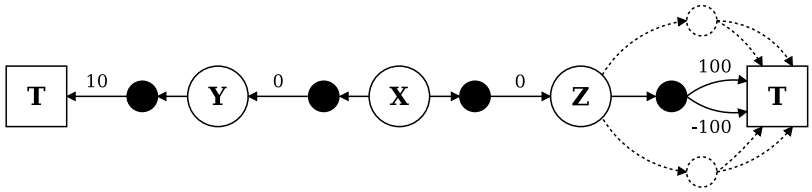

Figure 5.1: The four state MDP.

Similar to $n_t(s)$ that denotes the number of times state s has been visited, $n_t(s,a)$ denotes the number of times action a is chosen in state s in the first t time steps. The learning rates are $\alpha_t(s_t,a_t) = 1/n_t(s_t,a_t)$ and $\beta_t(s_t) = 1/n_t(s_t)$ and the used exploration is ϵ-greedy, where a random action is chosen with probability $\epsilon_t(s_t) = 1/n_t(s_t)$. This exploration is greedy in the limit with infinite exploration. In all cases all state and state-action values are initialized to zero. The discount factor was set to 0.99.

The idea behind this MDP is that the value of state Z is independent on the action that is chosen there. Algorithms that only use action values are then expected to take longer to learn good policies than QV-learning. Especially Q-learning will be optimistic about the value of going to state Z, because of overestimated actions. This issue was examined in detail in Chapter 4. It is expected that QV-learning will learn more quickly that the value of state Z under any policy is in fact zero and will then choose to go to state Y instead.

The first row in Figure 5.2 shows the average rewards for each of the algorithms on this MDP. The hypothesized behavior is precisely what we observe in the results. The left plot shows the results for when there is only one action from state Z to state T. The middle and right plots show the results for when there are 4 and 16 actions, respectively. Recall that all actions in state Z have the same expected rewards and therefore their number does not change the optimal solution for the MDP, which is to go from state X to state Y.

The first row of Figure 5.2 shows that the average rewards for QV-learning do not change if more identical actions are added. This is to be expected in this setting, since the number of actions has no influence on the state values. However, for Expected Sarsa and especially for Q-learning, the number of actions makes a large difference. The reason for this is that if there are more actions, there is a higher probability that one of these actions will have a value larger than 10. This leads Q-learning to believe that state Z has a higher value than state Y and therefore it chooses the wrong action.

As far as Sarsa and Expected Sarsa is concerned: these algorithms perform better than Q-learning, since they do not only consider the greedy action. These algorithms are therefore less optimistic about the value of state Z because of their on-policy updates. However, because the policy becomes greedy, the value of whatever happens to be the greedy action quickly has a

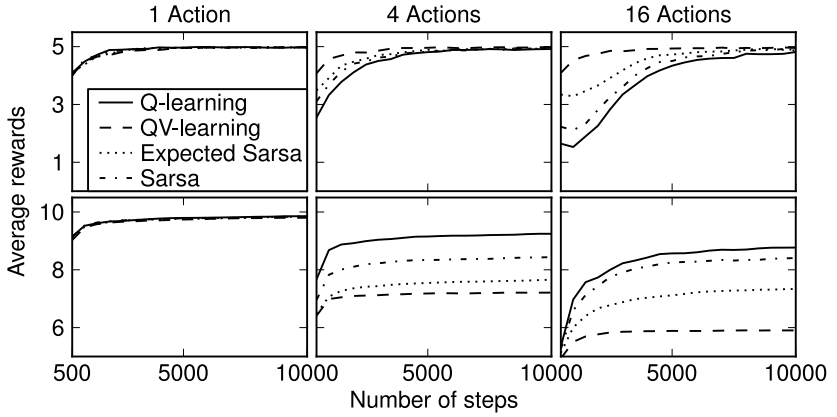

Figure 5.2: The results for the four state MDP shown in Figure 5.1. The first row shows the results when all actions in Z yield rewards of 100 and -100 with equal probability. The second row shows the results when one of these actions is replaced with an action that deterministically yields 20. The left, center and right plots correspond to the MDPs with 1, 4 and 16 actions in state Z, respectively. All lines are averages over 1000 experiments.

big influence on the value of the action that leads to state Z. Therefore, they too suffer from the same problem as Q-learning. Expected Sarsa performs slightly better on this problem than Sarsa.

The use of state values can also be a disadvantage. We demonstrate this by using the same MDP as before, shown in Figure 5.1, but this time we ensure that there is always exactly one action in state Z that yields a deterministic reward of 20. The results on this adapted MDP are shown in the second row in Figure 5.2. Interestingly, the results are reversed in comparison to the first MDP. This time, Q-learning and Sarsa perform better than Expected Sarsa and QV-learning, which performs worst. The reason that QV-learning performs poorly is that the state values resulting from the initially random exploration policy now give a poor indication of the optimal action value in state Z. When there are many actions with lower values, as is the case here, QV-learning will prefer to go to state Y. Since the exploration quickly decreases, QV-learning does not always update the value of Z sufficiently later on to learn that this state is to be preferred.

The learning rates and the exploration scheme in the experiments made it difficult for algorithms to recover from poor initial estimates. In practice, it is often better to choose a fixed learning rate or a learning rate that decreases more slowly. Additionally, although the used exploration rate ensures infinite exploration in the limit, in practice it decreases so quickly that often some of the 16 actions in state Z are not tried even once in the first 10,000 time

steps. The results thus far have shown that there is no single algorithm that is superior to all other algorithms in terms of convergence towards a good control policy.

5.6 Actor Critic

Actor critic algorithms store an indication on how good the current policy is in a critic and a policy in a separate actor (Barto et al., 1983; Sutton, 1984; Sutton and Barto, 1998; Konda and Borkar, 1999; Konda and Tsitsiklis, 2003). As such QV-learning can be viewed as an actor critic algorithm. It stores a critic that determines how good the current policy is through V_t and then uses this critic to influence a policy through Q_t, where we assume that the policy is indeed chosen with use of the current action values. In a broad sense, many algorithms fall into this category. However, in this dissertation, we will use the term actor critic mostly to refer to a specific instance of these algorithms. This instance approximates the state value by use of (5.5) and then updates preference values as follows:

$$P_{t+1}(s_t,a_t) = P_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\right) \ . \tag{5.19}$$

The actor can be based on these preference values. We call these action values preference values to contrast this approach with the action values that approximate the discounted cumulative future reward of a policy—be it the current or the optimal one. In contrast, the preference values do not have such an interpretation. Rather, one can interpret the preference values as the difference between the value of the state and the value of each action in that state. The action that has the highest preference value compared to the state value should correspond to the action that has the highest action value. Therefore, one would expect actor critic algorithms to converge to similar policies as the other on-policy algorithms we have discussed.

If the policy is mostly greedy and the increase is not a random perturbation, the state values will adapt and over time learn the new, larger value. Under some conditions on the policy and the learning rates, actor critic algorithms can be shown to converge to the optimal values V^* (Konda and Borkar, 1999; van Hasselt and Wiering, 2007b). When this is the case, the preference value of the optimal action will converge to zero, while all non-optimal actions will have negative preferences. For a stationary policy, the preference values converge to the expected temporal-difference error of the state values, conditioned on the corresponding action. In the limit then

$$\begin{aligned}\lim_{t\to\infty}\sum_a \pi(s,a)P_t(s,a) &= \lim_{t\to\infty}\sum_a \pi(s,a)E\left\{\delta_t \middle| s_t = s, a_t = a\right\} \\ &= \lim_{t\to\infty} E\left\{\delta_t \middle| s_t = s\right\} = 0 \ ,\end{aligned}$$

where the latter equality holds because of the eventual convergence of the state values.

5.7 Actor Critic Learning Automata

A recent variant on the actor critic algorithm above has been shown to perform well in some problems (Wiering and van Hasselt, 2007; van Hasselt and Wiering, 2009). This algorithm works on the same principle as other actor critic algorithms. A state value is stored, which is updated with the TD learning update (5.5). When an action is performed, the state value will be updated. If the update is positive, apparently the action was a good choice and it makes sense to increase the probability of selecting the action. If the state value was negative, the action was probably not such a good choice and its probability should be decreased. However, instead of using the size of the TD error, which is very problem dependent, one could also only use its direction. A possible update then is

$$P_{t+1}(s_t,a_t) = P_t(s_t,a_t) + \alpha_t(s_t,a_t)(1 - P_t(s_t,a_t)) \qquad \text{if } \delta_t > 0, \tag{5.20}$$
$$P_{t+1}(s_t,a_t) = P_t(s_t,a_t) + \alpha_t(s_t,a_t)(0 - P_t(s_t,a_t)) \qquad \text{if } \delta_t \leq 0. \tag{5.21}$$

Note that the choice of zero and one is arbitrary; we could have chosen any other pair of values so long as the positive TD error corresponds to a higher value than the negative TD error.

The algorithm defined by equations (5.20) and (5.21) is called the actor critic learning automaton (Acla) algorithm. It uses a separate critic for the state values and the actor update resembles a learning automaton update (Narendra and Thathachar, 1974, 1989) of the linear reward-inaction (L_{R-I}) type, which updates an action dependent policy as follows:

$$\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha_t(a_t)(1 - \pi_t(a_t)) \qquad \text{if } r = 1,$$
$$\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha_t(a_t)(0 - \pi_t(a_t)) \qquad \text{if } r = 0.$$

Although the updates are similar, there are many important differences. The L_{R-I} update is stateless, which can equivalently be interpreted as a learning algorithm for an MDP with only a single state. Furthermore, the update uses a Boolean reward, which we mimic in Acla by checking whether or not the TD error is positive. Finally, the learning automaton directly updates the action selection probabilities. Although the preference values of Acla are used to decide which action to take, they are not directly interpreted as action selection probabilities. Rather, for instance we can use Boltzmann exploration to construct a policy.

5.7.1 Acla as an L_1 Gradient Descent Update

Differing from these other algorithms, the Acla algorithm can be thought of as following the gradient of the L_1 norm instead of the L_2 norm of the TD error $\|\delta_t\|_1 = |\delta_t|$. Following the discussion in Section 5.2, the preference values can then be updated with a gradient descent update as

$$P_t(s_t,a_t) = P_t(s_t,a_t) - \alpha_t(s_t,a_t)\frac{\partial|\delta_t|}{\partial s_t}$$

The gradient of this norm is equal to minus one if the TD error is positive, and it is equal to one if the TD error is negative. Therefore, we obtain the following algorithm:

$$\begin{aligned} P_{t+1}(s_t,a_t) &= P_t(s_t,a_t) + \alpha_t(s_t,a_t) && \text{if } \delta_t > 0, \\ P_{t+1}(s_t,a_t) &= P_t(s_t,a_t) - \alpha_t(s_t,a_t) && \text{if } \delta_t < 0. \end{aligned}$$

Although interesting in its own right, this algorithm has the problem that the resulting preference values depend highly on the choice of learning rate and if and how these learning rates are decreased over time. In theory the preference values are unbounded, which is undesirable since it may cause numerical problems. Although this variant may have its own merits, we have not researched it further and will only discuss the version given by equations (5.20) and (5.21). These algorithms are equivalent under a suitable adaption of the learning rates such that the preference values always fall in a predefined range.

5.7.2 Convergence

Unfortunately, Acla does not necessarily converge to the optimal policy in stochastic settings. The reason is that Acla by its very nature increases the probability of actions with a positive TD error and decreases the probability of actions with a negative TD error. Although similar to algorithms that improve the action value and therefore the probability of selecting an action when the TD error is positive, these two objectives are not equivalent. To see why, consider the following example.

Consider an MDP with a single state and two actions. We assume an episode ends immediately after selecting either of these actions, so we only have to consider the immediate reward. These rewards are stochastic such that when action a_1 is chosen the agent receives a reward of 100 with probability 0.1 and a reward of -10 with probability 0.9. Conversely, when a_2 is chosen, the reward is 10 with probability 0.9 and -100 with probability 0.1. The expected values are thus $+1$ for a_1 and -1 for a_2, making a_1 the optimal action. However, now assume that the preference values of Acla are initialized at 0.5 and let $P_t(i)$ and $\pi_t(i)$ denote $P_t(s_t,a_i)$ and $\pi_t(s_t,a_i)$. Then, for

some learning rate $\alpha = \alpha_t(s_t, a_t)$, the expected updates are

$$\begin{aligned} P_{t+1}(1) &= P_t(1) + \pi_t(1)0.1\alpha(1 - P_t(1)) + \pi_t(1)0.9\alpha(0 - P_t(1)) \\ &= 0.5 - \pi_t(1)0.4\alpha \ , \\ P_{t+1}(2) &= P_t(2) + \pi_t(2)0.9\alpha(1 - P_t(2)) + \pi_t(2)0.1\alpha(0 - P_t(2)) \\ &= 0.5 + \pi_t(2)0.4\alpha \ . \end{aligned}$$

This implies that the expected update decreases the probability of selecting the optimal action a_1 and increases the probability of selecting a_2, if we assume the probabilities are dependent on the preference values, as would normally be the case. The state value will then converge towards -1, the value of the suboptimal action a_2.

This counterexample does not provide much hope for the usefulness of Acla in stochastic settings. However, in many settings the expected sign of the TD error will correspond more closely with the quality of the action resulting in that TD error. In deterministic settings, the preference values of all actions that yield negative TD errors will necessarily decrease. Then, the state values will update towards a value that is at least a high as the lowest value of the remaining actions. At some point the state value will surpass this value after which this action's selection probability will decrease. This will continue until there is only one action left. In the case of immediate rewards, this reasoning is easily transformed into a proof of convergence for deterministic rewards, showing that Acla has potential in such settings.

Furthermore, using the L_1 norm in place of the L_2 norm has the advantage of being able to learn quickly in problems with large plateaus in the error space. This proves to be a beneficial property when we apply Acla with neural networks to the cart pole and mountain car problems in Chapter 7. However, we do note that more research is useful and much better algorithms may exist that build upon L_1-normed errors.

5.8 Experiments

In this section we perform some experiments to observe the differences between the algorithms that were discussed in this chapter. To save space in the tables, we use the abbreviations in Table 5.1. The names of Sarsa and Acla are short enough to use as such.

The experiments are on a small maze problem, on a slightly bigger maze problem and on the well known mountain car benchmark. In all cases, the *total online* results show the average reward per step during learning, the *final online* results show the average reward per step for the last 5% of training steps and the *greedy* results show the average reward per step for following the greedy policy after training has concluded. This greedy policy was always

Table 5.1: Abbreviations for the names of the algorithms.

Algorithm	Abbreviation
Q-learning	Q
Double Q-learning	DQ
Expected Sarsa	ESarsa
QV-learning	QV
Actor Critic	AC

followed for a number of steps equal to 10% of the total number of training steps.

5.8.1 Parameter Settings

In all cases a learning rate of $1/n_t(s_t,a_t)^{0.8}$ was used, where $n_t(s_t,a_t)$ gives the number of times action a_t has been selected in state s_t in the first t time steps. The exponent of 0.8 is inspired by a paper by Even-Dar and Mansour (2003) and was indeed found to yield better results in our experiments than a learning rate of $1/n_t(s_t,a_t)$. This was also apparent in the experiments we conducted in Chapter 4.

Boltzmann exploration was used and the results are shown corresponding to the best choice of temperature, where $\tau \in 10^x$ and $x \in \{-2,-1,\ldots,3\}$. Further tweaking of the temperature and learning rate might yield slightly better results, but the general conclusions remain similar. In contrast to ϵ-greedy exploration, the amount of exploration that results from Boltzmann exploration is not scale independent. Suppose we have two enumerated sets of values $X = \{x_1,\ldots,x_n\}$ and $Y = \{y_1,\ldots,y_n\}$, where the elements in Y differ from those in X only through scaling with some factor z, such that $y_i = zx_i$. A Boltzmann distribution with temperature τ_X on X would result in

$$p(x_i) = \frac{e^{x_i/\tau_X}}{\sum_j^n e^{x_j/\tau_X}} \quad ,$$

where $p(x_i)$ gives the probability of selecting element x_i A Boltzmann distribution with temperature τ_Y on Y would result in

$$p(y_i) = \frac{e^{y_i/\tau_Y}}{\sum_j^n e^{y_j/\tau_Y}} = \frac{e^{x_i(z/\tau_Y)}}{\sum_j^n e^{x_j(z/\tau_Y)}} \quad .$$

Therefore, if the values a set Y differ only from those in a set X by scaling with some factor z, the Boltzmann distribution selects elements with equal probabilities if the temperatures differ with the same scaling, such that $\tau_Y = z\tau_X$.

This property has consequences for our experiments. If we would use preference values for Acla that by definition lie between 0 and 1 as suggested in Section 5.7 and the action values used by Q-learning lie between 0 and 100, Acla would need a much lower Boltzmann temperature to reach similar levels of exploration. This effect is apparent in previous work where Acla indeed preferred different settings of this parameter than other algorithms (Wiering and van Hasselt, 2008, 2009). We attempt to make the amount of exploration for each of the algorithms more or less uniform. In all MDPs, we choose the reward function in such a way that all optimal action values lie between 0 and 100 or between −100 and 100, if negative values were in order. Additionally, we updated Acla towards 100 and 0, instead of 1 and 0.

5.8.2 Statistical Significance

Finally, before we get to the experiments, we will say a few words about statistical significance. According to a Welch's test (Welch, 1947), the difference D between the means of two sets with sizes N_1 and N_2 is significant when

$$\frac{D}{\sqrt{se_1^2 + se_2^2}} > x \ ,$$

where se_i is the sample standard error of set i. Here x is the value from a t-distribution table for the corresponding degrees of freedom and desired threshold α for our p-value. In our experiments, we obtain sufficient samples to assume $df \approx \infty$. If we use $\alpha = 0.01$, we obtain $x = 2.326$. Therefore, we will call the difference between the sampled mean reward of two algorithms significant if

$$D > 2.326\sqrt{se_1^2 + se_2^2} \ . \tag{5.22}$$

We will explicitly state the standard errors of our results, so it is possible to redo the significance calculations for instance if a lower threshold is preferred.

It should be noted there can be a selection bias in the results, since at some points we choose the best result for a number of parameter settings. As such, the results in theory can have a positive bias. A normal way to handle this would be to test the best found parameters on a new test run. However, this may introduce a negative bias. For a lengthier discussion about these biases, see Chapter 3.

Since the exploration parameters lie far apart, in most cases there is a large difference in performance between the best setting and worse settings, unless multiple settings either reach near optimal policies or when the best setting is near random. We compared the best results for each algorithm to the validation of these parameters and found that the results are usually very close. This does mean that if the experiments are repeated, in some cases different parameter settings may emerge as the best settings and that in some

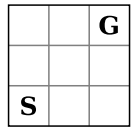

Figure 5.3: A small grid MDP.

cases the results for the parameter settings as displayed may be somewhat lower. However, this does not have any impact on the main conclusions.

5.8.3 Grid World

We start with a very small grid world MDP as show in Figure 5.3. In each state there are 4 actions, corresponding to the directions the agent can go. The starting state is in the lower left position and the goal state is in the upper right. There are no obstacles and each time the agent selects an action that would walk off the grid, the agent stays in the same state. The discount factor was set to 0.95. We constructed a few different versions of this MDP, varying the reward function in each. Every variant gives a reward of +100 for each action in the goal state and then ends the episode. The variants differ in what reward is given at every non-terminating time step. The optimal policy is always to end an episode in the goal state after five actions.

5.8.3.1 Deterministic Grid World

In the first variant every time step yields a reward of zero. This variant is fully deterministic and therefore the easiest variant we discuss. Each algorithm we discussed in this chapter learns to solve this very simple MDP perfectly within 1000 training steps. For instance, if a Boltzmann exploration with a temperature of 1 is used the last 50 steps of the 1000 training steps were always flawless for all algorithms, showing good on-policy performance. Additionally, the greedy policy was always perfect.

5.8.3.2 Neutral Stochastic Grid World

In the second variant, the agent receives a random reward of -100 or $+100$ on each step with equal probability. We call this variant neutral stochastic since every non-terminating step yields a stochastic reward with an expected value of zero. The only difference with the first variant is the stochasticity of the rewards. Table 5.2 shows the results for this MDP.

Clearly all algorithms take longer to find good policies on the MDP, because of the added noise in the feedback. The results in the table show that QV learning has the best mean on-policy performance over the $1,000$ train-

Table 5.2: Results for the neutral stochastic grid world with non-terminating rewards of +100 or −100. The table shows average results over 1,000 repetitions of the experiments (μ), standard errors of these averages (se) and the Boltzmann temperature (τ) that resulted in this best performance.

After 1,000 steps									
	total online			final online			greedy		
	μ	se	τ	μ	se	τ	μ	se	τ
Q	3.46	0.10	1000	2.99	0.46	1000	0.90	0.34	1
DQ	4.78	0.23	10	6.74	0.53	10	6.03	0.43	10
Sarsa	4.13	0.09	100	4.78	0.45	100	2.87	0.37	1000
ESarsa	4.45	0.09	100	5.82	0.45	100	3.62	0.39	1000
QV	5.48	0.10	100	6.28	0.44	100	4.11	0.40	100
AC	4.70	0.10	1000	5.08	0.48	100	3.75	0.40	100
Acla	4.33	0.10	100	4.40	0.45	100	1.92	0.36	10
best	5.48	**QV**		6.74	**DQ**		6.03	**DQ**	
After 10,000 steps									
	total online			final online			greedy		
	μ	se	τ	μ	se	τ	μ	se	τ
Q	3.44	0.03	1000	3.21	0.14	1000	1.71	0.35	10
DQ	7.22	0.87	10	8.88	1.10	10	9.16	1.35	10
Sarsa	5.01	0.03	100	5.04	0.14	100	10.19	0.42	1000
ESarsa	5.22	0.03	100	5.32	0.14	100	10.34	0.44	1000
QV	5.58	0.03	100	6.96	0.32	10	10.13	0.42	1000
AC	7.42	0.21	100	9.21	0.27	100	8.92	0.44	100
Acla	4.09	0.07	10	4.26	0.15	10	1.56	0.34	10
best	7.42	**AC**		9.21	**AC**		10.34	**ESarsa**	

ing steps. The difference is statistically significant with all other algorithms. For both the on-policy and off-policy final performance, Double Q-learning is the best algorithm, although in the on-policy case the difference with QV-learning, Expected Sarsa and Actor Critic is not statistically significant. In the off-policy case, the difference between Double Q-learning and the other algorithms is statistically significant. The reason for this is that Double Q-learning is an off-policy algorithm that can estimate the optimal policy when following a different policy. The poor performance of Q-learning is striking and from Chapter 4 we know this is due to the overestimation bias.

We also ran the same experiment for 10,000 time steps. The results of this experiment are shown in the lower part of Table 5.2. Again, Double Q-learning and to a lesser degree QV-learning perform well, although Actor Critic has a better on-policy performance. Apparently it takes longer on this

problem before Actor Critic finds good policies, but then the online performance surpasses that of QV-learning. The difference between Actor Critic and Double Q-learning is not significant. Both the total and the final on-policy performance of Actor Critic and Double Q-learning is significantly better than all other algorithms. However, the final greedy policy that is found by Sarsa, Expected Sarsa and QV-learning after 10,000 steps is better than that of Double Q-learning, although the difference is not significant.

It is interesting to note that on this problem the on-policy algorithms seem to reach performance levels at least as good as the off-policy algorithms, as long as the learning parameters are explicitly optimized for this performance metric. However, this may be very problem dependent, since we see that the best greedy performance is reached for a behavior policy that is nearly completely random, with a temperature of $\tau = 1000$. Such a policy would indeed favor states closer to the goal state, also in the on-policy values of a random policy. However, in many problems a random policy will not be good enough to reach certain interesting parts of the state space. This means that the results here at least show some peculiarities of the problem, rather than just differences between the algorithms. In all likeliness, this holds for any specific problem setting.

In all settings, the worst performance is obtained by Q-learning and Acla. The on-policy performance of Q-learning is approximately random, which can also be seen by the fact that the best Boltzmann temperature for Q-learning was $\tau = 1,000$. In this setting, this leads to almost random policies. Apparently, every policy found by Q-learning is worse than random. This can also be concluded from its greedy performance, which indeed is worse on average than a random policy.

5.8.3.3 Negative Stochastic Grid World

In this next variant, the rewards on each non-terminating step are +90 or −110 with equal probabilities. This results in an expected value of −10 on each step. Therefore, the optimal policy is now only worth on average 12 per step, compared to 20 in the previous case. The negative values on each step present an additional incentive for the algorithms to reach the goal state as quickly as possible. This seemingly small adaptation of the problem can have different effects on the different algorithm. For completeness we mention that on an MDP with deterministic negative rewards on each non-terminating transition each algorithm reaches optimal on-policy and off-policy withing 1,000 learning steps. The results for the stochastic variant are shown in Table 5.3.

The results show that after 1,000 training steps, Double Q-learning significantly outperforms the other algorithms in a statistically and absolute sense. On average, its total online performance is more than 2 points better

Table 5.3: Results for the negative stochastic grid world with non-terminating rewards of +90 or −110. The table shows average results over 1,000 repetitions of the experiments (μ), standard errors of these averages (se) and the Boltzmann temperature (τ) that resulted in this best performance.

After 1,000 steps									
	total online			final online			greedy		
	μ	se	τ	μ	se	τ	μ	se	τ
Q	-6.01	0.10	1000	-6.63	0.46	1000	-7.00	0.38	10
DQ	0.20	0.24	10	5.38	0.50	10	4.47	0.46	10
Sarsa	-4.31	0.10	100	-2.51	0.44	100	-3.88	0.42	100
ESarsa	-4.07	0.10	100	-2.81	0.44	100	-3.69	0.43	100
QV	-2.46	0.25	10	0.91	0.53	10	1.47	0.46	10
AC	-2.28	0.20	100	0.17	0.46	100	-1.12	0.45	100
Acla	-5.23	0.09	100	-5.17	0.48	10	-6.94	0.37	10
best	0.20	**DQ**		5.38	**DQ**		4.47	**DQ**	
After 10,000 steps									
	total online			final online			greedy		
	μ	se	τ	μ	se	τ	μ	se	τ
Q	-1.52	0.23	10	3.83	0.32	10	4.71	0.43	10
DQ	8.62	0.16	10	11.17	0.18	10	11.55	0.32	10
Sarsa	-0.16	0.23	10	5.46	0.31	10	10.52	0.35	100
ESarsa	-0.20	0.23	10	5.66	0.31	10	10.38	0.35	100
QV	6.89	0.20	10	10.47	0.20	10	10.82	0.33	100
AC	6.00	0.16	100	9.68	0.18	100	10.71	0.33	100
Acla	-3.94	0.07	10	-4.28	0.16	10	-8.54	0.35	1000
best	8.62	**DQ**		11.17	**DQ**		11.55	**DQ**	

than the next best algorithms QV-learning and Actor Critic. Although these algorithms also improve their online performance, for the last 50 steps this difference has risen to more than 4 points. Naturally, this can be a statistical overestimation since the performance of the greedy policy is somewhat lower, but it does show that for this problem Double Q-learning learns good policies much faster than the other algorithms.

After 10,000 training steps, Double Q-learning is still significantly better than the other algorithms in its online performance, although its greedy policy is no longer significantly better than that of Sarsa, QV-learning and Actor Critic. This is due to the improved behavior of these algorithms. With the exception of Acla, all algorithms find better policies than in the stochastic setting without negative expected rewards on each step. This can be seen by

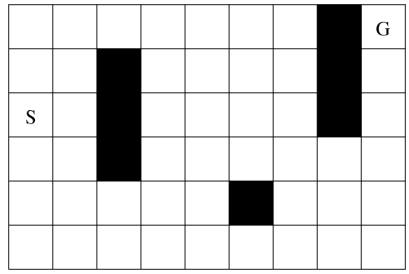

Figure 5.4: Sutton's Dyna maze. The starting position is indicated by S and the goal position is indicated by G. The black squares represent obstacles.

the fact that the results are much closer to the optimal value of 12 than the results in Table 5.2 were to the optimal value of 20. Acla can not handle the stochasticity in these problems well and performs poorly.

5.8.4 Dyna Maze

Next, we consider the performance of the algorithms on a larger maze, which has been used a number of times in the past to measure the performance of reinforcement learning algorithms in various settings (Sutton, 1990; Sutton and Barto, 1998; Wiering and van Hasselt, 2008, 2009). The maze is shown in Figure 5.4.

The setting is more or less the same as before, only now the agent receives a negative reward of -0.1 on each step, as well as a reward of -2 whenever it bumps into an obstacle. The black squares and the outer boundary of the grid are such obstacles. When the goal is reached, a reward of 100 is obtained. The optimal policy takes 14 steps to reach the goal and the optimal average reward per step is therefore 7.05. The results of $10,000$ steps of training are shown in Table 5.4.

Because of the longer routes to the goal, in a state relatively far from the goal state the difference in value between the best action and a suboptimal action will be much lower. Additionally, in the beginning the action values will appear to be overestimates, because of the -0.1 reward on each step. This implies there is an incentive to explore and therefore the best performing Boltzmann temperatures are now also lower in general. The best algorithms for this problem are the traditional algorithms Q-learning and Sarsa. Expected Sarsa is slightly better than Sarsa, as expected, although the difference is not statistically significant. QV-learning also finds a good greedy policy, although its online improves slightly slower than that of Sarsa and Q-learning, as can be seen from the somewhat lower total online performance. For all metrics, Actor Critic and Acla perform the worst.

Table 5.4: The results for the Dyna maze: the best average results over 100 runs of the experiments (μ), standard errors of these averages (se) and the corresponding Boltzmann temperature (τ) are shown. Training was 10,000 steps for the deterministic maze and 50,000 steps for the stochastic maze.

Deterministic Dyna Maze									
	total online			final online			greedy		
	μ	se	τ	μ	se	τ	μ	se	τ
Q	6.04	0.03	10^{-3}	6.97	0.02	10^{-4}	6.97	0.03	10^{-4}
DQ	2.47	0.17	10^{-3}	4.70	0.21	10^{-4}	4.66	0.32	10^{-3}
Sarsa	6.04	0.03	10^{-4}	6.97	0.02	10^{-4}	6.97	0.03	10^{-4}
ESarsa	6.08	0.02	10^{-4}	7.01	0.02	10^{-4}	7.03	0.01	1
QV	5.10	0.10	0	6.96	0.04	10^{-3}	7.01	0.02	10^{-3}
AC	1.82	0.18	1	3.20	0.19	1	3.58	0.35	1
Acla	1.58	0.24	0.1	2.59	0.22	0.1	4.47	0.32	1
best	6.08	**ESarsa**		7.01	**ESarsa**		7.03	**ESarsa**	

Stochastic Dyna Maze									
	total online			final online			greedy		
	μ	se	τ	μ	se	τ	μ	se	τ
Q	3.13	0.13	1	4.33	0.12	1	5.33	0.01	100
DQ	3.50	0.15	0	4.41	0.10	1	5.30	0.01	100
Sarsa	3.02	0.14	1	3.98	0.13	1	5.30	0.01	10
ESarsa	3.15	0.14	1	4.15	0.12	1	5.30	0.01	10
QV	2.63	0.18	0	3.47	0.11	1	5.24	0.05	10
AC	2.40	0.18	1	3.33	0.15	1	5.05	0.08	10
Acla	1.70	0.19	0	2.41	0.16	0	4.76	0.11	100
best	3.50	**DQ**		4.41	**DQ**		5.33	**Q**	

The lower half of Table 5.4 shows the results on a stochastic version of the Dyna maze, in which there was a 20% chance that the selected action was replaced by a random action. This implies that the transitions are now stochastic and one may end up in different states when performing the same action in the same state twice. The optimal average reward per step in this maze is close to 5.3. In this setting the same algorithms have good performance and additionally Double Q-learning performs well with the best online performance, although the difference with Q-learning and Expected Sarsa is not significant. The results are closer than in the deterministic maze because we let the algorithms train for 50,000 steps, instead of 10,000 steps. Eventually, all algorithms find an optimal policy in both these settings if they are given enough learning steps.

We note that Acla generally performs somewhat poor in these problems.

However, in previous work it was shown that Acla can perform as well as or better than the other algorithms in many problems if the learning rates are optimized for each problem (Wiering and van Hasselt, 2007). This indicates that Acla may be somewhat harder to tune and therefore somewhat less practical than the other algorithms. The main reason we include the algorithm in our analysis is that there are some problems, such as the well known cart pole and mountain car benchmarks, in which Acla was in fact significantly better (van Hasselt and Wiering, 2009; Wiering and van Hasselt, 2009). It seems Acla performs well when combined with neural networks. We will discuss these results in more detail in Chapter 7.

5.8.5 Mountain Car

A well known benchmark in reinforcement learning is the mountain car problem (Singh and Sutton, 1996; Sutton and Barto, 1998). In this problem, a car is initialized at the bottom of a two-dimensional valley and the goal is to climb the hill to its right. The setting is interesting because of three characteristics. First, the inputs to the reinforcement learning agent are continuous. However, in this chapter we will discretize the state space, removing this point of interest. For results on the continuous version of the mountain car, we refer to Chapter 7. Second, the engine of the car is too weak to climb the hill directly, and the car must therefore first make momentum by driving up the hill to its left. Third, the reinforcement learning agent does not receive positive feedback until it reaches the top of the right hill and must therefore explore to solve the problem. To elaborate on this last point, one can imagine that the agent does not know anything about the world it is in, so it might think that every possible setting is of the same quality as the setting it is initialized in. Without some incentive to explore, either explicit or implicit, there is no compelling reason for it to leave this setting.

In this chapter, we construct a tabular version of the mountain car, simply by partitioning the position and the velocity into 8 equally sized segments, for a total of 64 different states.[1] In Chapter 7, we consider the same problem but without the discretization of the state space.

We used the same problem description as in the book by Sutton and Barto (1998). This means that the position of the car x and its velocity dx are updated as follows:

$$\begin{aligned} x_{t+1} &= x_t + dx_{t+1} \ , \\ dx_{t+1} &= dx_t + 0.001 a_t - 0.0025 cos(3x_t) \ . \end{aligned} \tag{5.23}$$

The position is bounded to $[-1.2, 0.5]$ and the velocity is bounded to $[-0.07, 0.07]$. When the position becomes lower than -1.2 it is assumed the car hits the left

[1] Counting the terminal state that is reached upon reaching the goal position, one could argue in fact there are 65 states.

boundary of the problem and its position is set to -1.2 with zero velocity. An episode is considered a success and ends when a position higher than 0.5 is reached. The action space is finite and consists of the set $A(s) = \{-1, 0, 1\}$ for each state. The transitions are as described by the equations in (5.23). Whenever an episode ends, the car is reset to $x = -0.5$ and $dx = 0.0$, which is near the bottom of the track with zero velocity. The reward function is defined such that the agent receives a reward of -1 on every time step, except when an episode ends in a success and a reward of $+100$ is received. If no success is obtained within 500 time steps, the episode is considered a failure and the episode ends. This last transition rule actually makes the problem non-Markovian since the agent does not receive the time step in its state description. In practice this is not a large problem and it makes the experiments run faster.

The performance on the mountain car was quite poor when only a learning rate of $1/n_t(s_t, a_t)^{0.8}$ was used, so instead we mildly optimized a constant learning rate for each algorithm. The possible learning rates were 10^x for $x \in \{-4, -3, -2, -1\}$.

Table 5.5 shows results on the mountain car. The most remarkable result is the result of the Actor Critic algorithm, which reaches a very good greedy policy after training. All the algorithms show improvement over the training. Interestingly, Sarsa performs better on this setting than Expected Sarsa, although only the difference in the final online performance is significant. A possible explanation for this may be that Sarsa's inherently noisier updates are somewhat beneficial in this problem since the performance is very dependent on the first time that an algorithm accidentally reaches the goal state.

The problem is not stochastic enough to cause overestimation problems for Q-learning. Therefore, Q-learning performs better than Double Q-learning on the mountain car.

5.9 Conclusion

In this chapter we have discussed many different algorithms to learn action values. We have attempted to highlight the most important differences and similarities between the algorithms and we have performed some experiments which show that there are real differences in terms of how quickly each algorithm on average finds a good policy.

One of the more interesting results is that Q-learning behaves well in the deterministic settings, but performs very poorly in both stochastic grid world settings. The reason for this was discussed in Chapter 4. In the stochastic grid world Q-learning seems to suffer from less problems, possibly because the stochasticity in that setting was only in the transitions and not in the rewards.

Table 5.5: The results for the mountain car (lower is better): the best average results over 100 runs of the experiments (μ), standard errors of these averages (se) and the corresponding Boltzmann temperature (τ) are shown. Training was 100,000 steps, testing was 5,000 steps.

	total online		final online		greedy	
	μ	se	μ	se	μ	se
Q	281.4	2.2	194.3	4.3	211.4	8.7
DQ	342.2	1.9	211.3	3.9	246.5	12.3
Sarsa	286.9	2.3	181.5	3.6	187.4	7.6
ESarsa	282.9	2.3	200.7	5.6	208.6	9.5
QV	293.0	4.1	201.5	3.3	219.8	10.1
AC	255.7	3.3	188.6	4.9	136.3	0.5
Acla	322.1	7.1	253.5	11.8	241.8	12.9
best	255.7	**AC**	181.5	**Sarsa**	136.3	**AC**

	parameters								
	τ	α	β	τ	α	β	τ	α	β
Q	0.01	0.1	-	0.01	0.1	-	0.01	0.1	-
DQ	0.01	0.1	-	0.01	0.1	-	0.01	0.1	-
Sarsa	0.01	0.1	-	0.01	0.1	-	0.01	0.1	-
ESarsa	0.01	0.01	-	0.01	0.1	-	0.01	0.1	-
QV	0.01	0.1	0.1	0.01	0.1	0.01	0.01	0.1	0.01
AC	0.1	0.1	0.01	0.1	0.1	0.01	1	0.01	0.01
Acla	0.01	0.01	0.01	0.01	0.001	0.001	0.01	0.001	0.001

Another good thing to note is the consistent performance of the QV-learning algorithm. Although it was rarely the best algorithm, it never performs very poorly. The only drawback seems to be a slightly slower convergence rate, but this can also be due to the specific learning rate that we used. In previous work, some results have been published that show that QV-learning can be tuned to perform better than Q-learning and Sarsa in many occasions (Wiering, 2005; Wiering and van Hasselt, 2007, 2009).

Further, it is interesting to note that all algorithms perform well on at least one of the tasks. Which algorithm performs best is clearly very task dependent. On the one hand, this is good news since it means that there are many ways to find a good policy. On the other hand, this adds complexity for someone who wants to solve a problem, since an algorithm needs to be selected in addition to the tuning of parameters.

For this reason, in the next chapters we will investigate ensembles of the different algorithm. Ideally, this removes the need to choose between the different algorithms. Additionally, we will see that the ensemble often performs

better than each of the individual algorithms.

5.10 Proofs

This section contains the proofs for the theorems in this chapter.

5.10.1 Proof for Theorem 5.1

Proof. To prove this theorem, we simply check that all the conditions of Lemma 2.1 are fulfilled. The first and second conditions of this lemma correspond to the first and second assumptions of the theorem. The fourth assumption in the theorem ensures that $\text{Var}\{F_t(x_t)|P_t\}$ stays bounded. The process is then upper bounded by Q-learning and lower bounded by a form of Q-learning that uses a min operator in place of a max operator. These Q-learning processes are known to converge and stay bounded if the reward function has finite variance. Below, we will show the third condition of the lemma holds.

We can derive the value of F_t as

$$\begin{aligned} F_t(s_t,a_t) &= \frac{1}{\alpha_t}\big(\Delta_{t+1} - (1-\alpha_t(s_t,a_t))\Delta_t(s_t,a_t)\big) \\ &= r_t(s_t,a_t) + \gamma \sum_a \pi_t^e(s_{t+1},a) Q_t(s_{t+1},a) - Q^*(s_t,a_t) \ , \end{aligned} \tag{5.24}$$

We can assume $F_t(s,a) = 0$, for all $(s,a) \neq (s_t,a_t)$. If we can show that $\|E\{F_t|P_t\}\| \leq \kappa\|\Delta_t\| + c_t$, where $\kappa \in [0,1)$ and c_t converges to zero, all the conditions of the lemma can be fulfilled and we have convergence of Δ_t to zero and therefore convergence of Q_t to Q^*. We derive this as follows:

$$\begin{aligned} &\|E\{F_t|P_t\}\| \\ &= \left\| E\left\{ r_t + \gamma \sum_{a'} \pi_t^e(s_{t+1},a') Q_t(s_{t+1},a') - Q^*(s,a) | P_t \right\} \right\| \\ &= \left\| E\left\{ r_t + \gamma \sum_{a'} \pi_t^e(s_{t+1},a') Q_t(s_{t+1},a') | P_t \right\} - \sum_{s'} P_{sa}^{s'} \left(R_{sa}^{s'} - \gamma \max_{a'} Q^*(s',a') \right) \right\| \\ &= \left\| \sum_{s'} P_{sa}^{s'} \left(\gamma \sum_{a'} \pi_t^e(s',a') Q_t(s',a') - \gamma \max_{a'} Q^*(s',a') \right) \right\| \\ &\leq \gamma \max_s \left| \sum_a \pi_t^e(s,a) Q_t(s,a) - \max_a Q^*(s,a) \right| \\ &\leq \gamma \max_s \left| \max_a Q_t(s,a) - \max_a Q^*(s,a) \right| + \gamma \max_s \left| \sum_a \pi_t^e(s,a) Q_t(s,a) - \max_a Q_t(s,a) \right| \\ &\leq \gamma \|\Delta_t\| + \gamma \max_s \left| \sum_a \pi_t^e(s,a) Q_t(s,a) - \max_a Q_t(s,a) \right| \ . \end{aligned}$$

The first equality uses (5.24), the second equality uses the Bellman optimality equation and the third equality uses algebraic rewriting. The first inequality uses the fact that a maximum weighed difference cannot be greater than the maximum difference over all the elements in the weighted sum. The second inequality results from the triangular inequality. The third inequality uses the fact that the difference between the maxima of two sets cannot be greater than the maximal difference between any two corresponding elements in these sets.[2] The last step also applies the definition in (2.28).

We identify $c_t = \gamma \max_s |\sum_a \pi_t^e(s,a) Q_t(s,a) - \max_a Q_t(s,a)|$ and $\kappa = \gamma$. Clearly, c_t converges to zero for estimation policies that are greedy in the limit. Therefore, if $\gamma < 1$, all of the conditions of Lemma 2.1 follow from the assumptions in the present theorem and we can apply the lemma to prove convergence of Q_t to Q^*. □

[2]To see this, assume two sets X and Y of equal size, where $X_M = \max_i X_i$ and $Y_N = \max_i Y_i$. If $M = N$, it follows that $|X_M - Y_N| = |X_M - Y_M| \leq \max_i |X_i - Y_i|$. If $M \neq N$, without loss of generality assume that $X_M \geq Y_N$. By definition of N we know $Y_N \geq Y_M$, so it follows that $|X_M - Y_N| \leq |X_M - Y_M| \leq \max_i |X_i - Y_i|$.

CHAPTER 6

ENSEMBLE ALGORITHMS IN REINFORCEMENT LEARNING

As we outlined in the previous chapters, there is no single value-based reinforcement learning algorithm that is dominant in terms of finding good policies quickly in all possible settings. This is the case in more general in the field of machine learning and is sometimes referred to as the *no free lunch theorem* (Schaffer, 1994; Wolpert and Macready, 1995, 1997). This theorem states that in order to get better on some subset of problems, an algorithm has to become worse at another subset of problems. The actual statement that is proved is more narrow and obviously there exist algorithms that can be improved on a subset of problems without suffering performance penalties on other problems. But in general we have seen a similar development in the reinforcement learning algorithms described thus far, that indicates that the best choice of algorithm can be very dependent on the structure of the MDP.

In supervised learning, ensemble methods such as bagging (Breiman, 1996), boosting (Freund and Schapire, 1996), and mixtures of experts (Jacobs et al., 1991) have been proposed. These methods combine multiple classifiers, for example with a weighted voting scheme. Each classifier casts a vote that may be weighted by its confidence on the classification and these votes are combined into a new classification. This ensemble classifier is then often at least as good as the best individual classifier, especially when the errors of the individual classifiers are reasonable low.

In reinforcement learning, ensemble methods have been used for representing and learning the value function (Singh, 1992; Tham, 1995; Sun and Peterson, 1999; Ernst et al., 2005). This allows combinations of multiple value-based algorithms, but it does not allow combinations of action value algorithms with algorithms that learn preference rather than action values, such as the actor-critic method described in Chapter 5. Also, it is not possible to include algorithms that do not use value functions at all, such as evolutionary algorithms that search directly in the policy space (Whitley et al., 1993; Moriarty and Miikkulainen, 1996; Moriarty et al., 1999; Taylor et al., 2006; Whiteson and Stone, 2006; Wierstra et al., 2008; Rückstieß et al., 2010). Therefore, in this chapter we explore ensemble methods that use voting schemes to directly combine policies of individual algorithms into a combined policy. Literature on such methods is extremely scarce, although the few results obtained so far have been promising (Jiang and Kamel, 2006; Wiering and van Hasselt, 2008; Hans and Udluft, 2010).

In this chapter we describe several ensemble methods that combine multiple reinforcement learning algorithms in a single agent. The aim is to en-

hance performance by combining the chosen actions or action probabilities of different algorithms. Each algorithm will learn its own policy, but the ensemble selects the action that is actually performed in the MDP.

Specifically, we will discuss ensembles of the seven value-based reinforcement learning algorithms that were outlined in the previous chapters: Q-learning, Double Q-learning, Sarsa, Expected Sarsa, QV-learning, Actor Critic and Acla. However, the ensembles we discuss allow other types of algorithms. For instance, it is straightforward to include hand-tuned algorithms or algorithms that search directly in the policy space. The reason that we choose to use the seven algorithms we have described is that we observed that they reach different intermediate policies. This allows us to look at the performance of the ensemble methods when the agents that they consist of are different. Additionally, some of the algorithms perform poorly when run individually some of the MDPs. Therefore, our results should indicate which approaches are robust to such poor performing agents in the ensemble.

Section 6.1 introduces the notation used in this chapter and discusses some of the general properties of ensemble methods. We can divide the ensemble methods that we propose in two general groups. The members of the first group of methods are called voting schemes, which will be discussed in Section 6.2. These methods draw largely on voting mechanisms that have been researched for human decision making, for instance through elections. There is a large body of research on such methods, including results on desirable properties (Arrow, 1950, 1963). However, voting schemes often discard some of the available information. Agents rank or express approval for one or more actions, but the actual action selection probabilities are not taken into account. This is why we also propose some methods in a second group, which we call probability based methods. These methods take the actual action selection probabilities of each algorithm and combine these into a policy for the ensemble. The probability based methods are discussed in Section 6.3. In Section 6.4 we summarize all the ensemble methods and look at the differences and similarities between the different approaches. In Section 6.5 we look at the performance of the discussed ensemble methods on a number of different settings and we discuss the results. Finally, Section 6.6 discussed out findings and gives pointers for future research and 6.7 concludes this chapter.

6.1 Ensemble Methods

In this section, we discuss some of the preliminaries concerning ensemble methods. This includes the notation that will be used in this chapter and a discussion about the limiting convergence properties of ensembles. Furthermore, we discuss some of the subtleties involved in using on-policy algorithms within an ensemble.

Ensemble methods have been shown to be effective in combining single classifiers in a system, leading to a higher accuracy than obtainable with a single classifier. Bagging (Breiman, 1996) is a simple method that trains multiple classifiers using a different partitioning of the training set and combines them by majority voting. If the errors of the single classifiers are not strongly correlated, this can significantly improve the classification accuracy. In reinforcement learning, ensemble methods have been used for combining function approximators to store the value function (Singh, 1992; Tham, 1995; Sun and Peterson, 1999; Ernst et al., 2005), and this can be an efficient way to obtain better performance.

In contrast to most previous research in reinforcement learning, we combine different algorithms that learn separate and possibly incomparable value functions and policies. Since the preference values learned by Actor Critic and Acla are different in nature than the action values learned by Q-learning and Sarsa, it is impossible to combine their value functions directly. Therefore in our ensemble approaches we combine policies instead of values. This has the additional advantage that in principle it is possible to add other methods, as long as these methods use some action selection policy. This includes the possibility to add hand-tuned and heuristic policies that may be suboptimal for the MDP as a whole, but in general are better than random. We will not add such policies in our experiments, but in some cases it may be very beneficial to be able to bootstrap the behavior of the ensemble on known good policies.

6.1.1 Notation

An ensemble agent consists of a set of K agents $\mathscr{A} = \{\mathscr{A}_1, \ldots, \mathscr{A}_K\}$ and an ensemble algorithm that determines the action that is taken to interact with the environment. The action selection policy of agent k is π_t^k. If we use this policy to select an action, this action is denoted by a_t^k. However, in many of the ensembles we will directly use the policies. At times, we will use an indicator function $\mathscr{I} : \{\top, \bot\} \to \{0, 1\}$. The input of this function should be considered a Boolean value equal to 'true' ($\top$) or 'false' ($\bot$). The function simply outputs a value of one when its input is true and a value of zero if it is false: $\mathscr{I}(\top) = 1$ and $\mathscr{I}(\bot) = 0$. As usual, the action that is selected by the ensemble to be performed at time t is denoted a_t. As mentioned, it is not necessarily the case that $a_t^k = a_t$ for any algorithm $\mathscr{A}_k$, although it will often hold for at least some of the algorithms.

The policies of the agents that constitute the ensemble are combined to form preference values. In order to distinguish these preference values from those of Acla and Actor Critic we denote these with a lowercase p, such that $p_t(s,a)$ is the preference of the ensemble for action a in state s at time t. Any method of exploration can be used, including ϵ-greedy and Boltzmann exploration. The tunable parameters ϵ for ϵ-greedy and τ for Boltzmann ex-

ploration can then be used to ensure sufficient exploration. For instance, for Boltzmann exploration the resulting ensemble policy is:

$$\forall a \in A(s) : \pi_t(s,a) = \frac{e^{p_t(s,a)/\tau}}{\sum_b e^{p_t(s,b)/\tau}} \ ,$$

where τ is the temperature parameter. How to obtain the preference value function p_t will be discussed below. After calculating the action probabilities, the ensemble selects an action and all algorithms learn from this selected action.

6.1.2 Convergence Considerations

The most important consideration in ensembles of multiple reinforcement learning algorithms is the selection of each action. Unless all the agents agree, there is a relatively large probability that the action that is actually selected by the ensemble is not the action that each individual agent prefers. From the viewpoint of each agent, we can then interpret the selection of an action through the ensemble as a manner of exploration.

The fact that the ensemble determines the selected action, implies that on-policy algorithms will approximate the value of the policy that is followed by the ensemble, rather than the policy that is indicated by its own current action values. In practice, this implies that Sarsa and QV-learning approximate the action values corresponding to the ensemble policy. In our experiments, we choose to let Expected Sarsa update towards the policy that results from its own action values. If agent k uses Expected Sarsa, the update is

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma \sum_a \pi_t^k(s_{t+1},a) Q_t(s_{t+1},a) - Q_t(s_t,a_t)\right) \ .$$

This update is different from the expected update for Sarsa after reaching state s_{t+1}, but before selecting a_{t+1}, which is equal to

$$E\{Q_{t+1}(s_t,a_t)|P_{t+1}\} = Q_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_{t+1} + \gamma \sum_a \pi_t(s_{t+1},a) Q_t(s_{t+1},a) - Q_t(s_t,a_t)\right) \ ,$$

where $P_{t+1} = \{Q_0, s_0, a_0, r_1, s_1, \ldots, r_{t+1}, s_{t+1}\}$ denotes the past experiences up to and including r_{t+1} and s_{t+1}, but excluding a_{t+1}. This means that the algorithm we refer to as Expected Sarsa is in fact an off-policy version of General Q-learning where the estimation policy is equal to the action selection policy of the agent, rather than the policy of the ensemble. We consider it an advantage for our experiments that this version of Expected Sarsa will likely

differ more from Sarsa and QV-learning than a version that uses the behavior policy of the ensemble for its update.

In general, the exploration policy of each agent and the ensemble policy may not be the equal. Therefore, it is not trivial to ensure that the ensemble policy will become greedy with respect to all the individual action value functions, which implies that convergence to the optimal policy may not be guaranteed. For greedy off-policy algorithms such as Q-learning and Double Q-learning, it is sufficient that infinite exploration occurs to ensure convergence so these algorithms can be guaranteed to converge. Similarly, we may make the estimation policy of Expected Sarsa greedy in the limit to ensure this algorithm also converges to the optimal policy. However, convergence of individual algorithms to the optimal policy does not imply convergence of the ensemble as a whole.

If one wants to build a guaranteed convergent ensemble algorithm, it makes sense to use only off-policy algorithms that themselves are guaranteed to converge. However, ensembles that include only similar agents in general fair little better than a single agent. In this case, the General Q-learning framework that was discussed in Chapter 5 can be useful. One could build an ensemble consisting of General Q-learning algorithms that all use different estimation policies. It is then possible to choose the estimation policies in such a manner that they differ, but that they all become greedy in the limit. By Theorem 5.1 we then know that each of the action value functions will then become optimal and therefore an ensemble that combines these policies in a sensible manner will also become optimal. Other convergent off-policy algorithms, such as Double Q-learning, can be added without endangering the convergence properties. In short, an ensemble consisting of one or more agents using Q-learning, Double Q-learning or General Q-learning is guaranteed to converge as long as the ensemble policy ensures infinite exploration and the estimation policy of General Q-learning becomes greedy in the limit.

For the rest of this chapter, we will not concern ourselves too much with convergence in the limit. Rather, we focus on the practical question which types of ensembles seem to work best when given limited time. We do this by including all the algorithms from Chapter 5 and then observing the performance of the different types of ensembles.

6.2 Voting Schemes

In this section we discuss voting schemes that can be used to combine the different policies of the individual agents. We list possible criteria that a voting scheme can adhere to and we discuss these criteria in the context of reinforcement learning. Then we explain how the voting schemes that we will use function. These voting schemes are plurality voting, stochastic approval

Table 6.1: Arrow's criteria for fair voting schemes (Arrow, 1950, 1963).

1. **Unrestricted domain** This criterion holds when all voters should be allowed to rank the alternatives in any strict partial order.
2. **Non-dictatorship** This criterion holds when there is no single voter that determines the outcome of the ensemble by itself.
3. **Pareto efficiency** This criterion holds when the ensemble will rank some alternative A over another alternative B whenever all the voters rank A above B.
4. **Independence of irrelevant alternatives** This criterion holds if the ranking of any two alternatives is dependent only on the ranking of these alternatives according to the voters. Therefore, this criterion does not hold when the ensemble originally ranks A over B, but when an alternative C is considered the ensemble ranks B over A.
5. **Non-imposition** This criterion holds when every ensemble preference is achievable and therefore some set of voter preferences exists for each ensemble preference.
6. **Monotonicity** This criterion holds when the ensemble ranking for each alternative is a monotonic function of the rankings of each voter for this alternative. In other words, if a voter ranks an option higher without changing the order of the other options then the ensemble should never respond by ranking that option lower.

voting, deterministic approval voting, instant runoff voting and Coombs' algorithm. The last two of these methods are majority voting methods in the sense that they eliminate options until a single option is preferred above all remaining options by a majority of agents. This is not an exhaustive list of voting schemes, but in our opinion it includes the most intuitive and promising alternatives.

6.2.1 Criteria for Voting Schemes

There are a number of criteria that a voting system can succeed or fail in meeting. A well known result from social choice theory is Arrow's impossibility theorem, that states that no single voting system can adhere to all members of a set of desirable criteria (Arrow, 1950; Black, 1958; Arrow, 1963; Ray, 1973; Fishburn, 1982). We list these criteria and discuss them in the context of reinforcement learning ensembles.

The criteria are listed in Table 6.1. This list is somewhat redundant, since

the last three criteria imply Pareto efficiency (Arrow, 1963; Osborne and Rubinstein, 1994). Arrow (1950, 1963) has shown that there is no voting method that adheres to all the requirements at the same time.

There are other potentially relevant criteria that we will briefly discuss. The *Condorcet criterion* states that if there is an alternative that is pairwise preferred to each of the other options, this alternative should be chosen (de Caritat, le marquis de Condorcet, 1785; Fishburn, 1977). This alternative is then called the Condorcet winner. Conversely, the Condorcet loser is the alternative that is pairwise preferred to none of the other options. The *Condorcet lower criterion* states that it should not be possible to select this Condorcet loser. Implied by the Condorcet criterion is the weaker *majority criterion* that states that if there is a single alternative that is ranked highest by a majority, this alternative is selected by the ensemble (Straffin Jr, 1980). We also mention the *independence of clones criterion* that states that the outcome remains unchanged if alternatives are added that are equal to ones already included in the set of alternatives (Tideman, 1987). Finally, the *separability criterion* states that if for each possible partitioning into subsets it should hold that if the same alternative is the winner in each subset, this should also be the winner in the combined set (Smith, 1973). Next, we discuss these criteria in the context of reinforcement learning ensembles.

6.2.2 Criteria for Voting Schemes Applied to Reinforcement Learning

We assume that the ensemble has insight into the actual action selection policies of the algorithms it consists of. We do not have to address any concerns about tactical voting, since we assume the algorithms have no interest in giving unfair information. Similarly, we will assume that each algorithm has unrestricted domain in the sense that every policy that is legal in the MDP that is to be solved is allowed by the ensemble.

One very important difference between voting in social choice and voting in a reinforcement learning environment is that the ensemble does not have to be fair. Therefore, we will not consider the non-dictatorship criterion to be very important. Incidentally, this criterion is met by most voting procedures.

Of the ones discussed above, sensible criteria for ensembles in reinforcement learning seem to be the monotonicity criterion, the related Pareto efficiency criterion and possibly the Condorcet criterion. The unrestricted domain and non-imposition criteria are either assumed to hold, or are assumed not to be harmful if violated. For instance, we may decide to impose a restriction on which actions are actually possible, based on domain knowledge. Such an ensemble will then fail the non-imposition criterion, but this may improve the resulting performance if our domain knowledge was accurate. It is not directly clear if meeting or failing the independence on irrelevant alternatives

has direct implications for the performance of an ensemble.

In theory, it seems like a good idea to have *independence of clones*, which states that adding identical alternatives does not change the outcome of the ensemble. As we saw in Chapter 4, for instance the performance of Q-learning can deteriorate quickly when actions with identical effects are added to a state. However, without prior knowledge an agent will not be able to know with certainty that different actions are identical. Therefore, even though two actions may be clones in terms of their effect in the MDP, they may not be identical in how the different algorithms value these actions. So from the point of view of the ensemble, then there are no redundant alternatives, which makes this criterion less applicable. On the other hand, assume there are actions that are clones in the sense that they are valued equal by all algorithms at some point during learning. These actions may or may not be identical in their effect on the long term reward. Therefore, in such a case it even seems undesirable to have a voting system that is independent of clones.

Now we will discuss the different voting schemes that we will use in our experiments.

6.2.3 Plurality Voting

The plurality voting algorithm lets each agent choose a single action. Then, the action with the most votes has the largest probability to be selected. The preference values are then determined by

$$\forall a \in A(s_t): \quad p_t(s_t, a) = \sum_{k=1}^{K} \mathscr{I}(a = a_t^k) \ ,$$

where $\mathscr{I}$ is the indicator function described in Section 6.1.1. The most preferred action is then simply the action that is most often selected by the agent. Note that although we call this procedure plurality voting, it is not necessary that the action with the most votes is actually selected, since the selection may be influenced by the exploration of the ensemble.[1]

Plurality voting fails the Condorcet criterion and can in fact select a Condorcet loser (de Caritat, le marquis de Condorcet, 1785; Fishburn, 1977). This can happen because no information about relative rankings of actions are used and only the top choice of every agent is considered. However, in contrast to the usual assumption in social choice theory, we assume stochasticity in the action selection. Therefore, in fact some information about the individual relative orderings is incorporated in the expected action selection proba-

[1]The plurality voting algorithm that is described here was named 'majority voting' by Wiering and van Hasselt (2008). We prefer the name 'plurality voting', since the action with the most votes is the action that is most likely to be selected, but an action does not need a majority of votes to be selected.

bilities of the ensemble. This invalidates some of the usual problems, at least in expectancy. We will discuss this further in Section 6.3.

Plurality voting does meet the monotonicity criterion and it seems quite vulnerable to clones. For instance, suppose we have an ensemble consisting of ten agents. Further suppose that the current state has four actions, three of which are optimal. Then if there are just four agents that select the suboptimal action, this action has a good chance of being selected by the ensemble because the other votes might get split over the three optimal actions. The more optimal actions there are, the larger the influence will be of groups of agents that incorrectly favor the wrong action. One could argue that usually more optimal actions will reduce the probability of a sensible reinforcement learning algorithm favoring the wrong option, but in Chapter 4 we have seen that in some cases Q-learning structurally favors actions that lead to states with many stochastic actions, even if all these actions have a low expected return. This can be interpreted as an argument in favor of using algorithms that are as different as possible, since this lowers the probability that different agents share the same biases and the resulting misconceptions about which action to choose.

6.2.4 Approval Voting

The approval voting algorithm lets each agent choose more than a single action (Brams and Fishburn, 1978). This means we have to decide in some manner which actions do and do not get approved. For this, we consider two methods: a stochastic and a deterministic method.

For the stochastic method, we can let every agent choose an action more than once. For instance, if we let each agent select two actions $a_t^{k,1}$ and $a_t^{k,2}$, both these actions will get a vote. If the agent chooses the same action more than once, the agent casts a single vote for that action. Agents with strong preferences for a single action will therefore more commonly vote for only a single action than algorithms that have less extreme selection probabilities. Because each action gets at most one vote per agent, it is hard for a single agent to dominate the preferences.

More formally, each agent receives D votes. Then agent $\mathscr{A}_k$ randomly selects actions $a_t^{k,d} \sim \pi_t^k(s)$, for $d \in \{1,\ldots,D\}$ to build a set A_t^k of approved actions, such that $A_t^k = \{a_t^{k,1},\ldots,a_t^{k,D}\}$. As mentioned, the agent may select the same action more than once, so this set may have less than D unique elements. The preference values are then determined by

$$\forall a \in A(s_t): \quad p_t(s_t,a) = \sum_{k=1}^{K} \mathscr{I}(a \in A_t^k) \ . \tag{6.1}$$

The most preferred action is the action that is approved by the most agents. In our experiments, we use $D = A(s)$. This algorithm is called stochastic ap-

proval voting because of the stochastic action selection that determines the contents of A_t^k.

Deterministic approval voting uses a threshold $\nu \in [0,1]$. The approved action set A_t^k of agent k then consists of all actions with selection probability larger than or equal to the threshold. The preferences are computed as in (6.1), which is equivalent to

$$\forall a \in A(s_t): \quad p_t(s_t,a) = \sum_{k=1}^{K} \mathscr{I}(\pi_t^k(s_t,a) \geq \nu) \ . \tag{6.2}$$

If $\nu = 1$ at most one action is approved, and then only when the agent is completely sure about its choice. If $\nu = 0$ all actions will be approved. In practice, both cases lead to random policies for the ensemble agent. In our experiments, we used $\nu = 1/A(s)$, which ensures that every action is approved that has a higher than random probability of being selected by the agent. This threshold is the largest possible that ensures that at all times at least one action is approved by each agent. It is also the lowest possible threshold that can lead to all actions being approved, although this will only happen when the agent has a perfectly uniform policy. When using greedy, ϵ-greedy or suitably scaled Boltzmann exploration procedures, selecting all actions is equivalent to selecting no actions. Therefore, purely random algorithms with equal selection probabilities for all actions do not harm the performance of the ensemble when using deterministic approval voting.

Approval voting meets the monotonicity criterion, but it does not meet the Condorcet criteria. It has been argued that approval rating is sensitive to strategic voting (Niemi, 1984; Saari and Newenhizen, 1988), but since we assume the algorithms will never vote strategically this is no disadvantage in our setting.

6.2.5 Majority Voting

There are settings in which it is desirable not only to have the largest subset of decision makers to favor some outcome, but rather to select an outcome that is deemed to be acceptable by a majority of voters. Notable examples include many real-world elections (Farrell, 2001). Therefore, a number of methods have been proposed to make these decisions. If there are only two options to choose between, plurality voting is one such method if the action with the highest number of votes is always selected. However, for more than two options, plurality and approval voting does not necessarily select a majority winner that is pairwise ranked above each other alternative, if one such majority winner exists.

Majority voting is used to refer to a group of methods. The goal is to obtain a majority for one decision. When there are only two alternatives, this

is equivalent to plurality voting. When there are more than two possible decisions to choose from, there are a number of ways to continue. In previous work, it was proposed to extend the majority voting rule to more than two alternatives by splitting the alternatives into two groups (Jiang and Kamel, 2006). Then, a decision is made through majority on the group, after which the procedure repeats with the chosen group. However, this procedure involves heuristic rules in order to decide how to split the alternatives into groups. Furthermore, depending on this split, it is possible to reach undesirable conclusions. Below, we discuss rank voting algorithms in which each of the agents specifies a partial total order on the set of actions. Two methods are discussed that lead to a majority vote, both of which are special cases of the more general single transferable vote algorithm (Hare, 1873; Tideman, 1995).

Rank voting methods use more information about the action preferences of the different algorithms than simple plurality voting does. Let $r_t^k(s,a)$ denote the ranking weight for action a in state s as specified by algorithm k. These ranking weights follow a partial total order, such that a higher ranked action corresponds to an action that is preferred more by the corresponding agent. For the agent preferences we simply use the policies, such that if $\pi_t^k(s,a) \geq \pi_t^k(s,b)$ then $r_t^k(s,a) \geq r_t^k(s,b)$. When these rankings are specified they can be used in a few different ways. The following three methods are special cases of rank voting, the last two of these methods are majority voting methods.

Borda Count An algorithm can give $A(s)-1$ points to the the most probable action, $A(s)-2$ to the second most probable and so on. For instance, if a is the greedy action in state s for agent k, the ranking weight of this action is given by $r_t^k(s,a) = A(s)-1$. The rank of the action with the lowest action selection probability is then equal to zero. The rank weights of all algorithms can be added to give a final score and the highest scoring action gets the highest preference. This method is called a Borda count (de Borda, 1784; Grazia, 1953; Black, 1958) and the preference values of the ensemble then are

$$\forall a \in A(s_t): \quad p_t(s_t,a) = \sum_{k=1}^{K} r_t^k(s_t,a) \ .$$

The rank weight $r_t^k(s,a)$ of action a in state s can be interpreted as the number of actions that have lower probabilities of being selected by agent k than action a in that state. Ties are broken randomly in our experiments so that always a full strict ordering is obtained. Borda count algorithms have been used to combine classifiers (Ho et al., 1994; Van Erp and Schomaker, 2000) and were used in reinforcement learning ensembles by Jiang and Kamel (2006) and under the name of rank voting by Wiering and van Hasselt (2008).

Instant Runoff The Borda count algorithm can be extended to a majority voting algorithm by determining in multiple rounds which action is to be preferred by eliminating undesirable options. Since we assume that the agents do not change their rankings between these rounds, this can be done using a single transferable vote, as proposed by Hare (1873). The specific case we consider here is also known as instant runoff voting (Richie et al., 2000). Instant runoff voting has been used to elect officials in elections in Australia, Ireland, San Fransisco and London, amongst numerous others (Richie et al., 2000; Marron, 2003).

Instant runoff voting works as follows. Each agent uses a Borda count to rank all the actions according to the policy. The action that is ranked as the most preferred action the least often is then eliminated and removed from the rankings. This procedure is repeated until only one action remains. Equivalently, one may stop when a single action is preferred most by a majority of agents.

Unlike plurality voting, instant runoff will never select a Condorcet loser. There is also some indication that there is a higher probability that this procedure will select the Condorcet winner, even though cases can be constructed in which plurality does and instant runoff does not select this alternative (Grofman and Feld, 2004).

Perhaps more importantly, instant runoff voting fails the monotonicity criterion (Fishburn, 1982). This implies the methods is not Pareto efficient (Arrow, 1963; Osborne and Rubinstein, 1994). Furthermore, instant runoff fails the independence of irrelevant alternatives criterion that states that the preferences of alternatives should not change if a different new alternative is added (Arrow, 1963; Ray, 1973). It is not immediately clear how harmful this is for the performance of the ensemble.

Coombs' Runoff An alternative to the instant runoff algorithms is Coombs' runoff voting algorithm (Coombs, 1960; Grofman and Feld, 2004). Like instant runoff, this algorithm ranks all the actions with a Borda count. But instead of eliminating the least favored action, Coombs' runoff voting eliminates the action that is most often ranked lowest.

An Example Consider five agents who rank three actions. Suppose that the agents rank the actions according to the values in Table 6.2. Action a_3 is favored only by agent $\mathscr{A}_5$. Both other actions are favored by two agents each. Therefore, instant runoff voting first eliminates action a_3. Then, of the remaining two actions a_2 is favored by three agents and therefore the ensemble gives this action a preference of one and both other actions a preference of zero.

On the same rankings Coombs' runoff voting reaches a different conclu-

Table 6.2: An example of s set of rankings for which instant runoff voting reaches a different conclusion than Coombs' algorithm.

	$r_t^1(s,a)$	$r_t^2(s,a)$	$r_t^3(s,a)$	$r_t^4(s,a)$	$r_t^5(s,a)$
a_1	2	2	0	0	0
a_2	0	0	2	2	1
a_3	1	1	1	1	2

sion. Action a_1 is ranked lowest by three agents. Therefore, Coombs' runoff voting eliminates this action first. Then, of the remaining two actions action a_3 is favored by three agents above action a_2. Therefore, Coombs' algorithm selects action a_3.

In this example the preferences according to a Borda count for the actions are $p_t(s,a_1) = 4$, $p_t(s,a_2) = 5$ and $p_t(s,a_3) = 6$. Therefore, a greedy Borda count ensemble agrees with the answer reached by Coombs' runoff voting. This does not hold in general. The action with the highest Borda count may be preferred by instant runoff voting and not by Coombs' runoff voting or all three methods may disagree. Of course, in many cases the algorithms agree on the action with the highest preference. But then there is still a difference between the runoff algorithms on the one hand and Borda count on the other hand. The runoff algorithms always give a preference of one to exactly one action and a preference of zero to all other actions. The possible preferences by a Borda count are much more varied, ranging from a possible minimal preference value of zero to a possible maximal preference value of $K(A(s)-1)$. The preference values will be closer together if the agents do not fully agree about which action is the best and which is the worst, but they still contain more information than the binary judgment of the runoff algorithms. However, we will see that this does not necessarily imply that the Borda count ensemble performs better.

6.3 Policy Based Ensembles

The following two methods transform the policies of the individual algorithms into a combined policy. The first method simply adds the action selection probabilities of the individual algorithms. This is similar to the voting procedures we described above, but uses the most information we can obtain about the algorithms' preferences for each action. The second method has a probabilistic interpretation and is the only method we discuss that uses a product over values obtained from the individual algorithms, as opposed to a summation. Both these methods by construction fulfill the monotonicity criterion as well as the majority criterion. However, both methods are somewhat easier in-

fluenced by agents with strong opinions than most of the other methods we discussed.

6.3.1 Policy Summation

In a policy summation ensemble, we sum the action selection probabilities of the different algorithms. Essentially, this is a variant of rank voting, using $r_t^k = \pi_t^k$ instead of a Borda count. Therefore, the preference values of the ensemble are:

$$\forall a \in A(s_t): \quad p_t(s_t, a) = \sum_{k=1}^{K} \pi_t^k(s_t, a)$$

This algorithm is related to the mean combination rule for combining multiple classifiers (Tax et al., 2000). This method is arguably the closest to range voting (Smith, 2000) , which can be shown to have low Bayesian regret. Bayesian regret can be viewed as the dissatisfaction of voters with the actual outcome. In our case, we can interpret it as the difference between the desired outcome for each agent, and the actual outcome. The difference between this approach and range voting is that in range voting the preference for each element can be chosen from a predetermined range, but the sum of the preferences can be any number. Using the probabilities as preferences means the preferences of each algorithm always sum to one. However, the actual preference of the algorithms is expressed more explicitly than when using the Borda count approach as described above.

Note that the expected value of $\mathcal{I}(a = a_t^k)$, where $\mathcal{I}$ is the indicator function, is equal to $E\{\mathcal{I}(a = a_t^k)|s = s_t\} = \pi_t^k(s_t, a)$. Therefore, the expected preference values as used by plurality voting are equal to the actual preference values resulting from policy summation. This does not imply that the expected policy of the ensemble is also equal for both methods. For instance, if the ensemble uses a greedy action selection mechanism, policy summation will result in a deterministic choice of the action with the largest summed selection probabilities. However, in plurality voting, there may be multiple actions with a non-zero probability of obtaining the plurality. Put differently, if the expected outcome of the ensemble is a non-linear function of the preference values, the expected outcome of plurality voting can be different from the outcome of policy summation, even though the expected preference values are equal. This is a result of Jensen's inequality (Jensen, 1906). In other words, the expected outcome of the ensemble would only be equal for both methods if the preference values were scaled to form a proper policy by dividing them by the number of agents. We do not consider this exploration methods, since we prefer the more flexible Boltzmann exploration. In practice, plurality voting will usually result in more stochastic ensemble policies than policy summation.

6.3.2 Policy Multiplication

Another possibility is multiplying all the action selection probabilities for each action based on the policies of the algorithms. An interpretation of the preference values is that they correspond to the unscaled probability that all algorithms agree on an action. The action that has the highest probability of being chosen unanimously will get the largest preference. The preference values of the ensemble are:

$$\forall a \in A(s_t): \quad p_t(s_t,a) = \prod_{k=1}^{K} \pi_t^k(s_t,a)$$

This algorithm is related to the product combination rule for combining multiple classifiers (Tax et al., 2000).

A potential disadvantage of this method is that if one algorithm sets the probability to 1 for one action and to 0 for the rest, these other actions can never get a non-zero probability in the multiplication. Therefore, this method is vulnerable to dictators and to agents who veto one of the options. However, if all algorithms use some exploration technique that ensures none of the action selection probabilities become extremely low, the preference values can be quite informative.

In practice, the preference values of all actions can be quite close to zero, especially when the ensemble consists of a large number of agents. Furthermore, the ratios between the preference values tend to be higher than for most other methods. For instance, if all algorithms use ϵ-greedy exploration, the preference value for an action a will be equal to $p_t(s,a) = (1-\epsilon)^m \epsilon^{K-m}$, where m is the number of algorithms that consider a best action in state s. Suppose that $\epsilon = 0.1$ and there are 10 agents. Now suppose one action is greedy for 6 agents, while another action is greedy for 3 agents. For policy summation, the respective preference values for this action would then be $6(0.9) + 4(0.1) = 5.8$ and $3(0.9) + 7(0.1) = 3.4$. For policy multiplication, the preferences would be $(0.9)^6(0.1)^4 \approx 5.3 \times 10^{-5}$ and $(0.9)^3(0.1)^7 \approx 7.3 \times 10^{-8}$. In this example, we see that indeed the preference values of policy multiplication are much lower and the ratio between the preference values is also much larger. This means we expect that when policy multiplication is combined with Boltzmann exploration, much lower temperatures are needed to get good performance.

6.4 Summary of Ensemble Methods

In this section we summarize all the methods we have described and briefly discuss some possible variations on these procedures. The eight methods we described are summarized in Table 6.3. The table shows the preference values and the expected preference values. For the expectancy the past up to and including s_t is assumed: $P_t = \{s_0, \ldots, s_{t-1}, a_{t-1}, r_t, s_t\}$.

Table 6.3: The different ensemble methods to combine policies of individual agents. The indicator function $\mathscr{I}$ is defined in Section 6.1.1. For SAV we used $D = A(s_t)$, for DAV we used $v = 1/A(s_t)$. For BC $r_t^k(s,a)$ denotes the number of actions that have lower probabilities than a of being selected by agent $\mathscr{A}_k$. Ties are broken randomly. IRV and CRV deterministically reach a single majority winner. The expected preference values are conditioned on P_t, where $s_t \in P_t$, $a_t^k \notin P_t$ and $a_t \notin P_t$.

Ensemble Method		$p_t(s,a)$	$E\{p_t(s,a)\|P_t\}$
Plurality	(PV)	$\sum_k \mathscr{I}(a = a_t^k)$	$\sum_k \pi_t^k(s,a)$
Stochastic Appr.	(SAV)	$\sum_k \mathscr{I}(a \in A_t^k)$	$\sum_k 1-(1-\pi_t^k(s,a))^D$
Deterministic Appr.	(DAV)	$\sum_k \mathscr{I}(\pi_t(s,a) \geq v)$	$\sum_k \mathscr{I}(\pi_t(s,a) \geq v)$
Borda Count	(BC)	$\sum_k r_t^k(s,a)$	$\sum_k r_t^k(s,a)$
Instant Runoff	(IRV)	0 or 1	0 or 1
Coombs' Runoff	(CRV)	0 or 1	0 or 1
Policy Summation	(PS)	$\sum_k \pi_t^k(s,a)$	$\sum_k \pi_t^k(s,a)$
Policy Multipl.	(PM)	$\prod_k \pi_t^k(s,a)$	$\prod_k \pi_t^k(s,a)$

6.4.1 Stochastic and Deterministic Ensembles

Instead of dividing the methods in Table 6.3 into voting schemes and probability based methods, one could divide the methods into stochastic and deterministic methods. The only two stochastic methods are plurality voting and stochastic approval voting. Both methods sample actions with use of the agents' policies to determine the preference values, so the preference values themselves may contain noise. Deterministic methods always reach the same preference values for a given set of individual policies.

Interestingly, the expected preference values of plurality voting and policy summation are equal although the actual preference values are likely to differ because plurality voting uses sampled actions. For stochastic approval voting, the expected contribution by each agent to the preference values is equal to the probability that the corresponding action is selected at least once out of D samples by that agent:

$$\begin{aligned} &P(\mathscr{A}_k \text{ selects } a \text{ at least once out of } D \text{ samples}) = \\ &\quad 1 - P(\mathscr{A}_k \text{ does not select } a \text{ in } D \text{ samples}) = \\ &\quad 1 - P(\mathscr{A}_k \text{ does not select } a \text{ in 1 sample})^D = 1 - (1 - \pi_t^k(s,a))^D \quad . \end{aligned}$$

For $D = 1$, the expected value of stochastic approval voting reduces to policy summation, which makes sense since then stochastic approval voting reduces to plurality voting. As mentioned before, we use $D = A(s)$ in our experiments.

All other methods are deterministic in the sense that the preference values are uniquely defined by the agents' policies and no sampling is involved.

For instant runoff voting and Coombs' runoff voting the preference value for each action is either zero or one. Which action gets a preference value of one depends deterministically on the policies of the individual agents. If we have to break ties, Borda count, instant runoff voting and Coombs' algorithm can also introduce some randomness, but this will have a more limited effect than the sampling of the aforementioned stochastic algorithms.

6.4.2 Variants

There are infinitely many ways to variate on the ensemble methods we have described. We will discuss some of the more promising approaches here. Although these variants are interesting, we do not consider them further in this dissertation and we will not include them in the experiments.

Like the runoff procedures, plurality voting is normally understood to produce a single winner. This would correspond to setting the preference of the action with the most votes to one and the others to zero. However, we choose to set each preference equal to the number of votes the corresponding action receives. The probability with which the plurality winner is then selected is left to the exploration policy of the ensemble. True plurality voting is obtained when the ensemble uses a greedy selection procedure on the preference values.

For instant runoff and Coombs' runoff, one could obtain a full ranking of actions instead of a single majority winner. If there are $A(s)$ actions, the winner of the runoff procedure could be given a preference of $A(s)-1$. Then the runoff procedure could be repeated with the other $A(s)-1$ actions. The winner of this runoff could be given a preference of $A(s)-2$, and so on.

For Borda count, policy summation and policy multiplication it is easy to construct variants by using any non-linear function on the ranking weights. For instance, one could use a power rule on the ranking weights such that the action with the n^{th} highest selection probability gets a ranking weight of $(A(s_t)-n)^x$. For $x=1$ one obtains the Borda count as we described it, for $x>1$ the highest values actions get weighted more heavily. Similar constructions are possible for the probability based methods and other non-linear functions are also possible.

6.4.3 Implementation Details

We will use Boltzmann exploration on the preference values that are obtained by the ensemble methods. As we have discussed in Section 5.8, Boltzmann exploration is sensitive to scalings of these values. Therefore, it is good to note structural differences in the ranges of these preference values. The possible ranges of any action and the possible ranges of the action with the highest preference value are shown in Table 6.4.

Table 6.4: The possible value for an arbitrary preference value and for the maximal preference value for each ensemble method.

	range of $p_t(s,a)$	range of $\max_a p_t(s,a)$
PV	$\{0,\ldots,A(s)\}$	$\{1,\ldots,A(s)\}$
SAV	$\{0,\ldots,A(s)\}$	$\{1,\ldots,A(s)\}$
DAV	$\{0,\ldots,A(s)\}$	$\{1,\ldots,A(s)\}$
BC	$\{0,\ldots,K(A(s)-1)\}$	$\{\lceil K(A(s)-1)/2\rceil,\ldots,K(A(s)-1)\}$
IRV	$\{0,1\}$	$\{1\}$
CRV	$\{0,1\}$	$\{1\}$
PS	$[0,K]$	$[K/A(s),K]$
PM	$[0,1]$	$[0,1]$

In our experiments, we will have 7 agents and at most 4 actions in any state. This means $K = 7$ and $\max_s A(s) = 4$, so the highest preference value of Borda count will in some cases be approximately ten to twenty times as large as the preference value of the runoff algorithms. To get the same amount of exploration, the Borda count ensemble then needs a larger temperature. This effect becomes larger when there are more actions to choose between.

The biggest difference in practice is that between policy multiplication and the other algorithms. Although in theory an action can obtain a preference value of one, this can only happen when all agents have deterministic policies that agree on the action to choose. In practice, often there will be at least one or two agents that disagree, and the action with the highest selection probability will have a selection probability lower than one. Even if an action is the greedy choice for a majority of the agents, the preference value of this action can easily be lower than 0.01. In short, we expect that policy multiplication will obtain its best results with lower temperatures for the ensemble than the other ensemble methods.

6.5 Experiments

We show the results for five different MDPs of varying complexity. Since the focus of this chapter is on the ensemble methods, we did not fully optimize the parameters of the individual algorithms. We discuss the parameters settings we used in the next section. Then we give the results on the different MDPs for some parameter settings for the individual algorithms and for the ensemble methods described above.

6.5.1 Parameter Settings

As mentioned, we make no attempt to fully optimize the parameters of the individual algorithms. This is in contrast with the work by Wiering and van Hasselt (2008), where first the individual algorithms were optimized and then these algorithms were used in the ensemble. Ideally, the parameters of the individual algorithms could be optimized while using the ensemble and additionally optimizing the parameters of the ensemble itself. However, each algorithm has at least two tunable parameters—an exploration parameter and at least one learning rate—and even if only three possible values would be considered for each parameter, a combined optimization procedure would have try a huge number of options. For instance, an ensemble for the seven algorithms we used here needs ten learning rates, seven exploration parameters and an additional exploration parameter for the ensemble itself. For only three possible values of each, this already adds up to $3^{18} = 387,420,489$ different parameter settings to perform experiments on.

To reduce the number of experiments, we only consider learning rates equal to $\alpha_t(s,a) = 1/n_t(s,a)^{0.8}$ and $\beta_t(s) = 1/n_t(s)^{0.8}$. The Boltzmann temperature was 10^x, where $x \in \{-3,-2,\ldots,3\}$ and additionally greedy exploration was tried. Policy multiplication was allowed to use temperatures as low as 10^{-5}, to compensate for the low preference values. The ensemble methods were only tested with the same exploration parameter for all agents. In the previous chapter, we have seen that the different algorithms prefer different exploration parameters, so restricting them to be equal is a potential disadvantage for the ensemble. However, it does greatly reduce the number of experiments we need to conduct and still gives us the opportunity to compare the different ensemble approaches on equal terms. Furthermore, we think it is useful to consider ensembles of algorithms where not all algorithms perform well, since this tells us more about how robust each of the methods is. For this reason, we also do not update Acla toward 0 and 100, but towards 0 and 1. Since all the MDPs will have a maximal reward of 100, this means that the individual Acla algorithm often reaches its best behavior with quite different exploration parameters than for instance Q-learning.

Now we will discuss the different MDPs and the results of the individual agents and of the ensemble methods. In all cases, the discount factor was set to 0.95. We will use τ to refer to the Boltzmann temperature of the agents and τ_E to refer to the temperature used by the ensemble. For a somewhat consistent notation, the greedy policy will be indicated by $\tau \to 0$, since in the limit Boltzmann exploration is greedy when the temperature approaches zero, although naturally in practice the greedy policy was implemented without using a Boltzmann distribution.

We will use the term *total online* average reward per step to refer to the average of all the rewards that were obtained during training. The *final on-*

line average rewards are obtained during the last 10% of training. How many steps are taken in the last 10% differs between the different problem domains. Because of the nature of the ensembles, we test the final offline performance in two ways. First, we only set the exploration of the ensemble to greedy. This way, the agents' policies still give relevant information to the ensembles that use this information to determine the preference values. Then, simply the action with the highest preference value is taken. A test for 10% of the total training time results in the *final offline* performance. Finally, we set both the agents' and the ensembles exploration to greedy. The *greedy offline* results are obtained by again running a test of 10% of the total training time.

As in Chapter 5 we use a Welch's test with $\alpha = 0.01$ to determine whether two approaches are significantly different. Therefore, we will call the difference between the sampled mean reward of two methods significant if this difference is greater than

$$2.326\sqrt{se_1^2 + se_2^2} \ ,$$

where se_1 is the standard error the first method and se_2 is the standard error of the second method. We will explicitly state the standard errors of our results, so it is possible to redo the significance calculations for instance if a lower threshold is preferred.

6.5.2 Stochastic Grid World

We start with the stochastic 3×3 grid world that was also discussed in Chapter 5. Each state has four actions and each reward is stochastic, except for the final reward which is received for each action in the goal state. The final reward was set to 100. In every other state, each action yields -100 or 100 with equal probability. All the algorithms where trained on the MDP for $1,000$ steps. The online results of this training are shown in Table 6.5.

Table 6.5 contains a wealth of information. First of all, we note that the final online and offline performance of the best ensembles are as good as the best individual algorithms. This is interesting, because as mentioned above we did not optimize the exploration rates of the algorithms in the ensemble. Only in a few cases the ensembles performance falls below the average performance of the algorithms it consists of, even though this average is based on the optimized individual algorithms.

For the total online performance, instant runoff voting is significantly better than all other ensemble methods, except Coombs' runoff voting. The difference between QV-learning and instant runoff voting is not significant. At the final online performance, the advantage of the runoff methods has disappeared and the plurality voting algorithm obtains the best performance. The differences with the other methods are small. The difference with policy multiplication—the worst performing ensemble—is only just significant. The

Table 6.5: Average rewards per step for the first grid world MDP. The average reward is shown for all 1,000 training steps (total online), for the last 100 training steps (final online) and for 100 test steps without exploration after training (final offline). For the ensembles results are shown for when only the ensemble is greedy (final offline) and for when both the ensemble and the consisting algorithms are greedy (final greedy). Values are averaged over 1,000 runs.

	total		**final online**		**final offline**		**final greedy**	
INDIVIDUAL RESULTS								
best	5.48	QV	6.02	DQ	6.03	DQ	—	—
average	4.48	—	5.13	—	3.32	—	—	—
worst	3.46	Q	3.55	Q	0.90	Q	—	—
ENSEMBLE RESULTS								
PV	4.54	0.11	**5.86**	0.32	**6.19**	0.32	3.25	0.37
SAV	4.35	0.10	5.30	0.33	5.18	0.34	3.25	0.37
DAV	4.46	0.10	5.02	0.32	2.60	0.36	2.93	0.37
BC	4.60	0.11	4.96	0.32	3.57	0.36	**4.53**	0.39
IRV	**5.07**	0.10	5.49	0.31	4.04	0.39	3.59	0.39
CRV	4.91	0.10	5.79	0.30	3.39	0.35	3.67	0.40
PS	4.44	0.09	5.72	0.31	3.67	0.39	3.24	0.37
PM	4.09	0.10	4.77	0.32	3.89	0.32	4.03	0.32
best	5.07	**IRV**	6.15	**PV**	6.19	**PV**	4.53	**BC**
worst	4.09	PM	4.15	PM	2.60	DAV	2.93	DAV
ENSEMBLE PARAMETERS								
	τ_E	τ	τ_E	τ	τ_E	τ	τ_E	τ
PV	10^{-1}	100	10^{-2}	100	0	100	10^{-3}	1000
SAV	1	10^{-1}	1	10^{-1}	10^{-2}	100	10^{-1}	1000
DAV	1	1000	1	100	1	10^{-1}	10	1000
BC	1	1000	1	1000	1	0	1	100
IRV	1	1000	1	1000	1	1000	1	1000
CRV	1	1000	1	1000	1	0	1	1000
PS	1	0	10^{-1}	100	10^{-2}	1000	10^{-1}	100
PM	10^{-5}	1000	10^{-3}	10	10^{-1}	0	10^{-5}	10

performance of policy multiplication is not so good, despite the extra exploration options it was given. This is likely due to the nature of the algorithm, which makes it sensitive to strong opinions of individual agents.

When we make the ensemble greedy, such that it always chooses the action with the highest preference value, the differences between the ensemble methods increases. Again, plurality voting obtains the best performance, but the difference is significant with all other methods, except stochastic approval voting. Apparently, for most methods the probability that the ensemble gets stuck increases if there is no exploration to break out of poor policies. Plurality voting and stochastic approval voting introduce some randomness into the preference values, which can act as sufficient noise to get out of apparent local optima in order to go towards the global optima.

In this setting, the local optima are suboptimal actions that are overrated. For instance, an action that bumps into a wall may have received a large number of high rewards by chance, making it seem profitable. Then, a greedy ensemble method can get stuck trying such an action over and over. For instance, we know from Chapter 4 that Q-learning may prefer such recurrent connections due to its overestimation bias. This also holds for Sarsa and Expected Sarsa, if the estimation policies are greedy enough. Therefore, when adding the policies, an action that bumps into a wall can easily be preferred over the other actions in at least some of the states. The stochastic ensemble methods will sometimes try another action, which increases the chance of walking towards the goal.

If we also make the individual agents greedy, the stochasticity disappears and we can see that the final greedy performance of the stochastic ensemble methods then decreases. None of the deterministic approaches reaches a significantly different result compared to the final offline result, but plurality voting and stochastic approval voting drop significantly to the same level as the deterministic approaches. This shows that at least in this setting it can be beneficial to use a stochastic ensemble method.

Plurality voting prefers the exploration to be greedy during training for its best final offline performance, showing that in this case a true plurality voting is preferred to an ensemble that gives a non-zero probability to actions that do not obtain a plurality. Apparently, the stochasticity resulting from the voting procedure results in sufficient exploration to reach good policies.

Policy multiplication may need even lower Boltzmann temperatures to reach good performances in this setting. For the final online and offline performance, it prefers a greedy setting to the lowest temperature it had available of 10^{-5}. We did not try other settings, since we see it as an inherent disadvantage of the method if it requires extra tuning of the parameters to reach good performance levels.

Table 6.6: Average rewards per step for the second grid world MDP. The average reward is shown for all 10,000 training steps (total online), for the last 1000 training steps (final online) and for 1000 test steps without exploration after training (final offline). For the ensembles results are shown for when only the ensemble is greedy (final offline) and for when both the ensemble and the consisting algorithms are greedy (final greedy). Values are averaged over 100 runs.

	total		**final online**		**final offline**		**final greedy**	
INDIVIDUAL RESULTS								
best	8.62	DQ	11.31	DQ	11.61	DQ	—	—
average	2.24	—	5.92	—	7.17	—	—	—
worst	-3.94	Acla	-4.12	Acla	-8.54	Acla	—	—
ENSEMBLE RESULTS								
PV	**6.80**	0.31	**10.12**	0.35	10.78	0.38	11.21	0.63
SAV	6.50	0.45	10.02	0.44	11.32	0.51	10.57	0.63
DAV	5.15	0.60	9.29	0.54	10.27	0.62	11.00	0.60
BC	2.12	0.69	6.45	0.69	9.49	0.75	**11.41**	0.36
IRV	-0.60	0.11	0.84	0.80	10.23	0.66	9.43	0.62
CRV	-0.72	0.10	0.76	0.81	10.25	0.61	10.66	0.48
PS	6.40	0.50	9.92	0.45	**11.59**	0.54	10.91	0.58
PM	0.74	0.65	4.11	0.67	5.95	0.82	-1.50	0.53
best	6.80	**PV**	10.12	**PV**	11.59	**PS**	11.41	**BC**
worst	-0.72	CRV	0.76	CRV	5.95	PM	-1.50	PM
ENSEMBLE PARAMETERS								
	τ_E	τ	τ_E	τ	τ_E	τ	τ_E	τ
PV	10^{-1}	10	10^{-1}	10	10^{-1}	10	10^{-1}	0
SAV	10^{-1}	1	10^{-1}	1	10^{-1}	1	10^{-1}	0
DAV	10^{-1}	10^{-1}	10^{-1}	10^{-1}	10^{-1}	10^{-1}	10^{-1}	10^{-1}
BC	10^{-2}	0	10^{-2}	0	1	100	1	100
IRV	1	100	0	0	1	1000	1	1000
CRV	1	100	10^{-3}	0	1	100	1	1000
PS	10^{-1}	10	10^{-1}	1	10^{-1}	10	10^{-1}	1
PM	0	10^{-1}	0	10^{-1}	0	10^{-1}	0	10^{-1}

6.5.3 Negative Stochastic Grid World

As in Chapter 5 we also consider a variant of the same problem, where the rewards on each step are -110 or 90, instead of -100 and 100. In Chapter 5 we saw that in this variant Double Q-learning was the best algorithm, especially in its total online performance. This settings is interesting also because Acla performed very poor. Therefore, we will be able to see if the ensembles

get effected negatively by agents that prefer bad policies.

The results for the negative stochastic grid world are shown in Table 6.6. The best online ensemble method is plurality voting and its performance if much better than the average individual algorithm. Plurality voting does not reach the high performance of Double Q-learning, but its performance is approximately equal to the second best algorithm, which was QV-learning. The difference is the largest in the total online performance, showing that the ensemble needs a little more time to get to good online performance levels than Double Q-learning.

The final offline and greedy offline performance levels are competitive to Double Q-learning. Interestingly, the offline performance of the runoff algorithms is respectable, although their online performance was extremely poor. If we ignore the online performance of the runoff algorithms, policy multiplication is the worst ensemble method for every metric. It is not immediately clear why the runoff methods' online performance is so poor on this particular setting. However, it does show that the runoff procedures can produce actions that are quite different from the plurality or policy summation winners.

It is interesting to note that the preferred exploration parameter for the agents is not always the parameter the individual agents prefer. For instance, for their best final online performance almost all the individual algorithms preferred a temperature of 10. Of the ensemble methods, only plurality voting and policy summation also reach their best performance when this temperature is used for the agents' policies. All other ensemble methods prefer other exploration settings. This shows that the parameters that optimize the individual agents are not always the best parameters to use when the agents are part of an ensemble. As another example, even plurality voting reaches its best final greedy performance when it was trained with a set of greedy agents.

6.5.4 Deterministic Dyna Maze

We also conducted experiments on the Dyna maze. The maze is shown in Chapter 5 in Figure 5.4. In this setting all ensembles were allowed to use Boltzmann temperatures as low as 10^{-4}. As in Chapter 5, we conducted one experiment with deterministic transitions and one experiment with noisy transitions. In the deterministic case, all ensembles reached good performance levels, close to the best performing individual algorithms. These findings are shown in Table 6.7. The only ensemble method to perform relatively poorly is policy multiplication.

Probably the good performance on the Dyna maze results from the relatively large group of good performing individual agents: Q-learning, Sarsa, Expected Sarsa and QV-learning all performed well on this problem. However, the other three algorithms—Double Q-learning, Actor Critic and Acla—

Table 6.7: Average rewards per step for the deterministic Dyna maze MDP. The average reward is shown for all 10,000 training steps (total online), for the last 1000 training steps (final online) and for 1000 test steps without exploration after training (final offline). For the ensembles results are shown for when only the ensemble is greedy (final offline) and for when both the ensemble and the consisting algorithms are greedy (final greedy). Values are averaged over 100 runs.

	total		**final online**		**final offline**		**final greedy**	
INDIVIDUAL RESULTS								
best	6.08	ESarsa	7.01	ESarsa	7.03	ESarsa	—	—
average	3.75	—	5.03	—	5.80	—	—	—
worst	1.58	Acla	2.59	Acla	3.58	AC	—	—
ENSEMBLE RESULTS								
PV	5.95	0.04	6.95	0.02	7.03	0.01	7.04	0.01
SAV	5.97	0.04	6.92	0.02	7.04	0.01	7.05	0.01
DAV	5.97	0.04	6.94	0.02	6.97	0.07	6.98	0.07
BC	5.97	0.03	6.97	0.02	**7.05**	0.00	**7.05**	0.00
IRV	**6.02**	0.04	**6.98**	0.02	7.02	0.01	6.98	0.03
CRV	**6.02**	0.04	6.97	0.02	7.03	0.01	7.02	0.02
PS	5.87	0.04	6.93	0.02	6.96	0.07	6.97	0.07
PM	1.97	0.24	2.68	0.21	6.38	0.19	2.26	0.19
best	6.02	**CRV**	6.98	**IRV**	7.05	**BC**	7.05	**BC**
worst	1.97	PM	2.68	PM	6.38	PM	2.26	PM
Ensemble Parameters								
	τ_E	τ	τ_E	τ	τ_E	τ	τ_E	τ
PV	10^{-2}	10^{-2}	10^{-1}	10^{-2}	1	10^{-2}	1	10^{-2}
SAV	10^{-4}	10^{-2}	10^{-1}	0	1	10^{-2}	1	0
DAV	10^{-1}	10^{-2}	10^{-1}	10^{-2}	1	10^{-2}	1	0
BC	10^{-3}	0	10^{-3}	0	1	10^{-1}	1	10^{-2}
IRV	10^{-4}	0	10^{-4}	0	1	10^{-1}	10^{-4}	0
CRV	10^{-4}	0	10^{-4}	0	1	10^{-2}	1	10^{-2}
PS	10^{-1}	0	10^{-1}	0	1	10^{-2}	1	0
PM	0	1	0	10	10^{-1}	10^{-2}	0	0

did not perform so well, so it is good to note that these algorithms apparently do not drag down the performance of the ensemble. The optimal policy receives 7.05 reward per step, so the final policies of all ensembles are optimal or near optimal in virtually all 100 runs.

Table 6.8: Average rewards per step for the stochastic Dyna maze MDP. The average reward is shown for all 50,000 training steps (total online), for the last 5000 training steps (final online) and for 5000 test steps without exploration after training (final offline). For the ensembles results are shown for when only the ensemble is greedy (final offline) and for when both the ensemble and the consisting algorithms are greedy (final greedy). Values are averaged over 100 runs.

	total		**final online**		**final offline**		**final greedy**	
INDIVIDUAL RESULTS								
best	3.50	DQ	4.41	DQ	5.33	Q	—	—
average	2.79	—	3.73	—	5.18	—	—	—
worst	1.70	Acla	2.41	Acla	4.76	Acla	—	—
ENSEMBLE RESULTS								
PV	4.01	0.08	**5.03**	0.04	5.29	0.01	**5.30**	0.01
SAV	**4.03**	0.08	4.98	0.05	5.29	0.01	**5.30**	0.01
DAV	3.79	0.14	4.29	0.10	5.30	0.01	**5.30**	0.01
BC	3.16	0.16	3.67	0.12	5.29	0.01	**5.30**	0.01
IRV	2.85	0.18	3.18	0.13	5.30	0.01	**5.30**	0.01
CRV	3.13	0.16	3.58	0.12	5.30	0.01	**5.30**	0.01
PS	3.83	0.13	4.65	0.09	**5.32**	0.01	**5.30**	0.01
PM	2.58	0.11	3.75	0.10	5.31	0.01	4.11	0.13
best	4.03	**SAV**	5.03	**PV**	5.32	**PS**	5.30	**most**
worst	2.58	PM	3.18	IRV	5.29	PV	4.11	PM
ENSEMBLE PARAMETERS								
	τ_E	τ	τ_E	τ	τ_E	τ	τ_E	τ
PV	10^{-1}	1	10^{-1}	1	1	10^{-2}	1	1
SAV	10^{-1}	1	10^{-1}	1	1	10^{-1}	1	1
DAV	10^{-1}	0	10^{-1}	0	1	1	1	10^{-2}
BC	10^{-1}	0	10^{-1}	100	1	10	1	10
IRV	0	0	10^{-1}	10^{-1}	1	1	1	1
CRV	10^{-1}	0	10^{-1}	0	1	10^{-2}	1	10^{-2}
PS	10^{-1}	1	10^{-1}	1	100	1	1	1
PM	0	0	0	0	10^{-1}	10^{-1}	0	0

6.5.5 Stochastic Dyna Maze

We add 20% random action replacement noise to get noisy transitions. In this stochastic Dyna maze most individual algorithms perform quite close to each other, but the differences between the algorithms are larger than in the deterministic case.

The results for the ensembles are shown in Table 6.8. Although the differences is not significant, plurality voting, policy summation and both approval voting methods all reach a better total online performance than the best individual agent. The results for the other metrics are equally impressive. Apparently many of the ensemble methods are more likely to be as strong as the strongest link, rather than the average or weakest link. As long as a large enough subset of the agents reaches reasonable policies, one or two poorly performing agents do not seem to hinder the performance.

Of the ensembles, the stochastic methods reach the best online performance. The offline performance for all ensembles is so good that we can not distinguish between them, although policy multiplication suffers a decrease in performance when the policies of the individual agents are made greedy. This is due to the vulnerability of this method to 'vetoes' of single agents, as discussed earlier.

6.5.6 Mountain Car

The final test case we consider in this chapter is the discretized version of the mountain car MDP. The state space is divided into 64 separate states by partitioning the position and the velocity of the car into 8 equally sized parts. There are three actions in each state: accelerate left, accelerate right or do nothing. For a more detailed description, we refer to Section 5.8.5.

Instead of just using a learning rate of $n^{-0.8}$, also learning rates of 10^{-1}, 10^{-2}, 10^{-3} and 10^{-4} were allowed. However, all agents were always given the same learning rate and algorithms that store two separate value functions were required to use the same learning rate for both functions. This means that the parameters used for the agents that are part of the ensemble are far from optimal, which implies that the ensembles have an additional disadvantage when compared to the optimized individual algorithms.

The results for the mountain car are shown in Table 6.9. Again, we see that the ensemble methods come close to the performance of the best individual algorithm, even though we severely limited the possibilities of the ensembles by restricting all algorithms that are part of the ensemble to use a single exploration parameter and a single learning parameter. The optimal choice for the learning parameter for all ensembles in almost all cases was 0.1. The only exceptions were the offline final performance of policy summation and policy multiplication and the greedy offline performance of policy multiplication. In these three cases a learning rate of 0.01 was preferred. The final offline result for policy multiplication is better than all other ensemble methods, although the difference with policy summation and stochastic approval voting is not statistically significant. This is interesting, because for the other three metrics policy multiplication is slightly worse than most other ensembles.

Table 6.9: Average number of steps until the goal state is reached in the mountain car MDP (lower is better). The average results are shown for all 100,000 training steps (total online), for the last 10000 training steps (final online) and for 10000 test steps without exploration after training (final offline). For the ensembles results are shown for when only the ensemble is greedy (final offline) and for when both the ensemble and the consisting algorithms are greedy (final greedy). Values are averaged over 10 runs.

	total		**final online**		**final offline**		**final greedy**	
INDIVIDUAL RESULTS								
best	255.7	AC	181.5	Sarsa	136.3	AC	—	—
average	294.9	—	204.5	—	207.4	—	—	—
worst	342.2	DQ	253.5	Acla	246.5	DQ	—	—
ENSEMBLE RESULTS								
PV	270.1	16.4	192.2	8.6	172.1	13.5	166.7	12.6
SAV	257.9	12.4	183.0	8.9	155.0	11.3	155.0	11.2
DAV	268.0	8.7	190.6	11.0	170.1	6.1	174.5	8.9
BC	267.8	8.7	181.2	7.9	162.3	5.7	162.3	5.5
IRV	266.5	10.3	185.3	5.7	166.7	11.6	166.9	11.4
CRV	273.1	19.0	187.2	9.7	167.5	10.5	166.1	9.7
PS	256.6	11.0	187.0	7.3	144.3	21.0	177.0	9.6
PM	289.5	32.3	214.3	18.7	129.5	10.3	271.7	35.2
best	256.6	**PS**	181.2	**BC**	129.5	**PM**	155.0	**SAV**
worst	289.5	PM	214.3	PM	172.1	PV	271.7	PM
ENSEMBLE PARAMETERS								
	τ_E	τ	τ_E	τ	τ_E	τ	τ_E	τ
PV	10^{-2}	0	10^{-3}	10^{-2}	10^{-4}	10^{-2}	10^{-4}	10^{-2}
SAV	0	10^{-2}	0	10^{-2}	0	10^{-4}	0	10^{-4}
DAV	0	0	10^{-5}	1	10^{-4}	1	10^{-1}	0
BC	0	10^{-3}	10^{-1}	1	0	10^{-2}	0	10^{-2}
IRV	10^{-5}	10^{-3}	10^{-2}	0	10^{-4}	10^{-4}	10^{-4}	10^{-4}
CRV	10^{-3}	10^{-2}	0	10^{-4}	10^{-1}	0	10^{-1}	0
PS	10^{-2}	10^{-2}	10^{-2}	10^{-2}	100	1	10^{-1}	10^{-3}
PM	0	10^{-1}	0	10^{-1}	10^{-3}	10^{-3}	1	10^{-3}

If we ignore policy multiplication, all ensembles score approximately equal on all performance metrics. The online results are comparable to the online results for the best individual algorithm. This is promising, because these results were obtained with an Actor Critic algorithm that was allowed to use different learning rates for its actor and for its critic. Recall that this was not allowed in the ensemble as all agents were required to use the same learning rate for all their value functions. In the offline performance, the probability based methods seem to do better than the voting schemes and Actor Critic performs better than all voting schemes. However, also the voting schemes are slightly better than the next best individual algorithm, which was Sarsa with an average number of steps to reach the goal state of 187.4.

No doubt the ensembles can be improved by tuning the agents they consist of individually, but this comes at the computational cost of far more extensive tuning of the parameters. Whether this tuning is worth the trouble is of course dependent on the cost of experimenting on the MDP. If (simulated) experiments on the MDP are cheap, it might be worth to obtain better results by using a fully tuned ensemble. If on the other hand there is less room for tuning the parameters, we have seen that most ensemble methods already reach respectable performance levels when only a very course optimization of the parameters is attempted.

6.6 Discussion and Future Research

The results have shown that most of the ensembles reach reasonably good policies, even when a small number of the individual algorithms they consist of perform poorly. If a single algorithm stands out, as was the case with Double Q-learning in the second grid world, the ensemble seems to benefit from this. The best ensembles performed as good as the second best algorithm on this problem and far better than the average performance of the individual algorithms. No ensemble reached the high performance of Double Q-learning in this settings, but the fact that the performance was better than the average of the constituting algorithms does suggest a potential important application for the ensembles.

Although we attempt to shed some light on this issue in this dissertation, it is often unknown which algorithm will reach good policies the fastest. In some problems, it is important that the performance of the agent is not too poor during training, for instance because this may damage the system it is trying to control. One way to solve this is to take explicit care in the exploration of the agent to avoid dangerous states (Hans et al., 2008). However, our results indicate that ensembles may also be very useful for good online performance, even if some of the individual algorithms are quite poor.

For instance, one could run such an ensemble and record the average re-

wards of each individual algorithm. For instance this can be done by averaging the rewards for each algorithm for the time steps on which the action of the ensemble agrees with the most preferred action by the algorithm. Alternatively, one could sample an action for each agent according to its policy and average the rewards on the steps when the ensemble action agrees with this sampled action. Finally, one could weight the rewards on each step with the probability that each algorithm would select that action and update the running averages with a step size that is dependent on that probability. This way, each algorithm will get a measure of how good the algorithm would perform outside the ensemble, without actually having to try each of them separately.

Alternatively, one could use the General Q-learning framework to update action values for a number of algorithms based on their own policy, rather than on the ensemble policy. Then, the action values give a good indication of the performance of each individual agent. Of course, the agents should be different in some sense to allow for different policies to emerge. Otherwise, the ensemble does not add anything. There are a number of ways these agents could be made to differ. For instance, some algorithms could be more greedy than others. Some algorithms may use a double action value approach, similar to Double Q-learning. And it is even possible to differentiate the discount factor for different algorithms. Then, the action values are no longer directly comparable to determine the relative worth of each algorithm, but it does seem interesting to have an ensemble consisting of both long-term optimizing agents and short-term optimizing agents. Finally, an easy way to obtain different General Q-learning algorithm is to use a different learning rate per agent. All of these ideas make interesting topics for future research into behavior based ensemble methods for reinforcement learning.

An extension of the ideas above leads to weighted ensembles. One could measure some performance metric for each agent and adapt the influence of each agent to this performance metric. Eventually the best performing agents will receive larger weights than poor performing agents. One instance of such a performance metric could be the average reward or the action values corresponding to the agents' policies, as discussed above. Other performance metrics can be devised. For instance, one could take the 'agreeability' of an agent as a metric of its performance: if many other agents agree with the policy or the sampled action of an agent, the agent's weight gets increased, while if less other agents agree with the agent, its weight gets decreased. Of course, to implement such an approach good formalizations of the terms 'more' and 'less' should be considered. This is also left for future research.

Plurality voting was shown to be a good ensemble method and it performs much better than the related policy summation in at least one setting, where it uses it inherent stochasticity to escape from a suboptimal solution. However, if the number of actions is (much) larger than the number of agents in the ensemble, we expect that policy addition or perhaps stochastic approval

voting will perform better. The reason is that we expect that if the number of actions is large, there are many occasions on which non of the agents agree on which action to take. Then, plurality voting essentially chooses at random between all the actions that are selected by each of the agents. In such a case, policy summation and stochastic approval voting can perhaps obtain better results because they use more of the available information about the full selection policies of the individual agents. This is an interesting hypothesis to test in future work.

6.7 Conclusion

Policy based ensembles of reinforcement learning algorithms seem like a promising way to get good policies fast in a wide range of problems. Even though we severely restricted the possible learning parameters of the ensembles, they were often as good as the best individual algorithm that was used to construct the ensembles, and sometimes even better. Most approaches seem very robust to adding one or two poorly performing agents.

As for the comparison of the different ensemble methods: there is not a single methods that stands out in particular. In one problem we saw that the inherent stochasticity of plurality voting and stochastic approval voting helped to break out of a local optimum when the ensemble was made greedy. In other problems we saw more or less equivalent performance of all voting schemes. The policy multiplication method was often much worse, probably because it needs a well tuned exploration parameter and our course tuning did not reach the best results.

If we weight ease of implementation, ease of use and performance, plurality voting seems like a good choice. When the ensemble used this method, good policies were found quite consistently. It was never significantly worse than the similar stochastic approval voting. It was also usually better and never significantly worse than policy summation, which can be viewed as a deterministic variant of plurality voting. However, it was significantly better on one occasion, when the extra stochasticity that is introduced by the action sampling of plurality voting caused the ensemble to break out of a local minimum.

Much more research can be done on behavior based ensembles, but the results have shown at least that the approach is promising. We have seen that often the best ensembles were as good as their strongest link. Often the performance of most ensembles was much better than the average performance of the individually optimized algorithms and in all cases the best ensemble was significantly better than the worst individual agent. This is especially important because we have seen that there is no single reinforcement learning algorithm that performs well on all occasions, so by using ensembles we

can prevent choosing a poor algorithm in problems were it is important that the performance is not too poorly.

CHAPTER 7

CONTINUOUS STATE AND ACTION SPACES

There are many problems for which reinforcement learning can successfully be used, that have large or continuous domains. Many traditional reinforcement learning algorithms assume finite discrete state and action spaces, such as the ones described in the previous chapter. In this chapter, we will discuss a few ways to apply the reinforcement learning algorithms from the previous chapters to problems with continuous state spaces. Additionally, we will discuss reinforcement learning algorithms that can learn good policies in MDPs with continuous action spaces. This includes a discussion on existing actor-critic, policy-gradient and evolutionary algorithms.

After reviewing current reinforcement learning methods that can be used with continuous state and action spaces, we will introduce a new model-free temporal-difference algorithm that can learn online in fully continuous domains. This algorithm is called the continuous actor-critic learning automaton (Cacla). This algorithm is fast and easy to implement and we will see that it can reach good solutions much faster in a pole balancing domain than the current state-of-the-art reinforcement learning algorithms for continuous action spaces.

7.1 Introduction

In a general sense, many learning problems can be interpreted as the problem of finding a mapping of an input set to an output set. In reinforcement learning, the input set is the state space and the output set is the action space or the policy space. When both sets are finite, we can use the algorithms the were outlined in the previous chapters. However, in some cases the state space or the action space can be very large or even continuous.

We start by extending the MDP framework to continuous spaces in Section 7.2. To deal with large or continuous spaces, we will need some function approximation techniques, which we will discuss in Section 7.3. We apply these techniques to reinforcement learning in Section 7.4, where we also discuss the current state of knowledge, including which convergence guarantees can be given when combining reinforcement learning with function approximation and the current state of the art in algorithms for continuous reinforcement learning problems. This includes a discussion on some recent temporal-difference algorithms, policy-gradient algorithms and evolutionary methods. We will discuss how to deal with problems with continuous action

spaces in Section 7.5. There we will also introduce the new continuous action algorithm called Cacla (van Hasselt and Wiering, 2007a; van Hasselt and Wiering, 2009). Section 7.6 shows the results of some experiments with this algorithm. We compare the Cacla algorithm on problems with discrete actions to the discrete algorithms from Chapter 5 and we compare it to the current state of the art on a difficult continuous double pole balancing task. Finally, Section 7.7 concludes this chapter.

7.2 Markov Decision Processes in Continuous Spaces

Much of the discussion on MDPs in Chapter 2 still holds in the continuous case. The main difference is that in this chapter we will consider the state space S will generally be an infinitely large bounded set. More specifically, we will assume the state space is a subset of a possibly multi-dimensional Euclidean space, such that $S \in \mathbb{R}^{D_S}$, where $D_S \in \{1,2,\ldots\}$ is the dimension of the state space. Likewise, when we talk about continuous actions, we assume $A \in \mathbb{R}^{D_A}$, where $D_A \in \{1,2,\ldots\}$ is the dimension of the action space. We will discuss two variants: continuous state MDPs and continuous state action MDPs. In the first case only the state space is continuous, but the action space is finite. In the second case both the state and the action space are continuous. Most of our analysis transfers to MDPs with finite state spaces and continuous action spaces, but we do not cover this case explicitly. Wherever we write 'continuous' the reader can usually also assume the results hold for 'very large finite' spaces. We will mention some results on large finite state spaces from the literature, since the techniques for such problems are very similar to the continuous case.

In this chapter we discuss two general types of function approximation: linear and non-linear function approximation. Linear function approximation is easier to analyze and implement, and there are global convergence guarantees that can not be given when non-linear function approximators are used. In particular, when TD learning is used to estimate the value of a given stationary policy under on-policy updates the value function converges when the feature vectors are linearly independent (Sutton, 1984, 1988). Later it was shown that TD learning also converges when eligibility traces are used and when the features are not linearly independent (Dayan, 1992; Peng, 1993; Dayan and Sejnowski, 1994; Bertsekas and Tsitsiklis, 1996; Tsitsiklis and Van Roy, 1997). More recently, variants of TD learning were proposed that converge also under off-policy updates (Sutton et al., 2008, 2009; Maei and Sutton, 2010). We will shortly discuss these later in Section 7.4.2. For more general information, we refer to the recent books by Bertsekas (2007) and Szepesvári (2010). A limitation of the aforementioned results is that they

apply to the prediction case, whereas we focus mainly on control problems. However, more recently some work has been done to extend these results to the control case. This has led to the Greedy-GQ algorithm, which essentially extends Q-learning to linear function approximation, without the danger of divergence (Maei et al., 2010).

The main drawback of linear function approximation compared to non-linear function approximation is the need for good informative features. These are usually assumed to be hand picked beforehand, which may require quite some domain knowledge in some cases. Even if convergence in the limit to an optimal solution is guaranteed, this solution is only optimal in the sense that it is the best possible linear function of the given features. This means that poor features imply poor solutions. Additionally, while the theoretical guarantees are less convincing, nice empirical results have been obtained by combining reinforcement learning algorithms with non-linear function approximators, such as neural networks (Haykin, 1994; Bishop, 1995, 2006; Ripley, 2008). Examples include Backgammon (Tesauro, 1992, 1994, 1995), robotics (Anderson, 1989; Lin, 1993; Touzet, 1997; Coulom, 2002) and elevator dispatching (Crites and Barto, 1996, 1998). In our experiments, we mostly use neural networks as function approximators.

7.3 Function Approximation

In the previous chapters, we have assumed that values were stored in a table. This allowed us to update each element in that table individually. However, such an approach quickly becomes infeasible if the state and action space are large. Many more general methods to learn a function from data exist and are the topic of active research in the field of machine learning. For some pointers, see the books by Vapnik (1995), Mitchell (1996) and Bishop (2006). Not all these methods are directly applicable to the setting of reinforcement learning. However, the general framework of parametrized function approximation can be used to extend reinforcement learning to problems with large or even infinite spaces.

In machine learning, it is usually assumed that a labeled data set is given that contains a number of inputs with their corresponding outputs. These given outputs are commonly referred to as targets. One can then answer statistical questions about the process that spawned the data. Often it is possible to construct a predictive model—or function—that describes the observed data and additionally can be used to predict the behavior of unseen data.

In reinforcement learning, targets are often non-stationary, because they may depend on an adapting policy or on adapting values of states. Therefore, targets may change during training and not all methods to learn a model or function from machine learning are directly applicable to reinforcement

learning. Nevertheless, many techniques from machine learning can successfully be applied to the reinforcement learning setting, as long as one is careful about the inherent properties of learning in an MDP. In this section we discuss two methods that can be used more in general to update parametrized functions: gradient descent and evolutionary algorithms. These methods have often been used in reinforcement learning with considerable success (Sutton, 1988; Werbos, 1989a,b, 1990; Whitley et al., 1993; Tesauro, 1994, 1995; Moriarty and Miikkulainen, 1996; Moriarty et al., 1999; Whiteson and Stone, 2006; Wierstra et al., 2008; Rückstieß et al., 2010). There exist other general methods for optimization that can be used, such as simulated annealing (Kirkpatrick, 1984) and particle swarm optimization (Kennedy and Eberhart, 1995). These methods share more or less the same properties as evolutionary algorithms in that they are stochastic global optimization techniques, with some sort of a localized search mechanism. We will not discuss these specific methods further in this chapter, but most of the general discussion will also apply to these approaches.

7.3.1 Generalization

We spend a few words on the limitations and advantages of function approximation in general. First, consider the tabular storage of each action value, as assumed in the previous chapters. Storing each value in a separate cell in a table has a clear advantage: since each cell is separate, there can be no interference and the approximation of the value of each action is only dependent on the algorithm, the MDP and the policy. For instance, for a given policy in a deterministic MDP the exact value of every action can be determined with a single Monte Carlo sample for this action and learning rate equal one. However, the strict separation of cells also has a disadvantage. If the state and/or action space is large, it can take a very long time before each action is visited. Furthermore, if the MDP is noisy each action has to be visited numerous times to average out the noise.

To avoid prohibitively slow convergence rates, one would ideally want to generalize over similar states and actions. For this, a generalizing function approximator can be used. There are two general ways a learned function can misinterpret the data that has been obtained in the sense that it generalizes poorly. The first problem occurs is the function approximator is too inflexible. As an extreme example, if we use a single value to represent the value of all possible states, the generalization will in general be poor. Such a function approximator is said to *underfit* the data. The second problem occurs when the function approximator is too flexible. If the function approximator can represent nearly any function and there are many parameters that can be tweaked, there is a risk that the function approximator will represent not only the general structure of the data, but also the noise in the data. The function

approximator is then said to *overfit* the data. In both cases the generalization error on a large portion of the value space can be large.

In reinforcement learning, a function approximator that underfits the data will in many cases learn quickly but will not reach a very good final performance. Dependent on the complexity of the problem and the simplicity of the function approximator, it may not reach reasonable performance at all. Conversely, a very flexible function approximator might overfit the data, especially when little data has yet been observed. In practice, this will usually translate to slow learning, since the estimations for unobserved states will be poor. As more experiences are obtained, a suitably trained flexible function approximator might reach very good final performance levels. In some cases, convergence to the best possible approximation in the limit can be guaranteed. Then, the more flexible the approximator, the better this final approximation will be. However, it may take prohibitively long before reasonable performance is obtained.

7.3.2 Linear Function Approximation

In Chapter 5 we mentioned that many tabular temporal-difference methods can be interpreted as optimizing a Bellman error through a gradient descent update. The idea is that if we can minimize the so called Bellman residual $\|V - \mathcal{T}V\|$, we come closer to the desired fixed point $V = \mathcal{T}V$. Naturally, this also holds when action values are considered instead of state values.

In this chapter we extend the discussion to non-tabular functions. The simplest of non-tabular functions is a linear function. In fact, as discussed in Chapter 5, a table can be interpreted as a linear function. The feature vector is then the same size as the state space and for each state precisely one element of the feature vector is equal to one, while the rest of the elements is equal to zero. More generally, the value function can be approximated by

$$V_t(s) = \vec{\theta}_t^T \phi(s) \quad . \tag{7.1}$$

Here $\vec{\theta}_t$ is the adaptable parameter vector at time t and $\phi(s)$ is the feature vector of state s. We will use K to refer to the number of elements in the feature vector. Since the function is linear in the parameters, we refer to it as a linear function approximator. Linear function approximators are useful since they are better understood than non-linear function approximators. Applied to reinforcement learning, this has led to a number of convergence guarantees, under various additional assumptions (Sutton, 1984, 1988; Dayan, 1992; Peng, 1993; Dayan and Sejnowski, 1994; Bertsekas and Tsitsiklis, 1996; Tsitsiklis and Van Roy, 1997). From a practical point of view, linear approximators are useful because they are simple to implement and fast to compute.

Many problems have large state spaces in which each state can be represented efficiently with a feature vector of limited size. For instance, the

standard cart pole problem that we will consider later in this chapter has continuous state variables and therefore an infinitely large state space. Yet, every state can be represented with a feature vector with four elements. This means that we would need a table of infinite size, but can suffice with a parameter vector with just four elements if we use a linear approximation, as in (7.1).

Of course, this reduction of tunable parameters of the value function comes at a cost. It is obvious that not every possible value function can be represented as a linear combination of the features of the problem. Therefore, our solution is limited to the set of value functions that can be represented with the chosen functional form. Furthermore, the parameters are shared over all states, which implies that an update to these parameters will change the value of unrelated states. This last point is an advantage as well as a disadvantage, since it implies that often sensible values can be obtained for states that have not yet been observed.

Linear function approximation is often used in combination with reinforcement learning, because it is easier to analyze and it has desirable convergence properties. However, the quality of the solution is very dependent on the quality of the features. If the features are not informative enough, sticking to a linear combination of these features may cause the function approximation to be too inflexible. This means that often considerable domain knowledge is needed to construct useful features for a given problem. There are also methods to determine such features automatically, but a discussion of such methods falls outside the scope of this dissertation.

If one does not know beforehand what useful features are for a given problem, it can be beneficial to use non-linear function approximation. A very flexible class of non-linear functions consist of so called neural networks. For extensive introductions we refer for instance to the books by Bishop (1995, 2006). We will use a specific type of neural network in our experiments, which we briefly explain in Section 7.3.6.

7.3.3 Discretize the State Space: Tile Coding

One often used method to find features for a linear function approximator is to simply divide the continuous space into separate segments and then to attach one or more features to each segment. These features are then active if the value of a point in the continuous space is considered that falls into the corresponding segment.

One prominent example of such a discretizing method that is often used in reinforcement learning is tile coding (Watkins, 1989; Lin and Kim, 1991; Sutton, 1996; Santamaria et al., 1997; Sutton and Barto, 1998), which is based on the Cerebellar Model Articulation Controller (CMAC) structure proposed by Albus (1975a,b). In tile coding, the state space is divided into a

number of partitions. For instance, one could use hypercubes such that if $S \subseteq \mathbb{R}^{D_S}$, then one such hypercube would be defined by the Cartesian product $H_k = [x_{k1}, y_{k1})] \times \ldots \times [x_{kD_S}, y_{kD_S}]$, where $x_k = \{x_{k1}, \ldots, x_{kD_S}\}$ is a vector containing the lower bounds of the partition and likewise $y_k = \{y_{k1}, \ldots, y_{kD_S}\}$ contains the upper bounds. Then, a feature $\phi_k(s) \in \vec{\phi}(s)$ corresponding to this partition is equal to one when $s \in H_k$ and equal to zero otherwise.

In general, if one wants to use a maximum of $K = MN$ different features, one can construct a single partitioning with K partitions to obtain an equally large feature vector of size K for each state. An alternative is to use M partitionings with N partitions each. The advantage of using multiple non-overlapping partitionings, is that then there are up to $\binom{MN}{M} \in O(N^M)$ different feature vectors possible, compared to just MN different feature vectors in the single case. It must be noted that the number $\binom{MN}{M}$ is an upper bound. In general certain combinations of active features may not be possible and number of different feature vectors will be smaller. However, often this number will still be far larger than the MN different features that are obtained with a single partitioning.

7.3.4 Issues with Discretization

One potential problem with discretizing methods such as tile coding is that the resulting function that maps states into features is not injective. In other words, $\phi(s) = \phi(s')$ does not imply that $s = s'$. This means that in fact, the resulting feature space MDP is partially observable and one could consider using an algorithm that is explicitly designed to work on POMDPs. Many good results have been obtained with tile coding, but the discretization and the resulting loss of the Markov property do imply that none of the convergence proofs for linear function approximation in MDPs are guaranteed to hold in this case. This holds for any function approximation that uses a feature space that is not an injective function of the Markov state space.

Another practical problem with tile coding and similar methods is related to the learning rate. The tile coding update can be summarized as

$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T + \beta_t(s_t)\delta_t\phi^T(s_t) \ , \tag{7.2}$$

where $\beta_t(s_t) \in [0, 1]$ is a learning rate and δ_t is the error for the current state. This can be a temporal-difference error, the difference between the current value and a Monte Carlo sample, or any other relevant error. A derivation and explanation of this update and variants thereof are given below, in Sections 7.3.7 and 7.4.2.

If we look at the update to the value that results from (7.2), we get

$$\begin{aligned} V_{t+1}(s_t) &= \vec{\theta}_{t+1}^T \phi(s_t) \\ &= \left(\vec{\theta}_t^T + \beta_t(s_t)\delta_t \vec{\phi}^T(s_t)\right)\phi(s_t) \\ &= \vec{\theta}_t^T \phi(s_t) + \beta_t(s_t)\delta_t \vec{\phi}^T(s_t)\phi(s_t) \\ &= V_t(s_t) + \beta_t(s_t)\vec{\phi}^T(s_t)\phi(s_t)\delta_t \ . \end{aligned}$$

In other words, the effective learning rate on the values is equal to

$$\beta_t(s_t)\vec{\phi}^T(s_t)\phi(s_t) \ . \tag{7.3}$$

In tile coding, $\vec{\phi}^T(s_t)\phi(s_t)$ will be equal to the number of tilings M. This means that the effective learning rate on the value function is larger than intended and can be larger than one. This can cause problems, such as divergence of the parameters.

Of course, these problems are easily prevented, for instance by scaling the learning rate by dividing it by M. When decreasing learning rates are used, for instance because they adhere to the Robbins-Munro conditions, divergence is also prevented because the learning rates will only be larger than one for a finite amount of time. However, the convergence may still be slowed if the learning rate is not scaled to prevent large effective learning rates.

This issue with the learning rates can occur for any feature space and linear function approximation, since then the effective learning rates in (7.3) are used for the update to the value function. This indicates that it can be a good idea to scale the learning rate appropriately, by using

$$\tilde{\beta}_t(s_t) = \frac{\beta_t(s_t)}{\|\vec{\phi}(s_t)\|^2} \ ,$$

where $\tilde{\beta}_t(s_t)$ is the scaled learning rate and $\|\vec{\phi}(s_t)\|$ is the length of the feature vector.[1] This scaled learning rate can prevent unintended small as well as unintended large updates to the values. A similar trick will appear below when we explain policy-gradient algorithms, where it is suggested to adapt the learning rate with a similar procedure to obtain a so called natural gradient update (Amari, 1998).

A final drawback of discretization is that it introduces discontinuities in the function. If the state changes a small amount, the approximated value may change a fairly large amount if the two states fall into different segments of the state space.

[1] One can safely define $\tilde{\beta}_t(s_t) = 0$ if $\|\vec{\phi}(s_t)\| = 0$, since in that case update (7.2) would not change the parameters anyway.

7.3.5 Non-linear Function Approximation

Because of the potential difficulty of finding good features for a linear approximator and the aforementioned issues with discretization of the state space, we discuss non-linear function approximation. This type of function approximation has the problem that in general it can get stuck in local optima, but in general more different functions can be represented and therefore the solution that is found can be of better quality than the more inflexible linear function approximators.

In a non-linear function approximator, the value function is usually represented by some predetermined parametrized function, such that

$$V_t(s) = f(\vec{\theta}_t, \vec{\phi}(s)) \ . \tag{7.4}$$

Here $\vec{\theta}_t$ is the parameter vector that is not necessarily of the same size as the feature vector. For instance, f may be a neural network and $\vec{\theta}_t$ is a vector with all the weights in the neural network at time t. We discuss a multilayer perceptron neural network in the next section. Note that we assume that f is fixed and that the value of the state is dependent on the state through the feature vector $\vec{\phi}(s)$ and on the time step through $\vec{\theta}_t$.

Because the mapping from states to features is often non-linear, many linear methods such as tile coding are non-linear when one considers the full mapping from states to values. In this view, one way to interpret the difference between linear and non-linear approximation is that in the former the non-linear part of the mapping is assumed to be fixed, while in the latter this part of the function can also be parametrized and adapted during learning.

Below, we explain how to use gradient descent and evolutionary algorithms to learn the values of the adaptable parameters. First we explain briefly what a neural network is. In the context of neural networks gradient descent is often referred to as backpropagation (Bryson and Ho, 1969; Werbos, 1974; Rumelhart et al., 1986). In essence, backpropagation is an algorithm that uses the chain rule and the layer structure of the networks to efficiently calculate the derivative of the network's output to all the parameters. However, the principle of gradient descent can be applied to any differential non-linear function.

7.3.6 Non-learning Functions: The Multilayer Perceptron

A multilayer perceptron is a specific type of neural network that consists of a set of weight matrices W_l with $l \in \{1,2,\ldots,L\}$ and a set of activation functions f_l with $l \in \{0,1,2,\ldots,L\}$. The number L is the number of layers in the network.[2] The weight matrices are the parameters of the function that can

[2]Some authors count the input and the output layer of the network and therefore report $L+1$ as the number of layers. Others only count the number of hidden layers and use $L-1$.

be tuned, while the activation functions are used to give the function desirable properties, such as being smooth with easily derivable derivatives. Each weight matrix is of size $(N_l + 1) \times N_{l+1}$, where N_1 equals the size of the input vector and N_{L+1} is the size of the output vector. The number of rows is $N_l + 1$ rather than N_l to account for a bias node in each layer. For instance, the input vector might be a feature vector of a state and the output might be the value function. Then we would have $N_1 = F$ and $N_{L+1} = 1$. The number of weight matrices and the values of N_l for $2 \leq l \leq L$ are determined beforehand as meta-parameters. All the layers except for the input and the output are called hidden layers.

A convenient way to describe a neural network is by defining operators $\mathscr{L}_l$, which take a vector of size N_l as their input and output a vector of size N_{l+1}. These operators are defined by

$$\mathscr{L}_l x = f_{l+1}(W_l^T \vec{x}+) \ ,$$

where $\vec{x}$ is a vector of size N_l and $\vec{x}+$ is an augmented version of $\vec{x}$ with an additional bias node, defined as

$$\text{if} \quad \vec{x} = (x_1, \ldots, x_{N_l})^T \quad \text{then} \quad \vec{x}+ = (x_1, \ldots, x_{N_l}, 1)^T \ .$$

We assume the activation functions $f_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l}$ operate elementwise, such that $f_l = (f_{l,1}, \ldots, f_{l,N_l})^T$ and $f_l(\vec{x}) = (f_{l,1}(x_1), \ldots, f_{l,N_l}(x_{N_l}))^T$. Furthermore, we assume all the functions in a layer are equal, such that $f_{l,i} = f_{l,j}$ for all $1 \leq i < j \leq N_l$. Then, with a slight abuse of notation we use f_l to refer both to the vector-based function from $\mathbb{R}^{N_l}$ to $\mathbb{R}^{N_l}$ and to the component-based function from $\mathbb{R}$ to $\mathbb{R}$. Then, $(f_l(\vec{y}))_i = f_l(y_i)$, where $\vec{y}$ is an arbitrary vector and $(\cdot)_i$ is the i^{th} element of a vector. Usually, we will assume that any necessary preprocessing to the input of the network is conducted before presentation of the input to the network, such that without loss of generality we can assume that f_0 is the identity function. The functional form of the neural network can then be described by

$$\text{NeuralNetwork}(\vec{x}) = \mathscr{L}_L \ldots \mathscr{L}_1 \vec{x} \ . \tag{7.5}$$

The activation functions play an important part in the usefulness of the network. In this chapter, we will only use a single hidden layer, such that there are two weights matrices and $L = 2$. We always use the identity function as the activation of both the input and output layer, such that $f_0(x) = f_2(x) = x$, for any $x \in \mathbb{R}$. The hidden layer has a non-linear activation function, known as the tanh function, which is defined by

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \ .$$

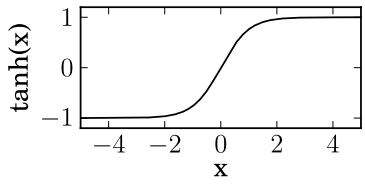

Figure 7.1: A plot of the tanh function.

This function can be interpreted as a smoothed threshold function that is almost equal to one for inputs larger than four and almost equal to minus one for inputs smaller than minus four. The function is shown in Figure 7.1. Apart from the smooth threshold shape and the conveniently bounded range, this function is additionally used because its derivative is very easy to compute:

$$\frac{\partial}{\partial x}\tanh(x) = 1 - \tanh^2(x) \ .$$

In summary, the neural networks we use are equivalent to the following function

$$\text{NeuralNetwork}(\vec{x}) = \begin{pmatrix} w_{2(H+1)1} + \sum_{h=1}^{H} w_{2h1} \tanh\left(w_{1(I+1)h} + \sum_{i=1}^{I} w_{1ih} x_i\right) \\ \vdots \\ w_{2(H+1)O} + \sum_{h=1}^{H} w_{2hO} \tanh\left(w_{1(I+1)h} + \sum_{i=1}^{I} w_{1ih} x_i\right) \end{pmatrix} ,$$

where w_{lij} represents the element in row i and column j of matrix W_l. Here I is the number of input nodes, H is the number of hidden nodes and O is the number of output nodes. The size of the matrices is then $I + 1 \times H$ for W_1 and $H + 1 \times O$ for W_2. The weights $w_{2(H+1)o)}$ and $w_{1(I+1)h}$ are the so called bias weights corresponding to output o and hidden node h, respectively. The matrices W_1 and W_2 store the adaptable parameters that we will attempt to learn using one of the methods below. That means that such a neural network has a total of $(I + O + 1)H + O$ adjustable parameters. The value of I and O is given by the function we want to approximate. The value of H should be chosen, where 'high' values result a flexible function that is slightly slower to use and that has a higher chance of overfitting observed data, whereas 'low' values result in a function that is less flexible, quick to use and that generalizes well to unseen data. We observed little effect of the exact value of H to our results and used 40 hidden nodes (i.e., H = 40) throughout the experiments.

7.3.7 Gradient Descent

Most of the methods in this chapter use some form of a gradient descent update. We explain the gradient descent algorithm in general and discuss its application to reinforcement learning. We have discussed this partially in Section 5.2. Here we summarize the discussion there and extend it to function approximation.

A gradient descent update follows the direction of the negative gradient over some metric to be minimized. The idea is that the gradient of a parametrized function to its parameters points in the direction in which the function increases, according to a first-order Taylor expansion. Under the assumption that the function is smooth, changing the parameters an infinitesimally small amount in the direction of the negative gradient should then result in a new function that has an infinitesimally smaller value at the given input. The assumption behind gradient descent algorithms is that this also usually holds when a larger step in this direction is taken.

Any update that follows the direction of the negative gradient is called a gradient descent update. In the definition below, we state this more precisely. In this definition, for simplicity we assume real-valued functions.

Definition 7.1 (Gradient Descent)**.** *Let $f : \mathbb{R}^P \times \mathbb{R}^N \to \mathbb{R}^M$ denote a parametrized function. We define $f_t : \mathbb{R}^N \to \mathbb{R}^M$ to be the function that corresponds to a parameter vector $\vec{\theta}_t \in \mathbb{R}^P$, such that $\forall x \in \mathbb{R}^N : f_t(x) = f(\vec{\theta}_t, x)$. In a gradient descent update, the parameters of f_t are updated so that the target of the update lies in the direction of the negative gradient. The update to the parameters then is*

$$\vec{\theta}^T_{t+1} = \vec{\theta}^T_t - \eta_t \nabla_\theta f_t(x) \ , \tag{7.6}$$

where $\eta_t \in [0,1]$ is a step size and $x \in X$ is some input. In (7.6), $\nabla_\theta f_t(x)$ is the gradient to the parameters, which is a row vector defined by

$$\nabla_\theta f_t(x) = \left(\frac{\partial}{\partial \theta_{t,1}} f_t(x), \ldots, \frac{\partial}{\partial \theta_{t,P}} f_t(x) \right)$$

Then, we can obtain a new function $f_{t+1} : X \to Y$, which is defined by $\forall x \in X$: $f_{t+1}(x) = f(\vec{\theta}_{t+1}, x)$.

It can be seen from Definition 7.1 that a gradient descent update is always defined for a given input, which is denoted by x in the definition. The update (7.6) is defined on the whole parameter vector. It is of course also possible to update a subset of parameters at a time. Then, one will use an elementwise gradient descent update, defined by

$$\theta_{t+1,p} = \theta_{t,p} - \eta_t \frac{\partial}{\partial \theta_{t,p}} f_t(x) \ .$$

It is possible to generalize this update further, for instance by making the step size η_t dependent on the input x, or perhaps on the parameter element $\theta_{t,p}$.

Usually, the function that we want to minimize through a gradient descent procedure is some error measure. The goal is then to minimize this error as much as possible. If instead we want to maximize rather than minimize a function, the minus in update (7.6) is replaced with a plus and the procedure is called gradient ascent.

Many applications of gradient descent involve a fixed set of inputs and target outputs to which the parameters of the function should be adapted. For instance, consider a data set of n input-output pairs $X = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \mathbb{R}^{N_x}$ and $y_i \in \mathbb{R}^{N_y}$. Suppose we want to adapt the parameters of a function $g : \Theta \times \mathbb{R}^{N_x} \rightarrow \mathbb{R}^{N_y}$ in order to minimize the difference between the outputs of this function and the given outputs for the corresponding inputs. Further assume we consider all input-output pairs as equally important and we measure the Euclidean distance between the output of the function and the desired output of the function. This means we want to minimize the following error function

$$\mathbf{E}(\vec{\theta}, X) = \frac{1}{2} \sum_{i=1}^{n} \Big(y_i - g(\vec{\theta}, x_i) \Big)^2 \quad .$$

The factor 1/2 is included only for convenience. We can then iteratively lower the error by using a gradient descent procedure, which results in

$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T - \eta_t \nabla_{\theta_t} \mathbf{E}(\vec{\theta}_t, x) = \vec{\theta}_t^T + \eta_t \sum_{i=1}^{n} \Big(y_i - g(\vec{\theta}, x_i) \Big) \nabla_{\theta_t} g(\vec{\theta}, x_i) \quad . \tag{7.7}$$

This approach has a number of disadvantages. For instance, it can be slow to converge and it can get stuck in a local optimum. Also, there is a probability that the trained function overfits the training data. For these problems many solutions have been proposed (see e.g., Bishop, 2006). Most of these solutions assume the whole training data set is available and fixed. Another common assumption for these improvements over the standard iterative gradient descent approach is that the data is independently and identically distributed. These assumptions do not normally hold in reinforcement learning.

Although it too can be slow to converge, one simple way to partially address the issue of local optima is to use stochastic iterative gradient descent. This algorithm uses only one data point at a time. This may then prevent getting stuck in a local optimum, since each update does not necessarily move the function into the direction of the gradient. It can not be guaranteed in general that the global optimum is found, but convergence to a local optimum of the error function can be guaranteed if the data points are chosen at random and the step size is chosen according to the Robbins-Munro conditions.

In other words, if $\sum_t \eta_t = \infty$ and $\sum_t (\eta_t)^2 < \infty$, we can guarantee that

$$\lim_{t \to \infty} \mathbf{E}(\vec{\theta}_t, x) = \min_{\vec{\theta} \in \mathscr{R}} \mathbf{E}(\vec{\theta}, x) \ ,$$

where $\mathscr{R}$ is a subset of the full parameter space. Unfortunately, this subset may not span the full parameter space, which is why this is called a local optimum.

There is some indication that stochastic gradient descent converges faster than batch gradient descent in many cases (Wilson and Martinez, 2003). An additional advantage of stochastic gradient descent over batch learning is that it is straightforward to extend online stochastic gradient descent to an adapting target function. In other words, one does not have to have an independent and identically distributed training set. These features make online gradient methods very suitable for reinforcement learning. We will discuss the application of gradient descent methods to reinforcement learning in Section 7.4.

7.3.8 Evolutionary Algorithms

Another way to optimize a parameter vector for a given optimization problem is through evolutionary algorithms (Holland, 1962; Rechenberg, 1971; Holland, 1975; Schwefel, 1977; Davis, 1991; Bäck and Schwefel, 1993). We will not give a full introduction to this method. For this we refer to introductory texts, such as the books by Bäck (1996) and Eiben and Smith (2003). We merely give a short overview of how such an algorithm can work.

In an evolutionary algorithm, a population of possible solutions is constructed, for instance by creating a number of random parameter vectors. Then, individuals from this population are selected for reproduction and mixed, after which the resulting offspring is mutated slightly to maintain diversity in the population. The offspring then usually replaces individuals in the existing population. The idea is to specify some fitness function on the individuals, such that both the selection of individuals for reproduction and the selection for which members of the population to discard in favor of the new offspring is guided by this fitness. By convention, higher fitness improves the probability to be selected. This procedure is then repeated until an individual with a high enough fitness has been produced. One iteration of the cycle of selection, reproduction, mutation and replacement is by convention usually called a generation.

Evolutionary algorithms therefore use two methods to adapt the solutions. The reproduction can be seen as a heuristic randomized search, that assumes that two partial solutions for a problem can be combined to form a better new solution. This is often indeed possible, but it does require that the individual solutions are mixed in a way that is unlikely to be too destructive. Otherwise,

the offspring will often be worse than the individuals that were used to create it and the progress towards better solutions will be slow. The mutation can be seen as random local search in the solution space. For completeness, we note that in an evolutionary algorithm either the reproduction or the mutation phase can be missing. If the reproduction phase is missing, the algorithm is said to be asexual and the only interaction between the individuals in the population occurs due to the selection on which individuals to keep in the next generation. We will discuss an application of evolutionary algorithms to reinforcement learning in Section 7.4.4.

7.3.9 Comparing Gradient Descent and Evolutionary Algorithms

Evolutionary algorithms provide a very general framework for optimization. The main requirement is a way to map an individual into a fitness. This is similar to the requirement of an error function in gradient descent. However, because of the importance of the reproduction step, care must be taken in the representation of the individuals and the definition of the reproduction step itself. This is non-trivial for many problems.

Evolutionary algorithms are often gradient-free, removing the need to calculate gradients. This is especially useful if gradients are hard or impossible to calculate for the used fitness function. As such, many evolutionary algorithms can be classified as a zeroth order stochastic search method, while online gradient descent as we discussed it is a first order stochastic search method. In practice, this means it is sometimes easier to define a meaningful fitness function for a problem then it is to construct a smooth, differential error function for gradient descent.

Furthermore, suitably tuned evolutionary algorithms can quite often escape from local maxima of the fitness function. This happens because more than one solution is stored in the population and the adaptations to the individuals through reproduction and mutation can be large. However, as in gradient descent the likelihood of ending up in a local optimum is dependent on the representation of the solutions and the characteristics of the problem. Other factors that can have a large influence on the convergence of an evolutionary algorithm include the definition of the fitness function and the precise methods used to reproduce and mutate the population.

For a given functional form gradient descent only uses a single step size meta-parameter. Evolutionary algorithms often use meta-parameters to guide the selection of individuals to reproduce, the reproduction itself, the mutation and the selection of which individuals to keep in the population. Although in many cases sensible settings exist that work well for many settings (for instance see the papers in Lobo et al., 2007), this does imply that it may be difficult to find meta-parameters that solve any particular problem efficiently.

Another important difference between the two approaches is that gradient descent usually stores only a single solution that is updated each step, while evolutionary algorithms store a whole population of solutions. Because of this and the random nature of the mutation and reproduction, evolutionary algorithms can take much longer to converge on some problems. However, extensive empirical comparisons of the two approaches seem rare.

Below we will discuss how both optimization approaches can be used in reinforcement learning to find good policies of behavior in MDPs.

7.4 Approximate Reinforcement Learning

In this section we apply the techniques described above to the reinforcement learning settings. We discuss some of the current state of the art in reinforcement learning in continuous domains.

7.4.1 Projected Bellman Equation

To be able to reason more formally about the function approximation, we introduce the concept of projections. A projection Π for a certain function approximator is a mapping of the value function space $\mathbb{R}^S$ to the space that can be represented with that function approximator. We use $\mathscr{F} \subseteq \mathbb{R}^S$ to denotes this space of representable functions. Intuitively, if $\mathscr{F}$ contains a large subset of $\mathbb{R}^S$, the function is flexible and can accurately approximate many value functions. However, it may be prone to overfitting of the perceived data and it may be slow to update since usually a more flexible function requires more tunable parameters. Likewise, if $\mathscr{F}$ is small compared to $\mathbb{R}^S$, the function is not so flexible and may not be able to approximate certain functions well, although it will probably generalize quickly.

Formally, the projection Π is an operator that maps a function to the closest representable function, under a certain norm. We will assume each function has a parameter vector $\vec{\theta} = \{\theta_1, \ldots, \theta_K\}$ of size K that can be adjusted during training. From here on further, we denote such parametrized value functions by V_θ or Q_θ, such that for instance $V_{\theta_t}(s)$ gives the current estimate for the value of state s. The space of representable state value functions is then given by

$$\mathscr{F} = \left\{ V_\theta | \vec{\theta} \in \mathbb{R}^K \right\} \ .$$

The projection for this function approximator under a weighted norm is defined by

$$\|V - \Pi V\|_w = \min_{v \in \mathscr{F}} \|V - v\|_w = \min_{\vec{\theta}} \|V - V_\theta\|_w \ ,$$

where $\|\cdot\|_w$ is a weighted norm. We will assume the norm is quadratic, such that

$$\|V - V_\theta\|_w^2 = \int_{s\in S} w(s)\Big(V(x) - V_\theta(s)\Big)^2 ds \ ,$$

or, if the state space is finite

$$\|V - V_\theta\|_w^2 = \sum_{s\in S} w(s)\Big(V(s) - V_\theta(s)\Big)^2 \ .$$

This means that any projection is defined by two aspects: the functional form of the function approximator and the weighted norm.

It is often not possible to find a parameter vector that fulfills the Bellman equation $V_\theta = \mathcal{T}^\pi V_\theta$, or the Bellman optimality equation $V_\theta = \mathcal{T}^* V_\theta$ for the whole state space. Rather, the best we can hope for is a parameter vector that fulfills

$$V_\theta = \Pi \mathcal{T} V_\theta \ , \tag{7.8}$$

where $\mathcal{T} = \mathcal{T}^\pi$ or $\mathcal{T} = \mathcal{T}^*$. This equation is called the projected Bellman equation. In equation (7.8), $\Pi : \mathbb{R}^S \to \mathcal{F}$ projects the outcome of the one step Bellman operator $\mathcal{T} V_\theta$ back to the space that is representable by the function approximation.

In some cases, it is possible to give a closed form expression for the projection. For instance, consider a large finite state space with size $|S|$ where the states are enumerated from 1 to $|S|$, such that with some abuse of notation we assume that s denotes the number of the state rather than the state itself. Assume a linear function with a parameter vector $\vec{\theta}$ of size $K \ll |S|$ that maps the features of each state $\phi(s) \in \mathbb{R}^K$ into a value with $V_\theta(s) = \vec{\theta}^T \vec{\phi}(s)$. Further, assume the expected probability of sampling each state is stored in the diagonals of the $|S| \times |S|$ matrix D, such that $d_{ss'}$ is the element in row s and column s' of D, $d_{ss'} = 0$ if $s \neq s'$ and $d_{ss} = P(s_t = s)$ for an arbitrary time step t (Tsitsiklis and Van Roy, 1997). For simplicity, we assume the states are always sampled according to these probabilities and therefore we do not have to consider a separate distribution for the initial states. Finally, assume the $|S| \times K$ matrix Φ holds the feature vectors for all states in its rows, such that $V_\theta = \Phi\theta$ and $V_\theta(s) = \Phi_s \theta = \theta^T \vec{\phi}(s)$. Then, the projection operator can be represented by the $|S| \times |S|$ matrix with the following definition:

$$\Pi = \Phi \Big(\Phi^T D \Phi\Big)^{-1} \Phi^T D \ . \tag{7.9}$$

The inverse exists if the features are linearly independent, such that Φ has rank K.

Note that in general $\Pi V_\theta = \Pi\Phi\theta = \Phi\theta = V_\theta$, but $\Pi\mathcal{T} V_\theta \neq \mathcal{T} V_\theta$, unless $\mathcal{T} V_\theta$ can be expressed as a linear function of the feature vectors. The existence of a projection matrix as defined in (7.9) is used in the analysis and in

the derivation of several algorithms (Tsitsiklis and Van Roy, 1997; Nedić and Bertsekas, 2003; Bertsekas et al., 2004; Sutton et al., 2008, 2009; Maei and Sutton, 2010).

7.4.2 Gradient Temporal-Difference Learning

We generalize the TD learning update (2.18) from Section 2.3.4 to the linear function approximation case. In Chapter 5 we argued that the tabular TD update can be seen as a variant of linear function approximation if we assume there are precisely as many features as there are states, in every state exactly one feature is active and in no two different states the same feature is active. We repeat the update for clarity

$$V_{t+1}(s_t) = V_t(s_t) + \beta_t(s_t)\delta_t \ ,$$

where

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \ .$$

The tabular TD learning update can be interpreted as following a stochastic gradient descent update on the one step temporal-difference error, defined as

$$\mathbf{E}_t(s_t) = \frac{1}{2}\left(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\right)^2 = \frac{1}{2}(\delta_t)^2 \ . \tag{7.10}$$

If $V_t = V_\theta$ is a parametrized function with parameter vector $\vec{\theta}$, the negative gradient with respect to the parameters is given by

$$-\nabla_\theta \mathbf{E}_t(s_t,\theta) = -\left(r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)\right)\nabla_\theta\left(r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)\right) \ ,$$

where we stress the dependence of the error on the parameter vector by writing it as a function thereof. Naturally, the error is also dependent on the MDP and the policy, but we do not specify these dependencies explicitly to avoid cluttering the notation.

A direct implementation of gradient descent based on the error in (7.10) would adapt the parameter to move $V_\theta(s)$ closer to $r_{t+1} + \gamma V_\theta(s_{t+1})$ as desired, but would also move $\gamma V_\theta(s_{t+1})$ closer to $V_\theta(s_t) - r_{t+1}$. This would imply that we move the value of the next state $V_{\theta_t}(s_{t+1})$ in such a way that the error in the current state becomes lower. Such an algorithm is called a residual gradient algorithm (Baird, 1995). Alternatively, we can interpret $r_{t+1} + \gamma V_\theta(s_{t+1})$ as a stochastic target value that is independent on the parameter vector. Then, the negative gradient is (Sutton, 1984, 1988)

$$-\nabla_\theta \mathbf{E}_t(s_t,\theta) = \left(r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)\right)\nabla_\theta V_\theta(s_t) \ .$$

This implies the parameters can be updated as

$$\vec{\theta}^T_{t+1} = \vec{\theta}^T_t + \beta_t(s_t)\delta_t \nabla_{\theta_t} V_{\theta_t}(s_t) \ . \tag{7.11}$$

This is the conventional TD learning update and it usually converges faster than ordinary the residual gradient update above (Gordon, 1995, 1999). However, because we ignore the parameters in the next state, this is not a real gradient descent algorithm. We will get back to this later.

For linear function approximation, for any θ we have $\nabla_\theta V_\theta(s_t) = \vec{\phi}(s_t)$ and we obtain the same update as was shown earlier for tile coding in (7.2). Similar updates can easily be created for action value algorithms, such as the ones discussed in Chapter 5.

We can incorporate accumulating eligibility traces with trace parameter λ with the following two equations (Sutton, 1984, 1988):

$$\vec{e}_{t+1}^T = \lambda\gamma\vec{e}_t^T + \nabla_{\theta_t} V_{\theta_t}(s_t)$$
$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T + \beta_t(s_t)\delta_t\vec{e}_{t+1} \ ,$$

where the trace vector $\vec{e}$ has a number of elements equal to the number of features. Replacing traces are somewhat less straightforward, although it seems sensible to use the suggestion by Främling (2007):

$$\vec{e}_{t+1}^T = \max(\lambda\gamma\vec{e}_t^T, \nabla_{\theta_t} V_{\theta_t}(s_t)) \ ,$$

since this corresponds nicely to the common practice when tile coding is used. Also, this update reduces to the conventional replacing traces update when a table is used as function approximator. However, as far as we know a good theoretical justification for this generalization of replacing traces is still lacking.

Update (7.11) can not be proven to converge if off-policy updates are used. This holds for any temporal-difference method with $\lambda < 1$ and when we use linear function approximation (Baird, 1995) or non-linear function approximation (Tsitsiklis and Van Roy, 1996). In other words, if we sample transitions from some simulated MDP and feed these experiences to the algorithm according to a distribution that does not comply completely to the state visit probabilities that would occur under a fixed behavior policy, the parameters of the function may diverge.

Recently, a class of algorithms has been proposed to deal with this issue (Sutton et al., 2008, 2009; Maei et al., 2009; Maei and Sutton, 2010). The main idea is to perform a stochastic gradient descent update, but on a different error function. This error function is not defined as the current temporal difference, but as the one-step projected temporal difference:

$$\mathbf{E}(\theta) = \frac{1}{2}\|V_\theta - \Pi\mathcal{T}V_\theta\|_D^2 \ . \tag{7.12}$$

Note that in contrast with the error in (7.10), this error is not explicitly dependent on the time step. It is dependent on the MDP, the backup operator

and of course on the parameter vector. We stress this last dependence by writing the error as a function of the parameters. The norm in (7.12) is weighted according to the steady-state probabilities that are stored on the diagonal of matrix D, as described in Section 7.4.1.

If we can minimize the error (7.12), we have reached the fixed point in (7.8). This implies that we can not do better for the given function approximation. If can be shown that the error in (7.12) can be rewritten as

$$\mathbf{E}(\theta) = \frac{1}{2}\left(E\left\{\delta \nabla_\theta V_\theta(s)\right\}\right)^T \left(E\left\{\nabla_\theta V_\theta(s) \nabla_\theta^T V_\theta(s)\right\}\right)^{-1} E\left\{\delta \nabla_\theta V_\theta(s)\right\} \quad , \tag{7.13}$$

where it is assumed that the inverse exists (Maei et al., 2009). The expectations are taken over the steady state probabilities in D. The error function can therefore be interpreted as the product of multiple expected values. These expected values can not be sampled from a single experience, because then the samples would be correlated. This can be solved by updating two parameter vectors instead of one. To see how this works, we use the shorthands $\phi = \vec{\phi}(s_t)$ and $\phi' = \vec{\phi}(s_{t+1})$ and we assume a linear function approximation. Then $\nabla_\theta V_\theta(s_t) = \phi$ and we can calculated the negative gradient of (7.13) as follows:

$$\begin{aligned} -\nabla_\theta \mathbf{E} &= E\left\{(\phi - \gamma\phi')\phi^T\right\}\left(E\left\{\phi\phi^T\right\}\right)^{-1} E\left\{\delta\phi\right\} \\ &\approx E\left\{(\phi - \gamma\phi')\phi^T\right\}\vec{w} \quad . \end{aligned}$$

The parameter vector $\vec{w}_t$ can be updated with the stochastic update

$$\vec{w}_{t+1}^T = \vec{w}_t^T + \eta_t(s_t)\left(\delta_t - \vec{w}_t^T\vec{\phi}(s_t)\right)\vec{\phi}^T(s_t) \quad ,$$

where $\eta_t(s_t) \in [0,1]$ is an additional step size parameter. It can be verified that $\vec{w}_t$ then approximates $\left(E\left\{\phi\phi^T\right\}\right)^{-1} E\left\{\delta_t\phi\right\}$, as required. This means there is only one expected value left to approximate, which can be done with a single sample. This leads to the update

$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T + \beta_t(s_t)\left(\vec{\phi}^T(s_t) - \gamma\vec{\phi}^T(s_{t+1})\right)\left(\vec{w}_t^T\vec{\phi}(s_t)\right) \quad ,$$

which is called the GTD2 (Gradient Temporal-Difference Learning, version 2) algorithm (Sutton et al., 2009). One can also write the gradient in a slightly different manner to obtain the similar TDC algorithm, which is defined as:

$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T + \beta_t(s_t)\delta_t\vec{\phi}^T(s_t) - \beta_t(s_t)\gamma\left(\vec{w}_t^T\vec{\phi}(s_t)\right)\vec{\phi}^T(s_{t+1}) \quad ,$$

where $\vec{w}_t$ is updated as above. This algorithm is named TD with gradient correction (TDC), because the update to the primary parameter vector is equal to the one we discussed before and is shown in (7.11), expect for a correction

term. This term prevents divergence of the parameters. Both GTD2 and TDC can be shown to converge to the projected fixed point in the limit when β_t and η_t follow the normal Robbins-Munro conditions and for GTD2 $\eta_t = k\beta_t$ for some constant $k \in [0, \infty)$, while for TDC $\lim_{t\to\infty} \beta_t/\eta_t = 0$ (Sutton et al., 2009). When non-linear smooth function approximators are used, it can be guaranteed that algorithms that are constructed along these lines reach local optima (Maei et al., 2009). The resulting updates in the non-linear are similar to the ones above with the addition of another correction term.

The aforementioned updates can be extended to a form of Q-learning in order to learn action values with eligibility traces. The resulting GQ(λ) algorithm is off-policy, which means it converges to the prediction of the value of a given estimation policy, even when the algorithm follows a different behavior policy (Maei and Sutton, 2010). Finally, the methods can be extended to solve the control problem (Maei et al., 2010), although at the present time it is not yet known how well the resulting Greedy-GQ algorithm performs in practice.

Although these theoretic insights and the resulting algorithms are very promising, we note that in practice often the old TD updates—such as (7.11)—perform equally well. Apparently, most problems do not have the precise characteristics that result in divergence of the parameters. Additionally, the newer algorithms require an additional learning parameter to tune. Therefore, in our experiments we use the conventional updates, even though they can not be proven to converge for every setting.

7.4.3 Policy-Gradient Algorithms

Temporal-difference methods can be used to find action values, which in turn can be used to find good policies. Conversely, policy-gradient algorithms attempt to optimize the policy directly (Williams, 1992; Sutton et al., 2000; Baxter and Bartlett, 2001; Peters and Schaal, 2008b; Rückstieß et al., 2010). These algorithms use a parametrized policy $\pi : S \times A \times \mathbb{R}^p \to [0, 1]$, where $\pi(s, a, \psi)$ denotes the probability of selecting a in s for a given policy parameter vector $\vec{\psi} \in \mathbb{R}^p$.

The idea of policy-gradient algorithms is to update the policy with gradient ascent on the cumulative expected value V^π. If we assume the gradient is known, we can update the policy parameters by

$$\vec{\psi}^T_{k+1} = \vec{\psi}^T_k + \alpha_t \nabla_\psi \int_{s\in S} P(s_t = s) V^\pi(s)\, ds \quad .$$

Here $P(s_t = s)$ denotes the probability that the agent is in state s at time step t. In this update we use a subscript k instead of t to distinguish between the time step of the actions and the update schedule of the policy parameters, which may not overlap. If the state space is finite, we can replace the integral with a sum.

As an alternative, we can use stochastic gradient descent:

$$\vec{\psi}_{t+1}^T = \vec{\psi}_t^T + \alpha_t \nabla_\psi V^\pi(s_t) \ . \tag{7.14}$$

Here the time step of the update corresponds to the time step of the action and we use the subscript t. Such procedures can at best hope to find a local optimum, because of the gradient updates and the fact that the value function is usually not convex with respect to the policy parameters. However, with such an update some promising results have been obtained, for instance in robotics (Benbrahim and Franklin, 1997; Peters et al., 2003).

The obvious problem with update (7.14) is that in general V^π is not known and therefore neither is its gradient. For a successful policy-gradient algorithm, we need an estimate of $\nabla_\psi V^\pi$. We will now discuss how to obtain such an estimate.

We will use the concept of a trajectory. A trajectory $\mathscr{S}$ is a sequence of states and actions:

$$\mathscr{S} = \{s_0, a_0, s_1, a_1, \ldots\} \ .$$

The probability that a given trajectory occurs is equal to the probability that the corresponding sequence of states and actions occurs with the given policy:

$$\begin{aligned} P(\mathscr{S}|s,\vec{\psi}) &= P(s_0 = s)P(a_0|s_0)P(s_1|s_0,a_0)P(a_1|s_1)P(s_2|s_1,a_1)\cdots \\ &= P(s_0 = s)\prod_{t=0}^{\infty} \pi(s_t,a_t,\vec{\psi})P_{s_t a_t}^{s_{t+1}} \ . \end{aligned} \tag{7.15}$$

The expected value V^π can then be expressed as an integral over all possible sequences for the given policy and the corresponding expected rewards:

$$V^\pi(s) = \int_{\mathscr{S}} P(\mathscr{S}|s,\psi) E\left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| \mathscr{S} \right\} d\mathscr{S} \ .$$

Then, the gradient thereof can also be expressed in closed form:

$$\begin{aligned} \nabla_\psi V^\pi(s) &= \int_{\mathscr{S}} \nabla_\psi P(\mathscr{S}|s,\vec{\psi}) E\left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| \mathscr{S} \right\} d\mathscr{S} \\ &= \int_{\mathscr{S}} P(\mathscr{S}|s,\vec{\psi}) \nabla_\psi \log P(\mathscr{S}|s,\vec{\psi}) E\left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| \mathscr{S} \right\} d\mathscr{S} \\ &= E\left\{ \nabla_\psi \log P(\mathscr{S}|s,\vec{\psi}) E\left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| \mathscr{S} \right\} \middle| s,\vec{\psi} \right\} \ , \end{aligned} \tag{7.16}$$

where we have used the general identity $f(x)\nabla_x \log f(x) = \nabla_x f(x)$. his useful observation is related to Fisher's score function (Fisher, 1925; Rao and Poti, 1946) and the likelihood ratio (Fisher, 1922; Neyman and Pearson, 1928). It was applied to reinforcement learning by Williams (1992) for which reason it

is also known as the REINFORCE trick, after the policy-gradient algorithm that was proposed therein (see, for instance, Peters and Schaal, 2008b).

The product in the definition of the probability of the trajectory as given in (7.15) implies that the logarithm thereof consists of a sum of terms, in which only the policy terms depend on $\vec{\psi}$. Therefore, the other terms disappear when we take the gradient and we obtain:

$$\begin{aligned}\nabla_\psi \log P(\mathscr{S}|s,\vec{\psi}) &= \nabla_\psi \left(\log P(s_0 = s) + \sum_{t=0}^{\infty} \log \pi(s_t, a_t, \vec{\psi}) + \sum_{t=0}^{\infty} \log P_{s_t a_t}^{s_{t+1}} \right) \\ &= \sum_{t=0}^{\infty} \nabla_\psi \log \pi(s_t, a_t, \vec{\psi}) \ . \end{aligned} \tag{7.17}$$

This is nice, since it implies we do not need the transition model. However, this only holds if the policy is stochastic. If the policy is deterministic we need the gradient $\nabla_\psi \log P_{sa}^{s'} = \nabla_a \log P_{sa}^{s'} \nabla_\psi \pi(s,a,\vec{\psi})$, which is available only when the transition probabilities are known. In most cases this is not a large problem, since stochastic policies are needed anyway to ensure sufficient exploration.

For instance, suppose the policy is a Boltzmann distribution with parameters $\vec{\psi}$, where $\vec{\phi}(s,a)$ is a feature vector of size p corresponding to state s and action a. Then

$$\pi(s,a,\vec{\psi}) = \frac{\mathrm{e}^{\vec{\psi}^T \vec{\phi}(s,a)}}{\sum_{b \in A(s)} \mathrm{e}^{\vec{\psi}^T \vec{\phi}(s,b)}} \ ,$$

and the gradient of the logarithm of this policy is given by

$$\nabla_\psi \log \pi(s,a,\vec{\psi}) = \vec{\phi}(s,a) - \sum_b \pi(s,b,\vec{\psi}) \vec{\phi}(s,b) \ .$$

As another example, consider a Gaussian policy with a state-dependent mean $\vec{\mu}_s \in \mathbb{R}^{D_A}$ and the $D_A \times D_A$ covariance matrix Σ_s as adjustable parameters in the action space, such that

$$\begin{aligned} \pi(s,a,\{\vec{\mu}_s,\Sigma_s\}) &= \frac{1}{\sqrt{2\pi \det \Sigma_s}} \exp\left(-\frac{1}{2} (a - \vec{\mu}_s)^T \Sigma_s^{-1} (a - \vec{\mu}_s) \right) \ , \\ \nabla_\mu \log \pi(s,a,\{\vec{\mu}_s,\Sigma_s\}) &= (a - \vec{\mu}_s)^T \Sigma_s^{-1} \ , \\ \nabla_\Sigma \log \pi(s,a,\{\vec{\mu}_s,\Sigma_s\}) &= \frac{1}{2} \left(\Sigma_s^{-1} (a - \vec{\mu}_s)(a - \vec{\mu}_s)^T \Sigma_s^{-1} - \Sigma_s^{-1} \right) \ . \end{aligned}$$

where the actions $a \in A(s)$ are vectors of the same dimension as $\vec{\mu}_s$. It makes sense use a state-dependent parametrized function to store the mean, such that $\mu_s : S \times \Psi_\mu \to \mathbb{R}^{D_A}$, where $\vec{\psi}_\mu \in \Psi_\mu$ is the adaptable policy parameter vector that determine the location of the mean for all states. Then the chain rule can be used to adapt the parameters of these functions. For instance, below we use a neural network to output the mean of the exploration policy;

the parameter vector $\vec{\psi}_\mu$ then contains the weights of the network. The covariance matrix can be the output of a adjustable function as well, although care should be taken that the amount of exploration. The gradient update may decrease the exploration before some interesting parts of the state space are reached, which is undesirable. In the experiments in this chapter we will sometimes use a covariance matrix of $\sigma^2 I$, where σ is a fixed exploration parameter rather than an adjustable parameter.

When we can obtain the gradient of the policy, we can sample the quantity in (7.16). For this, we need to sample the expected cumulative discounted reward. For instance, if the task is episodic we use $\gamma = 1$ and take a Monte Carlo sample that gives the cumulative reward for each episode. Additionally, if the task is episodic the sum in (7.17) is also finite rather than infinite. Then we obtain

$$\nabla_\psi V^\pi(s) \approx R_k \sum_{t=t_k}^{T_k} \nabla_\psi \log \pi(s_t, a_t, \vec{\psi}) \ , \tag{7.18}$$

where R_k is the total return for episode k that started on time step t_k and ended on T_k. This is an unbiased estimate for (7.16) and can then be used to update the policy through (7.14).

A drawback of estimate (7.18) is that the variance of R_k can be quite high, resulting in noisy estimates of the gradient. Williams (1992) notes that this can be mitigated somewhat by using the following update:

$$\vec{\psi}_{t+1}^T = \vec{\psi}_t^T + \alpha_t (R_k - b) \sum_{t=t_k}^{T_k} \nabla_\psi \log \pi(s_t, a_t, \vec{\psi}) \ . \tag{7.19}$$

Here b is a baseline that is independent on the policy. This baseline can be used to minimize the variance without adding bias to the update, since

$$\begin{aligned} \int_{\mathscr{S}} \nabla_\psi P(\mathscr{S}|s, \vec{\psi}) b(s) \, d\mathscr{S} &= b(s) \int_{\mathscr{S}} \nabla_\psi P(\mathscr{S}|s, \vec{\psi}) \, d\mathscr{S} \\ &= b(s) \nabla_\psi \int_{\mathscr{S}} P(\mathscr{S}|s, \vec{\psi}) \, d\mathscr{S} \\ &= b(s) \nabla_\psi 1 = 0 \ . \end{aligned}$$

Some work has been done to optimally set this baseline to minimize the variance and thereby increase the expected convergence rate of the algorithm, but we will not go into this in detail here (see, e.g., Greensmith et al., 2004; Peters and Schaal, 2008b).

The policy-gradient updates as defined above all use a gradient that updates the policy parameters in the direction of steepest ascent of the performance metric. However, the gradient update operates in parameter space, rather than in policy space. In other words, when we use normal gradient descent with a step size, we restrict the size of the change in parameter space: $(\vec{\psi}_{t+1} - \vec{\psi}_t)^T (\vec{\psi}_{t+1} - \vec{\psi}_t)$. It has been argued that it is much better to restrict

the step size in policy space. This is similar to our observation in Section 7.3.4 that an update in parameter space for a linear function approximator can result in an update in value space with a unintended large or small learning rate. For policy-gradient algorithms, we can improve the update by weighing the difference in parameter space with the Fisher information matrix. This matrix is defined as

$$F_\psi = E\left\{ \nabla_\psi P(\mathscr{S}|s,\vec{\psi}) \nabla_\psi^T P(\mathscr{S}|s,\vec{\psi}) \middle| \right\},$$

where the expectation ranges over the possible trajectories. This matrix can be sampled with use of the identity (7.17). Then, we can obtain a natural policy gradient, which follows the so called natural gradient (Amari, 1998). This idea was first introduced in reinforcement learning by Kakade (2001). The update then becomes

$$\vec{\psi}_{t+1}^T = \vec{\psi}_t^T + \alpha_t F_\psi^{-1} \nabla_\psi V^\pi(s_t) \ .$$

A disadvantage of this update is the need for enough samples to (approximately) compute the inverse matrix. The number of required samples can be restrictive if the number of parameters is fairly large, especially if a sample consists of an episode that can take many time steps to complete. For more details, see Kakade (2001); Peters and Schaal (2008a); Wierstra et al. (2008) and Rückstieß et al. (2010).

Many reinforcement learning algorithms exist that make use of such natural gradient updates. Examples include various forms of natural actor-critic algorithms (NAC, Peters and Schaal, 2008a; Bhatnagar et al., 2009) and natural evolutionary strategies (NES), where the latter combines ideas from evolutionary algorithms and policy-gradient methods (Wierstra et al., 2008; Sun et al., 2009). We will explain how this last algorithm approximately works, since it is an interesting combination of policy-gradient methods with evolutionary methods. The idea behind the algorithm is fairly simple, although many specific improvements are quite advances (Sun et al., 2009).

7.4.4 Natural Evolutionary Strategies

Now, we discuss natural evolutionary strategies (NES), an interesting particular algorithm that combines ideas from evolutionary algorithms and policy-gradient methods (Wierstra et al., 2008; Sun et al., 2009). The idea behind the algorithm is fairly simple, although many possible improvements are quite advanced (Sun et al., 2009). We will not give the specifics of these improvements and only discuss the high level view.

Instead of storing a single exploratory policy, NES creates a population of n parameter vectors $\vec{\psi}_1, \ldots, \vec{\psi}_n$. These vectors represent policies that have a certain expected payoff; this is their fitness. The goal is to improve the

parameters of the distribution that generates the population, such that if we update the parameters and create a new population these solutions will likely be better. In other words, we do not improve the policies themselves; we improve the process that generates the policies. For this, we use a gradient ascent step on the fitness of the current solutions.

The parameter vectors $\vec{\psi}_i$ in a population are all drawn from a Gaussian distribution $\vec{\psi}_i \sim \mathcal{N}(\mu_\psi, \Sigma_\psi)$. The distribution has a total of $p + p(p+1)/2$ parameters, where p parameters correspond to the mean and $p(p+1)/2$ parameters correspond to the covariance matrix.[3] Using Monte Carlo samples one can find an estimate of the gradient of the performance of the population to the meta-parameters in μ_ψ and Σ_ψ. This tells us how the meta-parameters should be changed in order to generate better populations in the future.

In NES, no crossover exists. A nice feature of NES compared to other evolutionary algorithms is that all solutions in the population are used to determine the gradient of the population parameters. In other methods, the mutation is not random, but moves into a direction of gradient ascent. Because of the choice of a Gaussian generating distribution, it is possible to calculate the Fisher information matrix analytically. With further algorithmic specifics, it is possible to restrict the computation for a single generation to $O(np^3 + nf)$, where n is the number of solutions in the population, p is the number of parameters of a solution and f is the computational cost of determining the fitness for a single solution. The potentially large variance in the fitness may make NES less appropriate for large, noisy problems. Note that the solutions can be deterministic policies, which may reduce the variance compared to stochastic policies. Whether the cubed parameter length in the complexity is a problem is task dependent. There are many tasks that can be solved with a controller with a fairly small amount of parameters, but constructing a suitable functional form may be non-trivial and therefore it may be important to have some domain knowledge to avoid overly limiting the solution space.

7.4.5 Adaptive Dynamic Programming

There is significant overlap between some of the policy-gradient ideas and many of the ideas in the related field of adaptive dynamic programming (ADP). For recent overviews see Powell (2007) and Wang et al. (2009). Essentially, reinforcement learning and ADP can be thought of as different names for the same research field. However, in practice there is a significant divergence between the sort of problems that are considered and the solutions that are proposed.

[3] Note that the covariance matrix is symmetrical, which is why only $p(p+1)/2$ parameters are needed, rather than p^2 parameters.

Usually, research papers on adaptive dynamic programming seem to look at the world more from an engineering's perspective. This results in a slightly different notation and a somewhat different set of goals. For instance, while the goal in reinforcement learning is often to reach a good policy for some problem, in ADP the goal is usually to stabilize a plant (Murray et al., 2002). This puts some restraints on the exploration that is to be used and also implies that often, the goal state is the starting state and the goal is to stay near this state, rather that to go out and find better states. Additionally, problems in continuous time are discussed more often in ADP than in reinforcement learning (e.g., Beard et al., 1998; Vrabie et al., 2009). For these problems a continuous version of the Bellman optimality equation is used, that it known as the Hamilton–Jacobi–Bellman equation (Bardi and Dolcetta, 1997). A further discussion of these specifics falls outside the scope of this dissertation.

A particular model-free algorithm we will discuss is called action dependent heuristic dynamic programming (ADHDP) (Werbos, 1977; Prokhorov and Wunsch, 2002). It can be interpreted as an actor critic type algorithm, where an actor stores $Ac(s,\vec{\psi})$, which is an estimate for the locally optimal action for the current value function and a critic stores $Q(s,a,\vec{\theta})$ which is an estimate of the value of the action. The actor can be updated with a gradient ascent update on the value of the critic by

$$\begin{aligned}\vec{\psi}_{t+1}^T &= \vec{\psi}_t^T + \alpha_t \nabla_\psi Q(s, Ac(s,\vec{\psi}),\vec{\theta}) \\ &= \vec{\psi}_t^T + \alpha_t \frac{\partial Q(s, Ac(s,\vec{\psi}),\vec{\theta})}{\partial Ac(s,\vec{\psi})} \nabla_\psi Ac(s,\vec{\psi}) \ .\end{aligned}$$

The critic can be updated with any action value algorithm. There are many variants of this algorithm. Many of these variants assume a known model of the environment, the reward function or both, or they construct such models.

There are many variants of this algorithm that we will not further discuss. Many of these variants use a known model of the environment, the reward function or both, or they construct such models. We restrict ourselves to the model-free case, but we do note that for some problems a model-based or model-building approach may yield better results. As argued before, this is dependent on the relative complexity of the model compared to the solution that is sought.

7.5 Continuous Actions

In this section we will look at reinforcement learning algorithms for continuous action spaces. We will discuss which aforementioned methods are applicable and we will look at a simple new temporal-difference algorithm called continuous actor critic learning automata (Cacla). This algorithm is very easy

to implement and computes updates and actions very fast. Additionally, it is shown to compare well to the current state of the art.

7.5.1 Previous Work

All the aforementioned algorithms that search directly in policy space can directly be applied to problems with continuous actions. This includes policy-gradient algorithms with Gaussian policies and evolutionary algorithms such as NES.

It is harder to extend online temporal-difference algorithms to continuous actions. Although it is possible to construct an action value function that gives an estimate of the value for each continuous action, it is then non-trivial to find the best action. One way to do so is by gradient ascent on the value, as in ADHDP. Another way is to simply discretize the continuous space, as we have done in Chapter 5 for the mountain car task. Our discretization there can be interpreted as a form of tile coding which was described earlier, but with only a single tiling. Using more tilings can increase the amount of information that is conveyed with each state and the resolution of the action that is output. However, the spaces are still discretized and there is no generalization between parts of the state and action space that may contain similar features, but do not fall into the same tiles. Depending on the problem, this can sometimes be an advantage. For instance, in the mountain car the optimal policy is highly discontinuous since the controller should at one point switch between pushing up a hill to pushing down a hill in very similar states. Such a policy is perhaps more easily stored in a truly discontinuous manner than with a smooth function approximator.

Another way to use temporal-difference algorithms is to perform a line search. For instance, binary action search uses an ordinary action value algorithm to perform such a line search (Pazis and Lagoudakis, 2009). In its original form, this algorithm assumes a one-dimensional action space. It then chooses the center of the space and constructs an augmented state, consisting of the current state and this first action. Then, an ordinary action value based algorithm is given this augmented state as input and is used to answer the question whether the action to be performed should be larger or smaller than the current estimate. After the decision, the center of the rest of the action space is chosen and the corresponding augmented state is fed again to the decision algorithm. This continues a fixed number of steps, until a sufficiently fine grained resolution is obtained. Strangely, in the original formulation of the algorithm, each decision is marked a discrete time step, such that each continuous action takes N discrete time steps. Then, the reward is only received at the last decision. It is unclear why not all decisions are interpreted as being made at the same time step and the reward and transition are used as feedback for all decisions simultaneously. Indeed, in our experiments this

proved to work better. However, the performance of the binary action search algorithm was not nearly as good as that of other methods, which is why we do not consider it further. Of course, binary action search does not use actual continuous actions, since the possible set of actions is fully determined by the range of the action space and the choice of the number of decision steps to take.

One final method that we like to mention is similar to both temporal-difference methods, policy-gradient methods and ADHDP. This algorithm is called wire-fitting (Baird and Klopf, 1993; Gaskett et al., 1999). This algorithm uses a function approximator to output a fixed number of candidate vectors that contain the (possibly multi-dimensional) action and the value of this action. The actions and values are then fed into a least squares interpolator to form the whole current estimate of the continuous action value function in the current state. Because of the interpolation, the maximal value of the resulting function will always lie precisely on one of the actions, thus facilitating the selection of the greedy action in the continuous space. The action value function can then be updated by using a one step temporal-difference error, for instance as in Q-learning. The function is updated with a gradient step to minimize this temporal-difference error. Because of the interpolation, the effect is that the actions that are close to the action that was actually selected move closer to the resulting action and target value. Unfortunately, this implies that actions near the current best action that give poor values not only pull the value of the best action down, but also move this action away from its previous position. This can perhaps be mitigated by differentiating the learning rate for actions that resulted in a large target value and actions that resulted in a low target value, but to the best of our knowledge such an extension to the algorithm was never published. Wire-fitting was shown to perform worse than other methods (van Hasselt and Wiering, 2007a), so we do not further consider this methods in this dissertation.

7.5.2 Continuous Actor Critic Learning Automaton

In this section we discuss the continuous actor critic learning automaton (Cacla) algorithm that can learn undiscretized continuous actions in continuous states with use of temporal-difference methods (van Hasselt and Wiering, 2007a; van Hasselt and Wiering, 2009). This algorithm was compared favorably to ADHDP and wire-fitting (van Hasselt and Wiering, 2007a). We will first describe how the algorithm works and later we will test it empirically. Additionally, we will present some previously unpublished extensions and improvements.

Cacla uses a critic that stores state values. For instance, the methods in Section 7.4.2 can be used to update this critic. Additionally, similar to ADHDP an actor is used that outputs a single—possibly multi-dimensional—

Algorithm 7 Cacla

1: Given γ, an initial state distribution I and an MDP to act on.
2: Initialize $\vec{\theta}$, $\vec{\psi}$, $s \sim I$.
3: **repeat**
4: Choose $a \sim \pi(s, \vec{\psi})$
5: Perform a, observe r and s'
6: $\delta = r + \gamma V(s') - V(s)$
7: $\vec{\theta}^T = \vec{\theta}^T + \beta\delta\nabla_\theta V(s)$
8: **if** $\delta > 0$ **then**
9: $\vec{\psi}^T = \vec{\psi}^T + \alpha(a - Ac(s, \vec{\psi}))\nabla_\psi Ac(s, \vec{\psi})$
10: **end if**
11: **if** s' is terminal **then**
12: $s \sim I$
13: **else**
14: $s = s'$
15: **end if**
16: **until** end

action from a continuous domain. On each time step, some exploration is used to determine an action that is to be performed. Then, the temporal-difference error is used to determine if that action was a good idea or not. As in most actor critic algorithms, if the temporal-difference error was positive, we judge the action to have been profitable and we reinforce it. In this case, this means we update the actor function approximator towards the action that was selected. This implies that similar to policy-gradient algorithms stochastic policies should be used: if there is no exploration the action will be equal to the output of the actor and an update would leave the actor unchanged.

A basic version of Cacla is shown in Algorithm 7. The policy in line 4 is derived from the actor's output. For instance, one can use Gaussian exploration around this action. As noted, it is important that $a_t \neq Ac(s_t, \vec{\psi}_t)$ in order for Cacla to update its actor. Of course, after training has concluded and the actor is no longer updated, the agent can deterministically use the action that is output by the actor.

The critic update in line 7 is an ordinary TD learning update. One can replace this with a TD(λ) update or with one of the updates from Section 7.4.2. The actor update in line 9 essentially performs gradient descent on the error between the action that was performed and the output of the actor. Note that an update only occurs when the temporal-difference error was positive. One might imagine that one could also include an update that moves away from an action when the temporal-difference error was negative. However, this is usually not a good idea since this is equivalent to updating towards some action that was not performed and of which it is not known whether it

is better than the current output of the actor. As an extreme case, consider an actor that outputs the optimal action in each state of a deterministic MDP. If an update would occur when the temporal-difference error is negative, the actor would change its output even though it was already optimal. This is an important difference between Cacla and policy-gradient methods: Cacla only updates its actor when actual improvements have been observed.

The Cacla algorithm essentially performs hill climbing in action space. This is slightly different from hill climbing in policy space, as policy-gradient methods do. This also means that Cacla is compatible with some types of exploration that policy-gradient algorithms are not. For instance, a uniform random exploration policy would still allow Cacla to improve its actor, whereas such a policy has no parameters to tune for policy-gradient methods. Other differences are that policy-gradient methods do not use a threshold, but determine the size of the update by a learning rate, or by natural gradients. In Cacla the size of the update towards the action is governed only by the learning rate parameter α_t and not by the size of the temporal-difference error. This allows Cacla to learn quickly also when there are plateaus in the value space. It was shown that this works better in at least some settings than when the learning rate is made dependent on the size of the temporal-difference error (van Hasselt and Wiering, 2007a). Intuitively, it makes sense that it is the distance to the better action that is important for the actor and not necessarily the size of the improvement in value.

The hill climbing of Cacla is stochastic because of two reasons: the value function is updated by the critic during the improvements to the policy and the temporal-difference error may contain noise. This allows Cacla to get out of some local optima. However, like any approximate hill climbing algorithm Cacla can get stuck in a local optimum.

Although it is designed for continuous action spaces, we will also compare Cacla to discrete algorithms. Then, we discretize the output of the actor of Cacla by rounding it to the nearest allowed action in the action space of the MDP. This makes sure that Cacla does not have the advantage of being able to more finely determine its action, although since it is a feature of the design of the algorithm, this advantage is arguably neither unrealistic nor unfair. By its nature, Cacla will assume the underlying action space is continuous and this can be an advantage even if the action space that is actually used is discrete. Specifically, this is an advantage if the finite set of actions follow an ordinal scale, rather than a nominal scale.

In the remainder of this chapter, we will compare Cacla to some of the other algorithms and show that it is competitive with the current state of the art, although it is much easier to implement than most other methods.

7.6 Experiments

In this section we compare the performance of the discrete temporal-difference algorithms from Chapter 5 on the mountain car and the cart pole when we do not discretize the state space. We will compare this performance with Cacla and additionally compare Cacla to some other state of the art continuous action algorithms.

Unless stated otherwise, as function approximator a neural network multi-layer perceptron (MLP) with a single hidden layers with 15 nodes and a tanh activation function was used. The input and output layers then use the identity function and learning is done with a gradient descent update as in (7.11), using backpropagation. This function approximation may not be suitable for every problem, as some problems may benefit from more or less hidden nodes, or a different activation function. We did some preliminary tests in our experiments and found no improvements in the results when using MLPs with more of less hidden nodes. To a certain point, less hidden nodes seem to result in faster learning to less optimal solutions and more hidden nodes seem to result in slower learning to better solutions. Of course, this depends on the complexity of good policies and the complexity of good value functions in the problem domain. Our focus is on the comparison of the algorithms that use the function approximators, rather than the function approximators themselves. However, we note that somewhat different results will probably be reached with other functional representations.

As in Chapters 5 and 6 we use a Welch's test with $\alpha = 0.01$ to determine whether two approaches are significantly different. Therefore, we will call the difference between the results of two methods significant if this difference is greater than

$$2.326\sqrt{se_1^2 + se_2^2} \ ,$$

where se_1 is the standard error of the first method and se_2 is the standard error of the second method.

7.6.1 Mountain Car

The mountain car setup is the same as in Chapter 5 with the difference that the position of the car x and its velocity dx are not discretized. Rather, they are given as inputs to a multi-layer perceptron, which then approximates the relevant action, state or preference value function.

The algorithms receive a reward of -1 on every time step, except when an episode ends in a success. Then a reward of $+100$ is received. If no success is obtained for 500 time steps, the episode is considered a failure and the episode also ends. Whenever an episode ends, the car is reset to the bottom of the track with zero velocity. The discount factor was $\gamma = 0.95$.

For all discrete algorithms, a Boltzmann exploration was used. For Cacla, Gaussian exploration was used and after adding this exploration, the resulting action was scaled to the nearest legal option. The exploration parameters and learning rates are given below.

Training in this task was limited to 1000 episodes, each of which lasted a maximum of 500 time steps. This actually biases the results somewhat in favor of poorly performing algorithms, since these get a larger number of total training steps if the episodes take longer. After every episode, a test episode was run without exploration and without updating any of the algorithms. This problem is arguably less suited to Cacla than the cart pole problem that we will consider later, because Cacla is designed explicitly for continuous action spaces and here there are only two actions of real interest: driving as fast to the left as possible and driving as fast to the right as possible. This means the ability of Cacla to finely tune its actor output becomes less important.

7.6.2 Results on the Mountain Car

Table 7.1 shows the average results over the whole training run (total online) of the last 10% of training episodes (final online) and for 100 test episodes without exploration after training has concluded (greedy). We see that in the greedy result Acla performs significantly better than Cacla, which in turn significantly outperforms all other algorithms. This implies that Acla has the most success in eventually finding a policy that works well. However, for this policy to be found, substantial exploration was needed. This explains the poor behavior of Acla in the online settings. In online performance, Cacla manages to outperform all the other algorithms with QV-learning a relatively distant second. The results for Expected Sarsa are not shown, but preliminary experiments indicate that these results are similar to those of Sarsa for this problem.

Interestingly, we see that Cacla uses very large values for its Gaussian exploration in this problem. The reason is that in the mountain car only the two extreme actions are relevant. The idea is to drive as fast as you can to the left and then as fast as you can to the right. Cacla learns to output actions far above and below the actual actions of 1 and -1, because they are rounded to these actions anyway. Even in the online results, Cacla can perform well with very high exploration, while most other algorithms use quite low temperatures that translate into little exploration. As mentioned, this explains the large difference between the offline and online performance of Acla. The offline performance uses a high temperature, resulting in much exploration. This apparently allows Acla to learn good offline policies, but such a high temperature will not lead to good online performance. This shows that although technically Acla is an online algorithm, it has some offline characteristics because it can learn good offline policies from noisy online behavior.

Table 7.1: Average number of steps until goal is reached (lower is better). Shown are the mean performance with exploration during training (*total online*), the final performance with exploration after training (*final online*) and the final greedy performance. Averages over 300 trials.

Total online performance					
	α	β	**exploration. mean**	**std error**	
Cacla	0.0002	0.05	20.	306.0	1.0
QV	0.00001	0.2	0.001	351.5	4.6
AC	0.001	0.2	1.	428.3	3.3
Acla	0.0002	0.002	0.01	430.2	5.3
Sarsa	0.0005		0.01	461.2	3.9
Q	0.001		0.01	478.5	2.6
Final online performance					
	α	β	**exploration**	**mean**	**std error**
Cacla	0.00005	0.02	50.	222.4	4.4
QV	0.00002	0.1	0.002	309.9	7.6
AC	0.001	0.2	2.	345.4	7.1
Sarsa	0.0002	-	0.01	453.6	6.0
Acla	0.0005	0.001	0.01	455.0	6.4
Q	0.0001	-	0.01	495.9	1.2
Final greedy performance					
	α	β	**exploration**	**mean**	**std error**
Acla	0.005	0.05	100.	132.2	0.9
Cacla	0.00002	0.05	100.	187.3	3.1
QV	0.00002	0.1	0.002	273.4	9.2
AC	0.0005	0.2	1.	416.4	8.5
Sarsa	0.0002	-	0.01	462.7	5.9
Q	0.00002	-	1.	472.1	5.1

We also tested Cacla with a fully random uniform exploration policy. The online performance is then -500 in all cases, since the random policy does not manage to drive up the hill. However, after only 90 episodes the final greedy performance is already quite good at 193.4, which is almost as good as the best final greedy performance with Gaussian exploration after $1{,}000$ episodes. The greedy policy was tested after every 3 episodes. The average of the first 30 of these tests—again, after 90 episodes in total—is 319.1: not much worse than the best online average over 1,000 episodes with Gaussian exploration. These results show that in this setting Cacla can learn about good policies without following them. Note that during training the goal state

was not reached a single time. This did not prevent Cacla from finding good policies for its actor, due to the -1 reward on each step that gives an incentive to get out of the valley. Naturally, there are problems in which good policies can not be learned without bootstrapping on a policy that is already at least somewhat reasonable, but this demonstrates that Cacla is fairly robust to the type of exploration that is used.

On the whole, the results show that Cacla is a very reasonable choice to solve problems similar to the mountain car. Depending on the performance measure that you consider, Cacla performs best or second best to Acla. This is somewhat surprising, since the discontinuous nature of the optimal policy in the mountain car and the fact that there are only two relevant actions would seem to favor discrete algorithms that only consider these actions, rather than an algorithm that considers the whole range of possibilities.

We can and should compare these results with the results in Table 5.5, which were obtained with a discretization of the state space. The results are not directly comparable, since for Table 5.5 the training time was always $100,000$ steps, regardless of the number of episodes. However, given the average number of steps in the total online performance in Table 7.1 we can conclude that our neural network counterparts in the continuous state space have used approximately 3 to 5 times that number of training steps. It is then interesting to note that only Cacla comes close in terms of its online performance. For the offline performance, Cacla is also comparable to the results in the discretized state space. The neural version of Acla outperforms all the other algorithms in terms of its greedy performance. Even the difference with the discretized Actor Critic is significant, although it is small. However, the neural network version Acla had the advantage of a finer tuning of the learning parameters, as well as a longer learning time. Therefore, we must conclude that the mountain car can be solve adequately with a simple discretization of the state space and does not profit from the use of neural networks for the controllers, at least in our experimental setup.

7.6.3 Discrete Cart Pole

The cart pole is a non-linear control problem that is often used as a benchmark in reinforcement learning. The goal is to balance a pole that is connected with a hinge to a cart by hitting the cart left and right. If the pole falls further than a specified amount of degrees, the episode is considered a failure. The problem is made harder by an extra condition that the cart may not drive further than a fixed amount to the left or to the right of the starting position. If it does, it hits a wall and the episode is considered a failure. We first consider the discrete version of the problem, where the agents can choose between 21 different actions. Later, we will look at the harder double pole version of the cart pole and allow the agents to use the whole continuous

action space.

For the cart pole task, we use commonly used system dynamics, which can be described as follows.

$$\ddot{\omega} = \frac{g \sin\omega(m_c + m_p) - (F + m_p l \dot{\omega}^2 \sin\omega)\cos\omega}{\frac{4}{3} l(m_c + m_p) - m_p l \cos^2\omega} \ ,$$
$$\ddot{x} = \frac{F - m_p l(\ddot{\omega}\cos\omega - \dot{\omega}^2 \sin\omega)}{m_c + m_p} \ .$$

Here $x \in [-2.4, 2.4]$ is the position of the cart, $\dot{x} = (d/dt)x$ is its velocity and $\ddot{x} = (d/dt)^2 x$ is its acceleration. We use ω to denote the angle of the cart, rather than the more common θ, since we already use this symbol to denote parameter vectors. Similar to the velocity and acceleration of the cart, $\dot{\omega} = (d/dt)\omega$ and $\ddot{\omega} = (d/dt)^2\omega$ denote the angular velocity and acceleration of the pole, respectively. In these formulas, $l = 0.5\,\text{m}$ is half of the length of the pole, $m_c = 1\,\text{kg}$ is the mass of the cart, $m_p = 0.1\,\text{kg}$ is the mass of the pole, $g = 9.81\,\text{m/s}^2$ is the gravitational constant and $F \in [-10\,\text{N}, 10\,\text{N}]$ is the force that is applied to the cart. The time between each two decisions the agent can make is 0.02 s. These dynamics are the same as for instance in Riedmiller et al. (2007) and we fix two small typographic errors in their formulation. The discount factor was $\gamma = 0.99$.

An episode is ended when either it hits one of the walls at 2.4 m in each direction, or when the pole drops further than 12° from its upright position. In both cases the MDP reaches a terminal state and after this it is reset. An episode ends and is considered a success when the pole is balanced for at least 40 s. Note that this does not imply that the MDP reaches a terminal state after 40 s. This would make the state non-Markovian if the elapsed time is not part of the state. Rather, the environment is reset after an ended trial and the agent is not updated on the transition from the last state of the former episode to the first state of the next episode. Therefore, the goal of the agent is to balance indefinitely and not only for a maximum of 40 s.

When an episode ends, the cart is reset at the center of the track with the pole tilted randomly between −3° and 3°, where 0° is upright. The dynamics do not include friction, but are realistic in the other aspects. The state vector given to the algorithms consists of the position and velocity of the cart and the angle and angular velocity of the pole. On every time step the algorithms receive a reward of +1, except when an episode ends with a failure. Then a reward of −1 is received. The reward on the transition from the end of a successful episode to the next episode is inconsequential, since this reward is not used in an update anyway.

An important difference between the cart pole and the mountain car is that a random or poorly performing algorithm will get a lot of meaningful

feedback on the cart pole, since the pole will drop quickly and will allow the algorithm to get information about good and bad situations to be in. In the mountain car such an algorithm will only observe rewards of -1 for all time steps, which by themselves do not carry much information. We will see that the two tasks are sufficiently different to require quite different settings of the parameters of the algorithms to reach the best performances.

7.6.4 Discrete Cart Pole Results

Table 7.2 shows the results on the cart pole task for 2000 simulated seconds of training. The total online performance gives the average failures per second for the whole training time. For the total greedy performance, after each 20 s of training a test run of 40 s was run without exploration. This tells us something about how quickly the algorithms find good greedy policies. For the final online performance the last 40 s of training were used. For the greedy performance after training had concluded a run of 40 s was used without exploration.

Acla is by far the best algorithm in terms of total online performance. This implies that Acla reaches good policies fast. Interestingly, Acla's online performance is even better than its offline performance. Apparently, Acla can use some stochasticity to reach good results. We did not investigate further why this is the case. Cacla finds good greedy policies the fastest, as indicated by its good total greedy performance. Also, the final policies found by Cacla with and without exploration are better than those found by the other algorithms. The success rate is also shown, which shows that Cacla is almost flawless and that almost 99% of its policies can balance the pole for 40 s after 2000 s of learning.

Table 7.3 shows how quickly on average each algorithm conducted its first 'perfect' run. A perfect run is defined here as balancing the pole for 40 s. We see that Acla finds a perfect policy the fastest if we include exploration, but Cacla finds a perfect deterministic policy the fastest. The greedy results are probably slightly overestimated, since the greedy policy was only checked every 20 s and not continuously. Therefore, probably a perfect policy is found slightly faster than the results shown in the figure. This does not hold for the online results, since these were collected during the actual training.

We see that although Actor Critic reaches similar results as Acla and Cacla in total performance, the latter two algorithms reach a flawless performance a lot faster. In offline performance Cacla outperforms all the other algorithms, while in online performance the same holds for Acla. In both cases, the difference is statistically significant. Interestingly, Actor Critic only manages to outperform Q-learning and Sarsa. For all algorithms except Acla the online performance is worse than the offline performance by a considerable margin. It is unclear what makes Acla special in this regard.

Table 7.2: Mean performance with exploration during training (*total online*), final performance with exploration after training (*final online*) and the performance of the best policy without exploration after training (*final greedy*). Training lasted 2000 s, each test run was 40 s. Results are averages over 300 repetitions of the experiment.

Total online performance					
	α	β	**exploration**	**mean**	**std err**
Acla	0.05	0.005	0.1	0.103	0.002
Cacla	0.0005	0.002	20.0	0.141	0.003
AC	0.01	0.002	5.0	0.148	0.002
QV	0.005	0.005	1.0	0.206	0.004
Sarsa	0.01	-	0.5	0.316	0.005
Q	0.01	-	0.5	0.320	0.006
Total greedy performance					
	α	β	**exploration**	**mean**	**std err**
Cacla	0.0005	0.002	10.0	0.107	0.003
AC	0.002	0.002	10.0	0.113	0.002
Acla	0.05	0.005	0.1	0.131	0.003
QV	0.005	0.005	1.0	0.229	0.005
Q	0.01	-	0.5	0.230	0.004
Sarsa	0.01	-	0.5	0.332	0.005
Final online performance					
	α	β	**exploration**	**mean**	**success**
Cacla	0.002	0.002	0.2	0.005	97.7 %
Acla	0.1	0.002	0.05	0.011	95.2 %
AC	0.002	0.002	5.0	0.012	87.7 %
QV	0.005	0.005	1.0	0.117	48.2 %
Q	0.005	-	0.2	0.128	37.6 %
Sarsa	0.005	-	0.01	0.150	35.5 %
Final greedy performance					
	α	β	**exploration**	**mean**	**success**
Cacla	0.0005	0.001	10.0	0.003	98.8 %
Acla	0.002	0.001	10.0	0.019	91.7 %
AC	0.002	0.002	0.5	0.037	82.6 %
QV	0.02	0.001	0.2	0.022	80.5 %
Sarsa	0.002	-	1.0	0.156	55.7 %
Q	0.05	-	0.01	0.131	54.8 %

7.6.5 Cart Pole with Removed Actions

For the following results we removed some of the available actions after the algorithms were trained on the cart pole task. Three scenarios were tested,

Table 7.3: Time in seconds before the start of the first perfect test and training run. Averaged over 300 trails.

Online performance					
	α	β	**exploration**	**mean**	**std err**
Acla	0.05	0.005	0.1	191.0	5.7
Cacla	0.005	0.002	5.0	281.6	5.2
QV	0.005	0.005	1.0	340.8	9.6
AC	0.01	0.005	2.0	439.8	13.5
Q	0.01	-	0.02	638.1	27.2
Sarsa	0.005	-	0.5	665.6	22.5
Greedy performance					
	α	β	**exploration**	**mean**	**std err**
Cacla	0.01	0.005	5.0	181.1	10.0
QV	0.005	0.005	1.0	232.0	7.2
Acla	0.05	0.005	0.1	236.4	8.3
AC	0.01	0.005	2.0	359.6	13.6
Q	0.005	-	1.0	385.5	12.0
Sarsa	0.01	-	0.5	453.6	22.4

where after 2000 s of training a subset of the actions was made unavailable. In the first scenario *Even*, all the odd positive and negative integer forces were removed, leaving the even integers -10, -8, ..., 8 and 10. In the second scenario *Bang*, we removed all actions except -10, 0 and 10, essentially testing the performance of the algorithms when only the actions of a bang-bang controller were allowed. For the third scenario *No Extreme* we only removed the most extreme options -10 and 10, since we observed that these were often used by all algorithms. In the third scenario therefore 19 of the original 21 actions are still available. For all discrete algorithms after the action have been removed the state action values corresponding to the removed actions are simply not considered anymore. For Cacla the output of the actor gets rounded to the closest available action from the smaller new action set.

After removing the actions, we measured the performance with a greedy test run of 40 seconds. The results are given in Table 7.4. The parameter settings are the same that were used for the best final greedy performance after 2000 seconds of training, as shown in Table 7.2.

Cacla can adapt very successfully to the changed situations. The percentage of successful test runs remains high and the average number of failures per second remains very small.

It is very interesting to consider the differences between the three scenarios. We see that on average performance goes down significantly for all

Table 7.4: Mean amount of failures per second of the first test run after removing odd actions (*Even*), all except −10, 0 and 10 (*Bang*), or just actions −10 and 10 (*No Extreme*). Means and success percentages are averaged over 300 trials and sorted by performance before removing the actions.

	Even		**Bang**		**No Extreme**	
	mean	**success**	**mean**	**success**	**mean**	**success**
Cacla	0.002	98.8 %	0.004	96.0 %	0.003	98.5 %
Acla	0.117	77.3 %	0.209	44.1 %	1.028	3.8 %
AC	0.068	61.5 %	0.154	45.9 %	1.920	0.0 %
QV	2.967	11.8 %	2.894	0.6 %	4.136	0.0 %
R	2.071	22.2 %	3.702	2.5 %	4.187	0.1 %
Sarsa	3.512	10.7 %	2.411	0.2 %	4.107	0.0 %
Q	3.200	11.5 %	3.334	1.5 %	4.284	0.1 %

algorithms except Cacla when the odd actions are removed. However, Acla and Actor Critic still manage to reach a perfect run without additional training in more than half of the 300 trials. In the scenario where we only removed the actions corresponding to −10 N and 10 N performance of all discrete algorithms drops considerably. This is due to the fact that almost all policies found by the algorithms in the first 2000 seconds use these actions regularly. Using its ability to generalize, Cacla will immediately push with 9 N where it used to push with 10 N, but the other algorithms have to relearn which action then to take, since they regard all actions as qualitatively different options and apparently have not learned that 9 N is the second best option when 10 N becomes unavailable.

To get a better intuition of what happens to the performances, Figure 7.2 shows the greedy performance of the best three algorithms: Cacla, Acla and Actor Critic. As explained above, after 2000 seconds actions are removed from the action space. In the left panel, we see that Acla and Actor Critic experience a temporary setback when the odd actions are removed, but quickly regain former performance levels. However, in the right panel we see that if the actions corresponding to −10 N and 10 N are removed they recover much more slowly and to a less optimal level. In the bang scenario the algorithms also recover more slowly, but they regain performance levels comparable and eventually even better than before the actions were removed. Note that all algorithms that are not shown in the figure perform much worse, as can be deducted from Table 7.4.

The results in this subsection show that Cacla can easily adapt to changing action spaces when some of the actions are removed. We expect this to also be the case if the action space is changed in other ways. It is non trivial how to adapt a conventional reinforcement learning algorithm such as

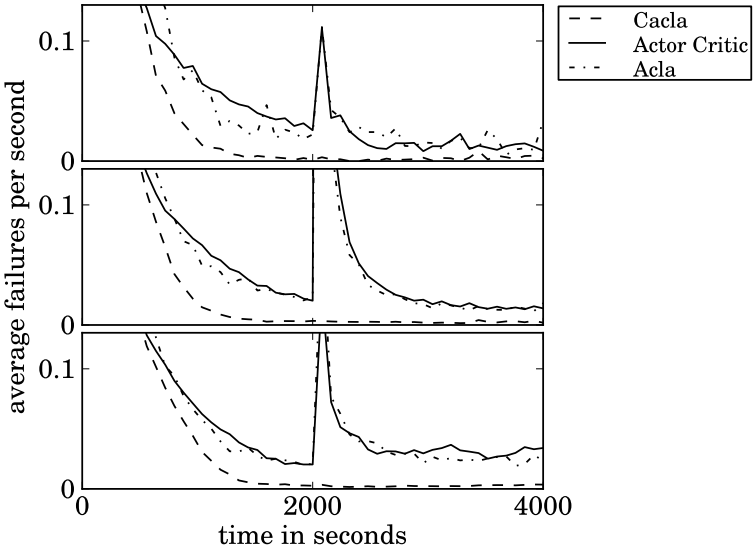

Figure 7.2: The offline performance measured in average failures per second for Cacla, Acla and Actor Critic. After 2000 s the odd actions (top panel, scenario *Even*), all actions except −10 N, 0 N and 10 N (middle panel, scenario *Bang*), or −10 N and 10 N (bottom panel, scenario *No Extreme* are removed. The difference between Cacla and the other two algorithms is significant.

Q-learning when for instance a different set of available actions is chosen from the continuous underlying action space, but Cacla can still simply use its generalization property and adapt with little problems as long as the action space that is used for training is somewhat representative of the action space that is used for testing. This is a useful property, also because often one will want to simulate a real-world problem and train an algorithm on this simulation. If then the actual problem turns out to use a different action space than the simulation, a trained Cacla algorithm can often still be used successfully. For completeness, we note that the online results that are not shown in this subsection are very similar to the offline results.

7.6.6 Double Cart Pole

As our next problem, we discuss the double cart pole. In this case, two separate poles are attached to the pole. The two poles differ in length and mass and both must be balanced by hitting the cart. We will compare the results of Cacla to those in Heidrich-Meisner and Igel (2008), so we copy their dynamics, which are taken from Wieland (1991). We will not explain the algorithms that are used for those results in full, but Covariance Matrix Adaptation Evolution

Strategy (CMA-ES) is an evolutionary algorithm that is considered the current state of the art in optimization (Hansen et al., 2003; Jiang et al., 2008). It was favorably compared to episodic Natural Actor Critic (NAC) (Peters and Schaal, 2008a; Bhatnagar et al., 2009) in the same paper that we take our dynamics from. The NES algorithm we described earlier can be seen as a simpler variant on the CMA-ES algorithm. It usually performs slightly worse than CMA-ES, even with multiple improvements that we did not discuss (Sun et al., 2009; Glasmachers et al., 2010). So therefore, if Cacla compares favorably to CMA-ES, it also compares favorably on similar problems to NES, NAC and other state of the art reinforcement learning algorithms.

The dynamics are as follows.

$$\ddot{x} = \frac{F - \mu_c \text{sign}(\dot{x}) + \sum_{i=1}^{2} 2 m_i \dot{\omega}_i^2 \sin\omega_i + \frac{3}{4} m_i \cos\omega_i \left(2\frac{\mu_i \dot{\omega}_i}{m_i l_i} + g \sin\omega_i\right)}{m_c + \sum_{i=1}^{2} m_i \left(1 - \frac{3}{4}\cos^2\omega_i\right)}$$

$$\ddot{\omega} = -\frac{3}{8 l_i}\left(\ddot{x}\cos\omega_i + g\sin\omega_i + 2\frac{\mu_i \dot{\omega}_i}{m_i l_i}\right)$$

Here $l_1 = 1\,\text{m}$, $l_2 = 0.1\,\text{m}$, $m_c = 1\,\text{kg}$, $m_1 = 0.1\,\text{kg}$, $m_2 = 0.01\,\text{kg}$ and $g = 9.81\,\text{m/s}^2$. This time, the friction is also modeled with coefficients $\mu_c = 5 \cdot 10^{-4}\,\text{N s/m}$ and $\mu_1 = \mu_2 = 2 \cdot 10^{-6}\,\text{N m s}$.

In this setting, the admissible state space was defined by $x \in [-2.4\,\text{m}, 2.4\,\text{m}]$ and $\omega_i \in [-36°, 36°]$. On leaving the admissible state space, the episode ends. Every time step yield a reward of $r_t = 1$. No explicit penalty is given when an episode ends; the agent should learn, based on the lack of future rewards. The agent can choose an action from the range $[-50\,\text{N}, 50\,\text{N}]$ every $0.02\,\text{s}$. Because CMA-ES and NAC need complete episodes, the task was made explicitly episodic by resetting the environment every $20\,\text{s}$. The state feature vector is $\vec{\phi}(s) = (x, \dot{x}, \omega_1, \dot{\omega}_1, \omega_2, \dot{\omega}_2)^T$. All episodes start in $\vec{\phi}(s) = (0, 0, 1°, 0, 0, 0)^T$. The discount factor in the work we compare with was $\gamma = 1$. Since we store a state value function, this would mean that the state values are in principle unbounded. Therefore, we use a discount factor of $\gamma = 0.99$. In this settings, the optimal policies for both versions will be very similar, if not equal. Furthermore, we look at the reward per episode as our performance metric. This performance metric was explicitly optimized by CMA-ES and NAC, while Cacla optimizes the discounted cumulative rewards.

The results by CMA-ES we will compare to are shown in Figure 7.3. In addition, NAC was shown to perform far worse, except if it was initialized close to the optimal policy. Then, after somewhere between 3,000 and 4,000 episodes, the median performance reaches the optimal reward per episode of 1,000 (see Heidrich-Meisner and Igel, 2008, for details).

CMA-ES and NAC were used to train a linear controller, so we will use Cacla to find a linear controller as well. We allow for a bias feature that is

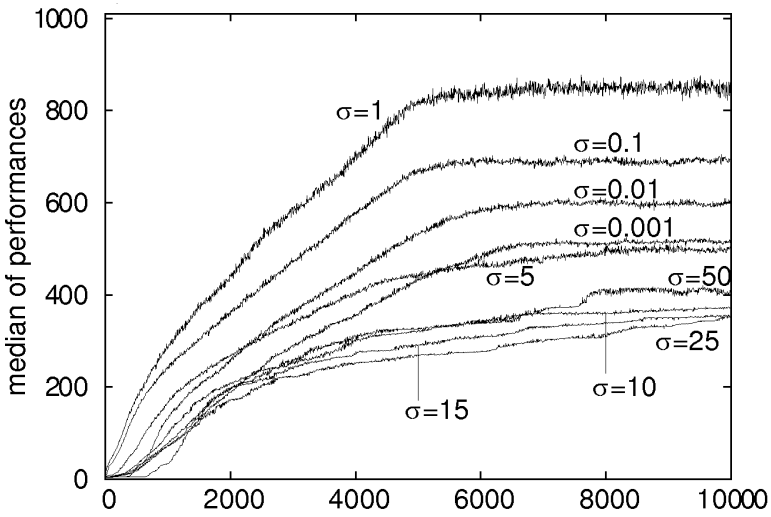

Figure 7.3: Median performance of CMA-ES out of 500 repetitions of the experiment. The x-axis shows the number of episodes. Picture is taken from Heidrich-Meisner and Igel (2008).

always equal to one, so we are looking for a parameter vector $\vec{\psi} \in \mathbb{R}^7$, such that the actor is defined by

$$Ac_t(s_t) = \vec{\psi}_t^T \vec{\phi}(s_t) + = \sum_{i=1}^{6} \psi_{t,i}\phi_i(s_t) + \psi_{t,7} \ .$$

We put in a hard threshold, such that if the output of the actor is larger than 50 N or smaller than −50 N, the agent outputs 50 N or −50 N, respectively. As before, the critic is an MLP with 40 hidden nodes.

We ran Cacla for 500 episodes with learning rates of $\alpha_t = \beta_t = 10^{-3}$ and Gaussian exploration with $\sigma = 500$ or $\sigma = 5000$. The reason for the high exploration is that Cacla effectively learns a bang controller: the resulting actor outputs values far above 50 N and far below −50 N. We also ran Cacla with a uniform random action selection policy in $[-50\,\text{N}, 50\,\text{N}]$ and an ϵ-greedy policy with $\epsilon = 0.1$. This version does not learn a bang controller, because the targets for the actor are always within the admissible range.

Table 7.5 shows the results of our experiments. In Figure 7.3 the median performances are shown, but we show the mean performances. The reason for this is that all exploration types except fully random uniform exploration find perfect policies with and without exploration within 500 training episodes. Therefore, the median results would be equal to 1,000 for all these variants. The Gaussian bang controllers are significantly better than the best median CMA-ES controller after only 500 episodes of learning. In comparison, the

Table 7.5: The results for Cacla with $\alpha = \beta = 10^{-3}$ for the training episodes 401 until 500 (*final online*) and the performance of the greedy policy after 500 episodes of training (*final greedy*). Averaged over 100 repetitions.

final online			
exploration	**mean**	**se**	**success**
$\sigma = 500$	956.4	18.7	94 %
$\sigma = 5000$	973.9	12.4	92 %
$\epsilon = 1$	29.1	0.2	0 %
$\epsilon = 0.1$	808.8	30.1	56 %
final greedy			
exploration	**mean**	**se**	**success**
$\sigma = 500$	969.5	15.4	96 %
$\sigma = 5000$	981.8	11.0	97 %
$\epsilon = 1$	533.4	34.0	29 %
$\epsilon = 0.1$	840.1	33.1	80 %

best CMA-ES controller tops off at about 850 after 5,000 episodes. The differences between $\sigma = 500$ and $\sigma = 5000$ are not significant. Interestingly, even the fully random uniform exploration reaches quite reasonable greedy policies in relatively little time. Naturally, its online performance is poor, because the policy is random. But the greedy performance is much better than the performance of CMA-ES after the same number of episodes. The ϵ-greedy exploration with $\epsilon = 0.1$ additionally reaches quite respectable online performance, even though on average one in every ten actions is fully random and Cacla's actor can only learn from these exploratory actions.

As mentioned before, in Heidrich-Meisner and Igel (2008) NAC was shown to reach a perfect median performance of 1,000 after somewhere between 3,000 and 4,000 episodes. However, this only occurs if NAC is bootstrapped on a policy that is already close to the optimal policy. In other words, considerable domain knowledge is needed to get this level of performance. In contrast, we put no domain knowledge into the Cacla algorithm and its initial controller was initialized with uniformly random parameters between -0.3 and 0.3. Additionally, no attempt was made to optimize this initial range for the parameters. However, for any reasonable exploration—if we consider fully random exploration as unreasonable, because it is too destructive—Cacla already reaches success percentages larger than 50% within the first 500 episodes for both its online and offline performance. As mentioned, this implies that the median performance at this time is already perfect, at 1,000. We did not investigate how fast the success percentage on average surpasses 50%, but the high percentages of the Gaussian exploration versions indicate

that this even happens a lot sooner than after 500 episodes.

This experiment indicates that the relatively simple Cacla algorithm is very effective at solving difficult continuous reinforcement learning problems. In other work, similar results have been published that show that natural gradient and evolutionary algorithms typically need a few thousand episodes to learn a good policy on the double pole, but also on the single pole task (see e.g., Sehnke et al., 2010). Naturally, this does not mean that the other approaches have no meaningful applications in reinforcement learning. For instance, we have only tested Cacla on MDPs. If the state space is partially observable, it may be beneficial to use a recurrent neural network. Then it is still possible to use Cacla to train such a network, for instance with real-time recurrent learning (Williams and Zipser, 1989) or backpropagation through time (Werbos, 2002). However, in such cases direct parameter search may be beneficial. Which method is then preferred is a matter of future research.

7.6.7 Multi-Actor Cacla

The actor of Cacla can potentially get stuck in a local optimum. We conducted some preliminary experiments where we used Cacla with multiple actors that were all initialized differently. Then, one can consider these actors as actions and we use a discrete selector algorithm to choose which actor to use. When an actor is selected, this actor explores and potentially updates, just like in ordinary Cacla.

Here we use QV-learning as the selector, since the state value function that is needed for QV-learning is already present anyway in the critic of Cacla. The additional learning rate for the action values—or in this case actor values is a more appropriate term—was also set to 10^{-3}. Any other discrete algorithm can be considered for the selector.

We conducted a small test on the double pole, using four actors and ϵ-greedy exploration with $\epsilon = 0.1$ for both QV-learning and Cacla. This implies that first QV-learning chooses the actor it deems best with probability 0.9 or a random actor with probability 0.1. Then, the actor that is chosen performs the action it outputs with probability 0.9 and a random uniform action from $[-50\text{N}, 50\text{N}]$ otherwise. If this random action results in a positive temporal-difference error for the value function, the actor is updated towards this action. The same value function was used as critic for all four actors and for the update by QV-learning.

The results for this QV-Cacla algorithm are shown in Table 7.6. Since we made no attempt to tune the algorithm, the results are very promising. The success rate of the online policy is lower that than of Cacla with ϵ-greedy exploration, but we note that since QV-Cacla explores first in the selector and then in the actor, this is to be expected. In one in every ten actions an actor that has only rarely been updated will be chosen and additionally, even when

Table 7.6: The results for QV-Cacla with $\alpha = \beta = 10^{-3}$ for the training episodes 401 until 500 (*final online*) and the performance of the greedy policy after 500 episodes of training (*final greedy*). Averaged over 100 repetitions.

final online			
exploration	**mean**	**se**	**success**
$\epsilon = 0.1$	814.8	26.1	25 %

final greedy			
exploration	**mean**	**se**	**success**
$\epsilon = 0.1$	905.5	24.6	86 %

the best actor is chosen then in one in ten of those time steps a uniformly random action is selected by the actor. This causes more noise than just the ϵ-greedy policy followed by ordinary Cacla. In that light it is all the more impressive that the mean reward, which equals the average number of steps per episode, is actually slightly better for QV-Cacla.

As for the offline performance, 86 out of the 100 runs were successful, compared to 80 out of 100 for the similar Cacla version. More impressively, the average number of steps per episode has gone up from 840.1 to 905.5. The difference is not statistically significant, but recall that we have made no attempt to tune any of the parameters of QV-Cacla to improve on the performance. All in all, the results look promising, although further analyses and more extensive tests of these multi-actor algorithms are needed to reach definitive conclusions. This will be left for future research.

7.7 Conclusion

We will conclude this chapter with a short summary and some pointers to future work.

7.7.1 Summary

We have discussed the existing relevant literature for reinforcement learning in continuous spaces. This includes policy-gradient algorithms and evolutionary algorithms, which can be considered the current state of the art for continuous reinforcement learning. The advantage of these methods is that they can be applied directly to problems where the action space is also continuous.

For problems with a discrete action space, we have discussed how to extend the temporal-difference algorithms that were described in Chapter 5 with function approximation. We discussed linear and non-linear function approximation and have tested a neural network approach on the mountain

car problem. Except for the surprising good performance of Acla, all algorithms were slightly worse than when a simple discretized state space was used, as in Chapter 5. This indicates a first method to deal with continuous spaces: discretize the space and find out if the algorithm of choice performs desirably.

As the state spaces become larger, discretizing these can cause prohibitively large state spaces, or too coarse a discretization to result in good performance. If the action space is small enough, still the temporal-difference algorithms from Chapter 5 can be used with fairly good results. However, if the action space becomes larger it can be better to resort to other methods.

For continuous and large ordinal action spaces, the continuous actor critic learning automaton (Cacla) algorithm was introduced. This algorithm was shown to be competitive to the discrete algorithms even on the mountain car, where there are only three available actions. On the cart pole problem, it eventually performed better than all other algorithms and Acla learned good online policies slightly faster.

Another advantage of Cacla in large ordinal action spaces is that it uses the underlying continuous space to learn from. This is an advantage if the set of actions that can actually be used changes, for instance when switching from a simulation to an actual plant, or when one of the available actions becomes unavailable due to a technical defect. We tested this by removing different sets of actions from the cart pole problem. The discrete algorithms always experienced a set back in performance and this set back was the largest when actions were removed that were used the most often in successful policies. Unfortunately, in real life this will often be the actions that will suffer the most wear and tear and therefore may be the first actions to become unavailable due to this. In contrast, Cacla's performance was barely affected by the removal of actions. This shows that Cacla is very robust to such changes. Incidentally, this is not a feature of Cacla per se, but will hold for any algorithm that learns on the underlying continuous action space, rather than to consider the available actions from an ordinal space as qualitatively different actions, as discrete algorithms tend to do.

Finally, Cacla was compared to a state of the art evolutionary method called Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen et al., 2003; Jiang et al., 2008; Heidrich-Meisner and Igel, 2008) on the double pole problem. This problem is much harder than the single pole problem. In earlier work, CMA-ES was compared favorably to other methods that can be used for reinforcement learning, such as Natural Evolutionary Strategies (NES) (Wierstra et al., 2008; Sun et al., 2009) and Natural Actor Critic (NAC) (Peters and Schaal, 2008a; Bhatnagar et al., 2009). Our results show that Cacla reaches much higher performance levels in a much smaller number of episodes. A reason for this is that Cacla is an online temporal-difference method, whereas the other methods need full episodes to deduce the direction

to update the policy on. In other words, Cacla uses the available experiences more effectively.

7.7.2 Future Work

In order to keep things simple, we have not extended Cacla with numerous possible improvements. These improvements were not necessary to allow the algorithm to perform better than the current state of the art on the double pole task, but for other large problems it may be beneficial to use the extensions we will mention here.

First, Cacla can be extended with eligibility traces. For instance, the value function can be learned with TD(λ) or the new variants TDC and GTD2 (Sutton et al., 2009), extended with eligibility traces. The actor can also be extended with traces that update the actor's output is a certain state a little bit towards the action that was taken there if positive TD errors are observed later, but it is unclear whether this actually improves the performance.

Second, and perhaps more effective, Cacla can be extended with batch updates that make more efficient use of the experiences that were observed in the past. For instance, (incremental) least-squares temporal-difference learning (Bradtke and Barto, 1996; Boyan, 2002; Geramifard et al., 2006) or a variant of (neural) fitted Q-iteration (Ernst et al., 2005; Riedmiller, 2005) can be used.

Third, Cacla can be extended with multiple actors, as in the QV-Cacla algorithm we introduced in Section 7.6.7. The preliminary results there look promising, but still a lot of work has to be done to discover with which discrete algorithm this multi-actor Cacla approach works best and how robust it is to the choice of parameters. This latter point is relevant, since the additional selector will usually introduce at least one additional learning rate and one additional exploration parameter. However, even without tuning these parameters we observed promising results.

Unfortunately, no convergence guarantees are known at this time for Cacla. Although we have mentioned that Cacla performs hill climbing in the action space and that it can get stuck in a local optimum, there is no guarantee that a local optimum in fact is found. Cacla can converge towards good actions quickly, regardless of how small the differences in the value space are, because it uses only the sign of the temporal-difference error rather than its size. However, we have shown in Chapter 5 that this cause Acla to converge towards suboptimal policies. This will also hold for Cacla. However, this is not too limiting on the applicability and the practical use of the algorithm, since in the problems where Cacla works best very limited convergence guarantees can be given anyway. The most advanced results we know of pose restrictions such as a fixed policy and a specific interpolating function approximation scheme for the actor (Szepesvári and Smart, 2004; Antos et al., 2008).

It will be interesting future work to determine under which conditions Cacla can be guaranteed to find good policies and under which conditions Cacla will perhaps perform poorly.

CHAPTER 8

DISCUSSION

In this chapter, we will discuss our general findings and give some pointers for future research. Additionally, we summarize our conclusions with some rules of thumb that can be used to find a good algorithm for many problems, without having to try all possible algorithms in all possible settings or having to select an algorithm without any guidance. First we give a summary of the previous chapters.

8.1 Summary

In Chapters 1 and 2 we introduce and discuss the relevant general concepts for this dissertation, such as the formal definitions of states, actions, rewards and agents. Additionally, we discuss some well-known algorithms, such as value iteration, Q-learning and Sarsa.

In Chapter 3 we show that the value of the maximal element from any set of random values will be a positively biased estimate for the maximal expected value of that set. This bias can be prevented by using a second set of estimates. The resulting double estimator approach can experience a negative bias, but the size of the bias is often smaller.

In Chapter 4 we show that Q-learning suffers from the positive bias that is discussed in Chapter 3. In stochastic problems, Q-learning can suffer from large overestimations of the action values and this may lead to poor policies. This problem is mitigated somewhat by Double Q-learning, that uses two action value functions and the double estimator approach to estimate the value of the best action in the next state. Although Double Q-learning can underestimate the action values, it can find good solutions much faster in some stochastic settings.

In Chapter 5 we look at other alternatives to Q-learning. We discuss Expected Sarsa, which is shown to be an improvement over Sarsa because it has a lower variance in its updates. Expected Sarsa is generalized to General Q-learning. Q-learning, Sarsa and Expected Sarsa can all be considered special cases of this algorithm that uses a behavior policy and an additional estimation policy. We prove convergence of General Q-learning to the optimal policy in the limit under standard conditions. Another algorithm named QV-learning is discussed and we show that this algorithm can sometimes reach good policies faster than Q-learning and Sarsa. Finally, we discuss actor-critic methods and the actor critic learning automaton (Acla) algorithm.

All the algorithms are compared on various problems and we can conclude that for different problems different algorithms perform best. In general, Q-learning performs well on deterministic problems and Double Q-learning performs well on many stochastic settings. QV-learning is one of the more consistent algorithms and Expected Sarsa also performs well in many occasions.

In Chapter 6 we discuss various ways to combine different reinforcement learning algorithms in ensembles. In contrast to most earlier work, we consider methods that can be used to combine the policies of the algorithms, rather than combining the action values. This allows us to construct ensembles that include algorithms that store action values as well as algorithms that store preference values. Even algorithms that search the policy space directly or that are based on heuristics can then be added without problems. We conclude that plurality voting often performs well and because it is easy to interpret and easy to implement, it therefore can be considered a practical choice. Interestingly, in at least one case plurality voting is far better than the ensemble method called policy summation, even though policy summation has the same bias and a lower variance in its policy. The reason is that a lower variance in a policy does not imply a better performance, since sometimes some extra noise is needed to break out of poor suboptimal policies.

In Chapter 7 we explain how the previously discussed algorithms can be applied to problems with continuous state spaces through function approximation. Additionally, some current state-of-the-art approaches are discussed, including policy-gradient methods and evolutionary algorithms. Some of these methods can easily be applied to problems with continuous action spaces, whereas this is not trivial for the temporal-difference algorithms from the earlier chapters. We introduce and discuss the model-free temporal-difference learning algorithm called continuous actor-critic learning automaton (Cacla) and show that it is very competitive in the problems we test it on. We consider Cacla to be a very practical choice, since it performs well, computes fast, is easy to implement and easy to interpret. The only drawback is that it can get stuck in local optima in the action space, but this can be mitigated somewhat if the algorithm is extended with multiple actors and an additional learning algorithm to choose between these actors. For instance, if QV-learning is used, the resulting QV-Cacla algorithm needs two additional parameters, but it often reaches better performance levels with the same number of experiences than Cacla and it is more likely to avoid most of the local optima.

8.2 Conclusions

In the introduction of this dissertation, we state that one of the reasons that we mainly look at model-free temporal-difference methods is that these al-

gorithms are fast, easy, widely applicable and that they often perform well. Indeed, the algorithms we discussed all only need a couple of lines of code to implement and their computation speed depends only linearly on the size of the state space. Also, we have seen some good performances on a variety of problems.

However, a major issue is that there are many different methods and it is apparently very problem-dependent which algorithm works best in a particular setting. We have not tuned any of the algorithms specifically for any of the problems we tested them on and we have not incorporated any domain knowledge. Therefore, the differences in performance are most likely structural differences between the algorithms. As such, we cannot really say that one algorithm is truly better than another in general, even if theoretical analyses seem to indicate this.

For instance, in Section 5.5 we compared Q-learning, Sarsa, Expected Sarsa and QV-learning on two very simple problems. On the first problem, QV-learning is better than Expected Sarsa, which is better than Sarsa, while Q-learning performs worst. On the second problem, the ordering of the algorithms is exactly opposite. In these small problems we can analyze why this is the case. As it turns out, the first problem favors algorithms that underestimate the action values, while the second problem favors algorithms that overestimate the action values. However, it may not always be clear beforehand what type of algorithms are preferred for a given problem, especially if the problem is complex and hard to analyze or if it is even not known beforehand what the exact characteristics of the environment are. If in such cases an algorithm is selected based on general theoretic guarantees under slightly different assumptions or based on its performance on an unrelated different problem, there is a good chance that the problem has some characteristics that do not match well with the selected algorithm.

Although fairly general convergence proofs exist for some reinforcement learning algorithms, there are still problems where the proofs do not hold, or where the guarantees these proofs offer are not of enough practical worth. There are ways to solve these issues, at least partially. Below we give some general rules of thumb. These rules can be used to find a reasonable algorithm for a given problem. Additionally, in principle these rules could themselves be incorporated into a meta-algorithm that first determines the type of MDP and then selects the best available algorithm for that problem. Such a meta-algorithm could even switch algorithms during operation and learn which learning algorithm works best in which situation. This could prove to be an important step towards a generally applicable learning algorithm that performs well on almost any problem.

8.3 Rules of Thumb

In this section we give some pointers for whenwhich algorithm shuold be used from the algorithms we have discussed. This does not mean there is a guarantee that the recommended algorithm will perform better than all the other available options, only that we have some evidence that indicates that it is a reasonable choice for this type of problem. Of course, if there is any domain knowledge available, one should try to incorporate as much of this knowledge beforehand. Here we will assume only general properties are known.

As mentioned above, these rules can be used to find a reasonable algorithm for a particular problem, or they could be incorporated into a meta-algorithm. The rules are unavoidably subjective and biased on the experience of the author with these algorithms. However, we believe it is better to have some indications than none at all and these rules can be considered a first step that can be fine-tuned by future research.

We will divide the possible problems one may encounter in a coarse manner. Other divisions are possible, but this can serve as something of a guideline in order to easily choose a fairly good algorithm for any problem. We believe the algorithms that we recommend will perform well enough to be considered a good 'default' option.

8.3.1 Problems with Small State Spaces

If the problem is very simple and can likely be modeled easily or if a model is already known, it can be beneficial to use a model-based algorithm such as value iteration. Possibly, some interaction with the environment can be used to first construct the model if it is not known and then the optimal policy can be found exactly by dynamic programming or by a planning algorithm. For the rest of this section we assume that the environment is either (partially) unknown, or that it is too complex to solve with an exact dynamic programming approach.

When the state and action spaces are both fairly small and the MDP is deterministic, one can use Q-learning with a learning rate equal to one. If the MDP is not deterministic, but the stochasticity is quite low, Q-learning with a slightly lower learning rate is probably still a good choice.

For a small stochastic MDP, the first choice is Double Q-learning. If it is unknown whether the MDP is stochastic or not, and this cannot reliably be checked beforehand, QV-learning or Expected Sarsa are fairly robust options. It is also a good idea to use one of these latter on-policy algorithms if too much exploration during training can result in actions that are costly, for instance because the experiments are done on a physical system that can get damaged if it is not controlled properly. If there is a safe way to solve a problem and there is a more dangerous way that is faster, on-policy algorithms are more

likely to find the first variant, while off-policy algorithms such as Q-learning are more likely to find the second variant.

All the algorithms above can be extended with eligibility traces to speed up learning. From the literature, it seems that values between $\lambda = 0.8$ and $\lambda = 0.9$ usually work well (see for instance Sutton and Barto, 1998). Furthermore, all of the suggested algorithms can be replaced with a plurality voting ensemble that contains at least that algorithm. To be effective, an ensemble should consist of different algorithms. This can mean that algorithms with different update rules such as Q-learning and Sarsa are used, or it can mean different variants of the same algorithm are used, such as General Q-learning with different estimation policies. Also one can consider making an ensemble with multiple versions of the same algorithm with different discount and eligibility parameters. If the action space is large compared to the number of included algorithms it is probably better to use a policy summation ensemble than a plurality voting ensemble, although this needs to be verified empirically in future research.

8.3.2 Problems with Large State Spaces and Small Action Spaces

In problems with large or continuous state spaces, some form of function approximation should be used. If the state space can probably be clustered or discretized while the general problem and solution structures are likely to stay intact, the same algorithms can be used on this discretized state space as discussed in the previous section.

If generalization across the state space is important and it is likely that optimal values and optimal policies change smoothly for small differences in states, it is probably better to use a smooth function approximation. Then, for deterministic MDPs we suggest to use Q-learning or Greedy-GQ (Maei et al., 2010). Greedy-GQ can be transformed into a double estimator variant that we call Greedy General Double Q-learning, or Greedy-GDQ. This variant is probably better in stochastic MDPs, although an extensive empirical verification of this is left for future research. A relatively safe option if little is known about the problem at hand is to use QV-learning with a gradient temporal-difference algorithm for the state value, such as TDC or GTD2 (Sutton et al., 2009).

8.3.3 Problems with Large State and Action Spaces

If the action space is large, but it consists of a large number of qualitatively different actions, we call this a nominal action space. Then, the best option is to use a policy summation ensemble with a relatively large number of agents.

Alternatively, one can try to find clusters in the action space and use any of the relevant methods from the former subsections.

If the action space is large and discrete, but the actions can be interpreted as points on a (possibly multi-dimensional) continuous underlying space, we call this an ordinal action space. Then, it is probably best to use a continuous-action algorithm on the underlying space. The output of the algorithm can then be rounded to the nearest available action. A good algorithm for continuous spaces is the Cacla algorithm. If the action space is likely to contain many locally optimal policies, it is a good idea to extend Cacla with multiple actors and use an additional selection algorithm, as in QV-Cacla. The value estimates in Cacla and QV-Cacla can benefit from eligibility traces for faster convergence.

For truly continuous action spaces the same suggestions hold as in ordinal action spaces, since we treat these spaces in a similar way. If it is known beforehand that good policies contain many discontinuities, it may be better to discretize the action space and to use a discrete action algorithm on the discretized space. For this, tile coding or adaptive tile coding (Sherstov and Stone, 2005) can be considered.

8.3.4 In Summary

The rules of thumb we discussed are summarized in Table 8.1. The abbreviations used in the table can be found in Table 8.2. Naturally, in most specific cases better algorithms exist than the model-free temporal-difference algorithms that are suggested here. We have seen a notable example in Chapter 7, where it was shown that the Acla algorithm performs much better than all other algorithms on the continuous mountain car task. However, the algorithms that are shown in the table have been shown to perform well on a fairly large number of different settings.

For many types of problems, Table 8.1 lists several alternatives. For instance, the first line indicates that in an MDP with a small (indicated by 's') state space, a small ('s') action space and deterministic ('dt.') transitions and rewards it is advised to use either Q-learning ('Q') or plurality voting with Q-learning, Expected Sarsa, QV-learning and perhaps other algorithms ('PV with Q, ES, QV, . . . '). In problems with large or continuous state spaces ('L'), it is advised to use some form of function approximation. This is indicated with the letters FA in the subscript of the algorithms. Which type of function approximation is useful is problem dependent. For instance, if the space is likely to contain many discontinuities it is advised to use tile coding rather than a smooth function approximator, as discussed earlier. These recommendations are not shown in the table to avoid making it too large and cluttered.

Table 8.1: Suggestions for which model-free temporal-difference algorithms to use in different problem classes. X_{FA} indicates that algorithm X should be combined with some form of function approximation.

S	A	P,R	**Recommended algorithm**
s	s	dt.	Q **or** PV with Q, ES, QV, ...
s	s	st.	DQ **or** PV with DQ, ES, QV, ...
s	s	-	QV **or** ES **or** PV with Q, DQ, ES, QV, ...
L	s	dt.	GQ_{FA}
L	s	st.	GDQ_{FA} **or** QV_{FA} with TDC
L	s	-	QV_{FA} with TDC
L	N	all	PS with Q_{FA}, DQ_{FA}, ES_{FA}, QV_{FA}, ...
L	O	all	Cacla **or** QV-Cacla
L	C	all	Cacla **or** QV-Cacla

8.4 Conclusion

We can expect the suggestions from the former section to become outdated as research continues and further insights are obtain and better algorithms are discovered. If we are very lucky, one day Table 8.1 can be replaced with a single line that states an algorithm that works well in all possible settings. We think it is most likely that such an algorithm would be a meta-algorithm that can choose between different algorithms that are good in different domains. Such an algorithm is especially useful if very little is known about the problem at hand.

A promising approach—and perhaps one of the most important concrete future research steps that can be conducted based on this perspective—is the extension of ensemble algorithms to include weights on the constituting algorithms. These weighted ensembles were shortly discussed in Section 6.6. The idea is to construct an ensemble with many potentially good algorithms—and possibly a fair number of poor ones—and to let the ensemble automatically weigh the algorithms based on their contributions. The resulting ensemble should eventually reach a performance level that is at least as good as the best individual algorithm that was included.

The ensemble methods we discussed are fairly light in terms of their computational overhead and the complexity of such an approach will be the same as the complexity of the slowest algorithm that is included. This implies that the ensemble's speed can be linear in the size of the observations if only temporal-difference methods and other similarly fast algorithms are used. The selection process that learns the proper weights for the different algorithms might cause a slight overhead in terms of required experiences before good policies are found, but the results in Chapter 6 indicate that this over-

Table 8.2: Abbreviations used in Table 8.1.

MDP	**Symbol**
State space	S
Action space	A
State transition function	P
Reward distribution function	R

Property	**Symbol**
Small space	s
Large or continuous state space	L
Large nominal action space	N
Large ordinal action space	O
Continuous action space	C
Deterministic MDP	dt.
Stochastic MDP	st.
Unknown	-

Algorithm	**Symbol**
Q-learning	Q
Double Q-learning	DQ
Sarsa	S
Expected Sarsa	ES
General Q-learning	GQ
General Double Q-learning	GDQ
QV-learning	QV
Plurality voting ensemble	PV
Policy summation ensemble	PA
Algorithm X with function approximation	X_{FA}

head is very light as well. In any case, a small overhead is a small price to pay to avoid the chance of selecting an algorithm that is far below optimal for the problem that you want to solve.

In the meantime, this dissertation can serve as a step towards better understanding of the algorithms that we discussed. Already, the existing reinforcement learning algorithms that we discussed can solve many interesting problems if they are applied correctly. As such, we feel the focus of the research field should not lie only on performance, but also on practical usability of the algorithms. We hope this dissertation can help towards this goal by indicating some pitfalls and pointing towards promising directions for future research.

PUBLICATIONS BY THE AUTHOR

This section lists the publications by the author and gives a short explanation of how these publications correspond to various parts of the dissertation. Many sections of this dissertation are either completely new or only very lightly based on previously published work. The sections that are more heavily inspired or based on earlier publications are discussed below.

The single estimator and the double estimator from Sections 3.3 and 3.4 are described in condensed form in van Hasselt (2010). This paper mentions the overestimation of Q-learning—which is the topic of Chapter 3—and it introduces Double Q-learning. There is significant overlap between the paper and Sections 4.4 and 4.5.

Expected Sarsa was rigorously compared to Q-learning and Sarsa in van Seijen et al. (2009). Our discussion in Chapter 5 is based loosely on this paper and includes the variance comparison to Sarsa that was published therein. In Chapter 5 we shortly mention the possibility for other algorithms related to QV-learning. These algorithms were investigated in Wiering and van Hasselt (2009). Acla was introduced in Wiering and van Hasselt (2007), along with QV-learning that had been proposed earlier (Wiering, 2005). The version of Acla in Chapter 5 is a slightly simplified and improved version of the original algorithm that was first published in van Hasselt and Wiering (2007a).

Chapter 6 is based on some of the ideas in Wiering and van Hasselt (2008), but greatly extends this paper with a better grounding in voting theory, more ensemble methods, a more thorough analysis and more pointers to relevant previous work.

The Cacla algorithm that we discussed in Chapter 7 was first introduced in van Hasselt and Wiering (2007a). In that paper it was shown that Cacla works better than a few other continuous algorithms and that it is beneficial for the performance to use the sign of the temporal-difference error, rather than its size. Both Cacla and Acla are compared to other discrete algorithms on the mountain car and the cart pole in van Hasselt and Wiering (2009). The results on these tasks that are shown in section 7.6 are taken from this paper. QV-Cacla and the results on the double pole task are new to this dissertation.

A related paper of which we barely used any material shows the convergence to the optimal policy of a temporal-difference method where the actor can use a model to look one step in the future (van Hasselt and Wiering, 2007b). We left this proof out of the dissertation, because we mainly focus on model-free approaches.

Finally, there are two papers of which we included no material, because

we feel the subject is largely orthogonal. Both these papers discuss the use of learning mechanisms to adapt so-called serious games to the player (Westra et al., 2009a,b). For instance, this can be usefully applied to a training simulation, to adapt the difficulty of the training to the level of the trainee, such that the training has the most positive effect on the learning experience.

H. P. van Hasselt. Double Q-Learning. In *Advances in Neural Information Processing Systems*, volume 23. The MIT Press, 2010.

H. P. van Hasselt and M. A. Wiering. Using continuous action spaces to solve discrete problems. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN 2009)*, pages 1149–1156, 2009.

J. Westra, H. P. van Hasselt, F. Dignum, and V. Dignum. Adaptive serious games using agent organizations. In *Agents for Games and Simulations, Trends in Techniques, Concepts and Design, Proceedings of the First International Workshop on Agents for Games and Simulations (AGS-2009)*, pages 206–220, 2009a.

H. van Seijen, H. P. van Hasselt, S. Whiteson, and M. A. Wiering. A theoretical and empirical analysis of Expected Sarsa. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 177–184, 2009.

M. A. Wiering and H. P. van Hasselt. The QV family compared to other reinforcement learning algorithms. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 101–108, 2009.

J. Westra, H. P. van Hasselt, V. Dignum, and F. Dignum. On-line adapting games using agent organizations. In *Proceedings of the IEEE Symposium On Computational Intelligence and Games (CIG-08)*, pages 243–250. IEEE, 2009b.

M. A. Wiering and H. P. van Hasselt. Ensemble algorithms in reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38 (4):930–936, 2008.

M. A. Wiering and H. P. van Hasselt. Two novel on-policy reinforcement learning algorithms based on TD(λ)-methods. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-07)*, pages 280–287, 2007.

H. P. van Hasselt and M. A. Wiering. Reinforcement learning in continuous action spaces. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-07)*, pages 272–279, 2007a.

H. P. van Hasselt and M. A. Wiering. Convergence of model-based temporal difference learning for control. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-07)*, pages 60–67, 2007b.

SIKS DISSERTATIONS

1998-1 **Johan van den Akker** (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 **Floris Wiesman** (UM) Information Retrieval by Graphically Browsing Meta-Information

1998-3 **Ans Steuten** (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 **Dennis Breuker** (UM) Memory versus Search in Games

1998-5 **E.W. Oskamp** (RUL) Computerondersteuning bij Straftoemeting

1999-1 **Mark Sloof** (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 **Rob Potharst** (EUR) Classification using decision trees and neural nets

1999-3 **Don Beal** (UM) The Nature of Minimax Search

1999-4 **Jacques Penders** (UM) The practical Art of Moving Physical Objects

1999-5 **Aldo de Moor** (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 **Niek J.E. Wijngaards** (VU) Re-design of compositional systems

1999-7 **David Spelt** (UT) Verification support for object database design

1999-8 **Jacques H.J. Lenting** (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 **Frank Niessink** (VU) Perspectives on Improving Software Maintenance

2000-2 **Koen Holtman** (TUE) Prototyping of CMS Storage Management

2000-3 **Carolien M.T. Metselaar** (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 **Geert de Haan** (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 **Ruud van der Pol** (UM) Knowledge-based Query Formulation in Information Retrieval.

2000-6 **Rogier van Eijk** (UU) Programming Languages for Agent Communication

2000-7 **Niels Peek** (UU) Decision-theoretic Planning of Clinical Patient Management

2000-8 **Veerle Coup** (EUR) Sensitivity Analyis of Decision-Theoretic Networks

2000-9 **Florian Waas** (CWI) Principles of Probabilistic Query Optimization

2000-10 **Niels Nes** (CWI) Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 **Jonas Karlsson** (CWI) Scalable Distributed Data Structures for Database Management

2001-1 **Silja Renooij** (UU) Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 **Koen Hindriks** (UU) Agent Programming Languages: Programming with Mental Models

2001-3 **Maarten van Someren** (UvA) Learning as problem solving

2001-4 **Evgueni Smirnov** (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 **Jacco van Ossenbruggen** (VU) Processing Structured Hypermedia: A Matter of Style

2001-6 **Martijn van Welie** (VU) Task-based User Interface Design

2001-7 **Bastiaan Schonhage** (VU) Diva: Architectural Perspectives on Information Visualization

2001-8 **Pascal van Eck** (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 **Pieter Jan 't Hoen** (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 **Maarten Sierhuis** (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 **Tom M. van Engers** (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 **Nico Lassing** (VU) Architecture-Level Modifiability Analysis

2002-02 **Roelof van Zwol** (UT) Modelling and searching web-based document collections

2002-03 **Henk Ernst Blok** (UT) Database Optimization Aspects for Information Retrieval

2002-04 **Juan Roberto Castelo Valdueza** (UU) The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 **Radu Serban** (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 **Laurens Mommers** (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 **Peter Boncz** (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 **Jaap Gordijn** (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 **Willem-Jan van den Heuvel** (KUB) Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 **Brian Sheppard** (UM) Towards Perfect Play of Scrabble

2002-11 **Wouter C.A. Wijngaards** (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 **Albrecht Schmidt** (Uva) Processing XML in Database Systems

2002-13 **Hongjing Wu** (TUE) A Reference Architecture for Adaptive Hypermedia Applications

2002-14 **Wieke de Vries** (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 **Rik Eshuis** (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 **Pieter van Langen** (VU) The Anatomy of Design: Foundations, Models and Applications

2002-17 **Stefan Manegold** (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 **Heiner Stuckenschmidt** (VU) Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 **Jan Broersen** (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03 **Martijn Schuemie** (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 **Milan Petkovic** (UT) Content-Based Video Retrieval Supported by Database Technology

2003-05 **Jos Lehmann** (UVA) Causation in Artificial Intelligence and Law - A modelling approach

2003-06 **Boris van Schooten** (UT) Development and specification of virtual environments

2003-07 **Machiel Jansen** (UvA) Formal Explorations of Knowledge Intensive Tasks

2003-08 **Yongping Ran** (UM) Repair Based Scheduling

2003-09 **Rens Kortmann** (UM) The resolution of visually guided behaviour

2003-10 **Andreas Lincke** (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 **Simon Keizer** (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 **Roeland Ordelman** (UT) Dutch speech recognition in multimedia information retrieval

2003-13 **Jeroen Donkers** (UM) Nosce Hostem - Searching with Opponent Models

2003-14 **Stijn Hoppenbrouwers** (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 **Mathijs de Weerdt** (TUD) Plan Merging in Multi-Agent Systems

2003-16 **Menzo Windhouwer** (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 **David Jansen** (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 **Levente Kocsis** (UM) Learning Search Decisions

2004-01 **Virginia Dignum** (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 **Lai Xu** (UvT) Monitoring Multi-party Contracts for E-business

2004-03 **Perry Groot** (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 **Chris van Aart** (UVA) Organizational Principles for Multi-Agent Architectures

2004-05 **Viara Popova** (EUR) Knowledge discovery and monotonicity

2004-06 **Bart-Jan Hommes** (TUD) The Evaluation of Business Process Modeling Techniques

2004-07 **Elise Boltjes** (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 **Joop Verbeek** (UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 **Martin Caminada** (VU) For the Sake of the Argument; explorations into argument-based reasoning

2004-10 **Suzanne Kabel** (UVA) Knowledge-rich indexing of learning-objects

2004-11 **Michel Klein** (VU) Change Management for Distributed Ontologies

2004-12 **The Duy Bui** (UT) Creating emotions and facial expressions for embodied agents

2004-13 **Wojciech Jamroga** (UT) Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 **Paul Harrenstein** (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 **Arno Knobbe** (UU) Multi-Relational Data Mining

2004-16 **Federico Divina** (VU) Hybrid Genetic Relational Search for Inductive Learning

2004-17 **Mark Winands** (UM) Informed Search in Complex Games

2004-18 **Vania Bessa Machado** (UvA) Supporting the Construction of Qualitative Knowledge Models

2004-19 **Thijs Westerveld** (UT) Using generative probabilistic models for multimedia retrieval

2004-20 **Madelon Evers** (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005-01 **Floor Verdenius** (UVA) Methodological Aspects of Designing Induction-Based Applications

2005-02 **Erik van der Werf** (UM)) AI techniques for the game of Go

2005-03 **Franc Grootjen** (RUN) A Pragmatic Approach to the Conceptualisation of Language

2005-04 **Nirvana Meratnia** (UT) Towards Database Support for Moving Object data

2005-05 **Gabriel Infante-Lopez** (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 **Pieter Spronck** (UM) Adaptive Game AI

2005-07 **Flavius Frasincar** (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 **Richard Vdovjak** (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 **Jeen Broekstra** (VU) Storage, Querying and Inferencing for Semantic Web Languages

2005-10 **Anders Bouwer** (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 **Elth Ogston** (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 **Csaba Boer** (EUR) Distributed Simulation in Industry

2005-13 **Fred Hamburg** (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 **Borys Omelayenko** (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 **Tibor Bosse** (VU) Analysis of the Dynamics of Cognitive Processes

2005-16 **Joris Graaumans** (UU) Usability of XML Query Languages

2005-17 **Boris Shishkov** (TUD) Software Specification Based on Re-usable Business Components

2005-18 **Danielle Sent** (UU) Test-selection strategies for probabilistic networks

2005-19 **Michel van Dartel** (UM) Situated Representation

2005-20 **Cristina Coteanu** (UL) Cyber Consumer Law, State of the Art and Perspectives

2005-21 **Wijnand Derks** (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 **Samuil Angelov** (TUE) Foundations of B2B Electronic Contracting

2006-02 **Cristina Chisalita** (VU) Contextual issues in the design and use of information technology in organizations

2006-03 **Noor Christoph** (UVA) The role of metacognitive skills in learning to solve problems

2006-04 **Marta Sabou** (VU) Building Web Service Ontologies

2006-05 **Cees Pierik** (UU) Validation Techniques for Object-Oriented Proof Outlines

2006-06 **Ziv Baida** (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 **Marko Smiljanic** (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 **Eelco Herder** (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 **Mohamed Wahdan** (UM) Automatic Formulation of the Auditor's Opinion

2006-10 **Ronny Siebes** (VU) Semantic Routing in Peer-to-Peer Systems

2006-11 **Joeri van Ruth** (UT) Flattening Queries over Nested Data Types

2006-12 **Bert Bongers** (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 **Henk-Jan Lebbink** (UU) Dialogue and Decision Games for Information Exchanging Agents

2006-14 **Johan Hoorn** (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of

Requirements Change

2006-15 **Rainer Malik** (UU) CONAN: Text Mining in the Biomedical Domain

2006-16 **Carsten Riggelsen** (UU) Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 **Stacey Nagata** (UU) User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 **Valentin Zhizhkun** (UVA) Graph transformation for Natural Language Processing

2006-19 **Birna van Riemsdijk** (UU) Cognitive Agent Programming: A Semantic Approach

2006-20 **Marina Velikova** (UvT) Monotone models for prediction in data mining

2006-21 **Bas van Gils** (RUN) Aptness on the Web

2006-22 **Paul de Vrieze** (RUN) Fundaments of Adaptive Personalisation

2006-23 **Ion Juvina** (UU) Development of Cognitive Model for Navigating on the Web

2006-24 **Laura Hollink** (VU) Semantic Annotation for Retrieval of Visual Resources

2006-25 **Madalina Drugan** (UU) Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 **Vojkan Mihajlovic** (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 **Stefano Bocconi** (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 **Borkur Sigurbjornsson** (UVA) Focused Information Access using XML Element Retrieval

2007-01 **Kees Leune** (UvT) Access Control and Service-Oriented Architectures

2007-02 **Wouter Teepe** (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 **Peter Mika** (VU) Social Networks and the Semantic Web

2007-04 **Jurriaan van Diggelen** (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 **Bart Schermer** (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 **Gilad Mishne** (UVA) Applied Text Analytics for Blogs

2007-07 **Natasa Jovanovic** (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 **Mark Hoogendoorn** (VU) Modeling of Change in Multi-Agent Organizations

2007-09 **David Mobach** (VU) Agent-Based Mediated Service Negotiation

2007-10 **Huib Aldewereld** (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 **Natalia Stash** (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 **Marcel van Gerven** (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 **Rutger Rienks** (UT) Meetings in Smart Environments; Implications of Progressing Technology

2007-14 **Niek Bergboer** (UM) Context-Based Image Analysis

2007-15 **Joyca Lacroix** (UM) NIM: a Situated Computational Memory Model

2007-16 **Davide Grossi** (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 **Theodore Charitos** (UU) Reasoning with Dynamic Networks in Practice

2007-18 **Bart Orriens** (UvT) On the development an management of adaptive business collaborations

2007-19 **David Levy** (UM) Intimate relationships with artificial partners

2007-20 **Slinger Jansen** (UU) Customer Configuration Updating in a Software Supply Network

2007-21 **Karianne Vermaas** (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 **Zlatko Zlatev** (UT) Goal-oriented design of value and process models from patterns

2007-23 **Peter Barna** (TUE) Specification of Application Logic in Web Information Systems

2007-24 **Georgina Ramírez Camps** (CWI) Structural Features in XML Retrieval

2007-25 **Joost Schalken** (VU) Empirical Investigations in Software Process Improvement

2008-01 **Katalin Boer-Sorbán** (EUR) Agent-Based Simulation of Financial Markets: A modular,continuous-time approach

2008-02 **Alexei Sharpanskykh** (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 **Vera Hollink** (UVA) Optimizing hierarchical menus: a usage-based approach

2008-04 **Ander de Keijzer** (UT) Management of Uncertain Data - towards unattended integration

2008-05 **Bela Mutschler** (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 **Arjen Hommersom** (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 **Peter van Rosmalen** (OU) Supporting the tutor in the design and support of adaptive e-learning

2008-08 **Janneke Bolt** (UU) Bayesian Networks: Aspects of Approximate Inference

2008-09 **Christof van Nimwegen** (UU) The paradox of the guided user: assistance can be counter-effective

2008-10 **Wauter Bosma** (UT) Discourse oriented summarization

2008-11 **Vera Kartseva** (VU) Designing Controls for Network Organizations: A Value-Based Approach

2008-12 **Jozsef Farkas** (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 **Caterina Carraciolo** (UVA) Topic Driven Access to Scientific Handbooks

2008-14 **Arthur van Bunningen** (UT) Context-Aware Querying; Better Answers with Less Effort

2008-15 **Martijn van Otterlo** (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 **Henriette van Vugt** (VU) Embodied agents from a user's perspective

2008-17 **Martin Op 't Land** (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 **Guido de Croon** (UM) Adaptive Active Vision

2008-19 **Henning Rode** (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 **Rex Arendsen** (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 **Krisztian Balog** (UVA) People Search in the Enterprise

2008-22 **Henk Koning** (UU) Communication of IT-Architecture

2008-23 **Stefan Visscher** (UU) Bayesian network models for the management of ventilator-associated pneumonia

2008-24 **Zharko Aleksovski** (VU) Using background knowledge in ontology matching

2008-25 **Geert Jonker** (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair

using Spender-signed Currency

2008-26 **Marijn Huijbregts** (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 **Hubert Vogten** (OU) Design and Implementation Strategies for IMS Learning Design

2008-28 **Ildiko Flesch** (RUN) On the Use of Independence Relations in Bayesian Networks

2008-29 **Dennis Reidsma** (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 **Wouter van Atteveldt** (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 **Loes Braun** (UM) Pro-Active Medical Information Retrieval

2008-32 **Trung H. Bui** (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 **Frank Terpstra** (UVA) Scientific Workflow Design; theoretical and practical issues

2008-34 **Jeroen de Knijf** (UU) Studies in Frequent Tree Mining

2008-35 **Ben Torben Nielsen** (UvT) Dendritic morphologies: function shapes structure

2009-01 **Rasa Jurgelenaite** (RUN) Symmetric Causal Independence Models

2009-02 **Willem Robert van Hage** (VU) Evaluating Ontology-Alignment Techniques

2009-03 **Hans Stol** (UvT) A Framework for Evidence-based Policy Making Using IT

2009-04 **Josephine Nabukenya** (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 **Sietse Overbeek** (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 **Muhammad Subianto** (UU) Understanding Classification

2009-07 **Ronald Poppe** (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 **Volker Nannen** (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 **Benjamin Kanagwa** (RUN) Design, Discovery and Construction of Service-oriented Systems

2009-10 **Jan Wielemaker** (UVA) Logic programming for knowledge-intensive interactive applications

2009-11 **Alexander Boer** (UVA) Legal Theory, Sources of Law & the Semantic Web

2009-12 **Peter Massuthe** (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services

2009-13 **Steven de Jong** (UM) Fairness in Multi-Agent Systems

2009-14 **Maksym Korotkiy** (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 **Rinke Hoekstra** (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 **Fritz Reul** (UvT) New Architectures in Computer Chess

2009-17 **Laurens van der Maaten** (UvT) Feature Extraction from Visual Data

2009-18 **Fabian Groffen** (CWI) Armada, An Evolving Database System

2009-19 **Valentin Robu** (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 **Bob van der Vecht** (UU) Adjustable Autonomy: Controling Influences on Decision Making

2009-21 **Stijn Vanderlooy** (UM) Ranking and Reliable Classification

2009-22 **Pavel Serdyukov** (UT) Search For Expertise: Going beyond direct evidence

2009-23 **Peter Hofgesang** (VU) Modelling Web Usage in a Changing Environment

2009-24 **Annerieke Heuvelink** (VUA) Cognitive Models for Training Simulations

2009-25 **Alex van Ballegooij** (CWI) RAM: Array Database Management through Relational Mapping

2009-26 **Fernando Koch** (UU) An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 **Christian Glahn** (OU) Contextual Support of social Engagement and Reflection on the Web

2009-28 **Sander Evers** (UT) Sensor Data Management with Probabilistic Models

2009-29 **Stanislav Pokraev** (UT) Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 **Marcin Zukowski** (CWI) Balancing vectorized query execution with bandwidth-optimized storage

2009-31 **Sofiya Katrenko** (UVA) A Closer Look at Learning Relations from Text

2009-32 **Rik Farenhorst** (VU) and **Remco de Boer** (VU) Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 **Khiet Truong** (UT) How Does Real Affect Affect Affect Recognition In Speech?

2009-34 **Inge van de Weerd** (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 **Wouter Koelewijn** (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 **Marco Kalz** (OUN) Placement Support for Learners in Learning Networks

2009-37 **Hendrik Drachsler** (OUN) Navigation Support for Learners in Informal Learning Networks

2009-38 **Riina Vuorikari** (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 **Christian Stahl** (TUE, Humboldt-Universitaet zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 **Stephan Raaijmakers** (UvT) Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 **Igor Berezhnyy** (UvT) Digital Analysis of Paintings

2009-42 **Toine Bogers** Recommender Systems for Social Bookmarking

2009-43 **Virginia Nunes Leal Franqueira** (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 **Roberto Santana Tapia** (UT) Assessing Business-IT Alignment in Networked Organizations

2009-45 **Jilles Vreeken** (UU) Making Pattern Mining Useful

2009-46 **Loredana Afanasiev** (UvA) Querying XML: Benchmarks and Recursion

2010-01 **Matthijs van Leeuwen** (UU) Patterns that Matter

2010-02 **Ingo Wassink** (UT) Work flows in Life Science

2010-03 **Joost Geurts** (CWI) A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 **Olga Kulyk** (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 **Claudia Hauff** (UT) Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 **Sander Bakkes** (UvT) Rapid Adaptation of Video Game AI

2010-07 **Wim Fikkert** (UT) Gesture interaction at a Distance

2010-08 **Krzysztof Siewicz** (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 **Hugo Kielman** (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborg-

ing

2010-10 **Rebecca Ong** (UL) Mobile Communication and Protection of Children

2010-11 **Adriaan Ter Mors** (TUD) The world according to MARP: Multi-Agent Route Planning

2010-12 **Susan van den Braak** (UU) Sensemaking software for crime analysis

2010-13 **Gianluigi Folino** (RUN) High Performance Data Mining using Bio-inspired techniques

2010-14 **Sander van Splunter** (VU) Automated Web Service Reconfiguration

2010-15 **Lianne Bodenstaff** (UT) Managing Dependency Relations in Inter-Organizational Models

2010-16 **Sicco Verwer** (TUD) Efficient Identification of Timed Automata, theory and practice

2010-17 **Spyros Kotoulas** (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 **Charlotte Gerritsen** (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 **Henriette Cramer** (UvA) People's Responses to Autonomous and Adaptive Systems

2010-20 **Ivo Swartjes** (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 **Harold van Heerde** (UT) Privacy-aware data management by means of data degradation

2010-22 **Michiel Hildebrand** (CWI) End-user Support for Access to Heterogeneous Linked Data

2010-23 **Bas Steunebrink** (UU) The Logical Structure of Emotions

2010-24 **Dmytro Tykhonov** Designing Generic and Efficient Negotiation Strategies

2010-25 **Zulfiqar Ali Memon** (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 **Ying Zhang** (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 **Marten Voulon** (UL) Automatisch contracteren

2010-28 **Arne Koopman** (UU) Characteristic Relational Patterns

2010-29 **Stratos Idreos** (CWI) Database Cracking: Towards Auto-tuning Database Kernels

2010-30 **Marieke van Erp** (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 **Victor de Boer** (UVA) Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 **Marcel Hiel** (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 **Robin Aly** (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 **Teduh Dirgahayu** (UT) Interaction Design in Service Compositions

2010-35 **Dolf Trieschnigg** (UT) Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 **Jose Janssen** (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 **Niels Lohmann** (TUE) Correctness of services and their composition

2010-38 **Dirk Fahland** (TUE) From Scenarios to components

2010-39 **Ghazanfar Farooq Siddiqui** (VU) Integrative modeling of emotions in virtual agents

2010-40 **Mark van Assem** (VU) Converting and Integrating Vocabularies for the Semantic Web

2010-41 **Guillaume Chaslot** (UM) Monte-Carlo Tree Search

2010-42 **Sybren de Kinderen** (VU) Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach

2010-43 **Peter van Kranenburg** (UU) A Computational Approach to Content-Based Retrieval of Folk

Song Melodies

2010-44 **Pieter Bellekens** (TUE) An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain

2010-45 **Vasilios Andrikopoulos** (UvT) A theory and model for the evolution of software services

2010-46 **Vincent Pijpers** (VU) e3alignment: Exploring Inter-Organizational Business-ICT Alignment

2010-47 **Chen Li** (UT) Mining Process Model Variants: Challenges, Techniques, Examples

2010-48 **Milan Lovric** (EUR) Behavioral Finance and Agent-Based Artificial Markets

2010-49 **Jahn-Takeshi Saito** (UM) Solving difficult game positions

2010-50 **Bouke Huurnink** (UVA) Search in Audiovisual Broadcast Archives

2010-51 **Alia Khairia Amin** (CWI) Understanding and supporting information seeking tasks in multiple sources

2010-52 **Peter-Paul van Maanen** (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention

2010-53 **Edgar Meij** (UVA) Combining Concepts and Language Models for Information Access

2011-01 **Botond Cseke** (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models

2011-02 **Nick Tinnemeier** (UU) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

2011-03 **Jan Martijn van der Werf** (TUE) Compositional Design and Verification of Component-Based Information Systems

2011-04 **Hado van Hasselt** (UU) Insights in Reinforcement Learning

SAMENVATTING (DUTCH SUMMARY)

Reinforcement learning is een onderzoeksveld waarin systemen die kunnen leren door te interacteren met hun omgeving centraal staan. Denk hierbij aan bijvoorbeeld robots of software programma's die een specifieke taak moeten oplossen, waarvoor de oplossing niet bekend is bij de ontwerper van het programma, maar er wel feedback uit de omgeving beschikbaar is over de waarde van gekozen acties. Een aanpak die goed blijkt te werken is het opslaan van de waarde van iedere actie in elke situatie op basis van de feedback die uit de omgeving verkregen wordt. Het bepalen van deze waardes kan echter lastig zijn, bijvoorbeeld omdat een actie een beloning tot gevolg heeft die pas enige tijd later waargenomen kan worden. Reinforcement learning houdt zich bezig met het onderzoeken en ontwerpen van automatische algoritmes dit op basis van interactie met de omgeving goede acties kunnen leren voor iedere mogelijke situatie. Er zijn vele voorbeelden van problemen die hiermee opgelost kunnen worden. Bijvoorbeeld in de robotica kan een robot bepaalde taken leren uitvoeren op basis van feedback van buitenaf. Ook kunnen software programma's ontworpen worden die automatisch kunnen leren om spelletjes te leren. Zo is er bijvoorbeeld met behulp van reinforcement learning een backgammonspeler ontworpen die net zo goed speelt als de beste menselijke spelers, terwijl de programmeur van deze speler geen goede kennis van goede backgammonstrategieën hoeft te hebben.

Dit proefschrift gaat met name over zogenaamde model-free temporal-difference learning algoritmes. Deze algoritmes zijn een specifiek type reinforcement learning algoritmes. De term *model-free* impliceert dat deze algoritmes niet een compleet model van de omgeving construeren, om hierna dit model te proberen op te lossen. Een voordeel hiervan is dat er vaak vrij veel ruimte nodig is om zo'n model op te slaan. Ook is het in sommige gevallen veel moeilijker om het model met genoeg zekerheid te bepalen, terwijl goede acties en strategieën vaak makkelijker te bepalen zijn. De algoritmes heten *temporal-difference* algoritmes, omdat ze bij het leren gebruik maken van eerder opgeslagen informatie. Dat werkt als volgt. Als het systeem een actie kiest en deze uitvoert zal het systeem de feedback van de omgeving waarnemen en de nieuwe situatie die het gevolg is van de actie. Een temporal-difference algoritme zal dan de zojuist genomen actie waarderen aan de hand van de directe feedback èn de huidige schatting van de waarde van de nieuwe situatie. Dit betekent bijvoorbeeld dat er niet gewacht hoeft te worden totdat een spel is afgelopen voordat het systeem beter kan leren spelen op basis van de in de tussentijd verkregen feedback.

Er zijn vele mogelijke varianten van model-free temporal-difference learning algoritmes. In dit proefschrift demonstreren we dat een veel gebruikt algoritme, genaamd Q-learning, last kan hebben van enorme overschattingen in bepaalde problemen. Deze overschattingen kunnen tot ongewenst gedrag leiden, zoals de voorkeur om roulette te blijven spelen, zelfs nadat er al vele duizenden dollars verloren zijn. De reden voor deze overschattingen wordt geanalyseerd en een gedeeltelijke opgelost met behulp van die analyse. Echter, het blijkt uit verdere analyses en experimenten dat er tot nog toe geen enkel algoritme bestaat dat heel goed werkt op alle mogelijke problemen.

Om toch redelijk goede resultaten te behalen op een probleem waarvan onbekend is welk algoritme daar het meest voor geschikt is, wordt aanbevolen om een ensemble te maken met verschillende algoritmes. Er wordt aangetoond dat zo'n ensemble vaak bijna zo goed werkt als het best presterende algoritme dat erin is opgenomen. In sommige gevallen werkt het ensemble zelf beter dan het beste losstaande algoritme. Dit geeft aan dat dit een interessante richting is voor verder onderzoek.

In het proefschrift wordt ook onderzocht hoe goed reinforcement learning algoritmes functioneren in omgevingen waarbij de situaties en acties continu zijn. In zulke continue problemen zijn er feitelijk oneindig veel situaties en acties mogelijk. Oudere reinforcement learning algoritmes zijn echter ontworpen voor problemen met een klein, eindig aantal mogelijke situaties en acties. Na een analyse van de huidige kennis over goede algoritmes om dit soort problemen aan te pakken, wordt een nieuw model-free temporal-difference algoritme geïntroduceerd. Dit algoritme blijkt beter te werken dan de huidig beste algoritmes op een taak waarbij er twee stokjes gebalanceerd moeten worden die met scharnieren aan een kar bevestigd zijn. Aangezien de stokjes gelijktijdig gebalanceerd moeten worden door tegen de kar aan te slaan, is dit een moeilijk dynamisch probleem. De goede resultaten van het nieuwe algoritme zijn erg bemoedigend en geven aan dat deze richting van onderzoek mogelijk kan leiden tot zeer goede algoritmes voor dit soort problemen.

Tot slot wordt een grove onderverdeling gemaakt van alle mogelijke soorten problemen die opgelost kunnen worden met reinforcement learning. Per type probleem wordt een algoritme genoemd dat waarschijnlijk goed werkt als er een probleem van dit type opgelost moet worden. Als verder onderzoek wordt aanbevolen dit soort aanbevelingen op basis van een automatisch meta-algoritme te doen, zodat op basis van algemene eigenschappen van een probleem een geschikt algoritme gekozen kan worden om het probleem op te lossen.

DANKWOORD

Ten eerste wil ik graag Marco Wiering bedanken. Zijn colleges over machine learning—en specifiek over reinforcement learning—waren voor mij een voornaame reden om verder te willen in de wetenschap. Ook wil ik Marco graag samen met John-Jules Meyer en Lambert Schomaker bedanken voor het mogelijk maken van dit promotieonderzoek en voor hun commentaar op eerdere versies van dit proefschrift. Han La Poutré wil ik graag bedanken voor het vertrouwen om mij aan te nemen nog voordat mijn proefschrift geheel voltooid was en voor de geboden kans om in de wetenschap actief te blijven. De leescommissie, bestaande uit Robert Babuska, Frans Groen, Eric Postma, Arno Siebes en Richard Sutton, wil ik bedanken voor het lezen en goedkeuren van mijn proefschrift.

Verder wil ik graag mijn collega's bedanken, onder meer voor de koffie, de borrels, de gezelligheid en het tafeltennissen. Bas, Christiaan, Eric, Frank, Huib, Joost, Liz, Marieke, Michal, Nick, Nieske, Paolo, Susan, Tom, en ieder die ik mogelijkerwijs vergeet, bedankt. Speciaal wil ik ook nog mijn oudkamergenoten Geert en Maaike bedanken en hierbij mijn excuses aanbieden als ik hen incidenteel van het werk heb gehouden als ik zelf even wat minder inspiratie had.

Hessel is verantwoordelijk voor de prachtige kaft van dit proefschrift, waarvoor ik natuurlijk erg dankbaar ben. Verder wil ik hem en mijn andere vrienden bedanken voor hun vriendschap en support, of ze dit nu lieten blijken door het aanhoren van mijn (muzikale) meningen in de band, door mij te laten winnen met squash, door op de katten te passen als we op vakantie waren of op welke andere manier dan ook.

Tot slot wil ik Viola bedanken voor haar liefde, geduld en support, die erg belangrijk voor mij zijn geweest tijdens het schrijven van dit proefschrift. Ik ben trots en gelukkig met haar mijn leven te kunnen delen.

BIBLIOGRAPHY

J. S. Albus. A theory of cerebellar function. *Mathematical Biosciences*, 10: 25–61, 1975a.

J. S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Dynamic Systems, Measurement and Control*, pages 220–227, 1975b.

S. I. Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998. ISSN 0899-7667.

C. W. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

A. Antos, R. Munos, and C. Szepesvari. Fitted Q-iteration in continuous action-space mdps. *Advances in neural information processing systems (NIPS-07)*, 20:9–16, 2008.

K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. part ii: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977. ISSN 0019-2082.

Aristotle. Metaphysics, 350BC.

B. C. Arnold and R. A. Groeneveld. Bounds on expectations of linear systematic statistics based on dependent samples. *The Annals of Statistics*, 7(1): 220–223, 1979.

B. C. Arnold, N. Balakrishnan, and H. N. Nagaraja. *A first course in order statistics*. Society for Industrial Mathematics, 2008.

K. J. Arrow. A difficulty in the concept of social welfare. *The Journal of Political Economy*, 58(4):328–346, 1950.

K. J. Arrow. *Social choice and individual values*. Yale Univ Press, 1963.

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

T. Aven. Upper (lower) bounds on the mean of the maximum (minimum) of a number of random variables. *Journal of applied probability*, 22(3):723–728, 1985.

T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, USA, 1996.

T. Bäck and H. P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

L. Baird. Residual algorithms: Reinforcement learning with function approximation. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 30–37. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

L. C. Baird and A. H. Klopf. Reinforcement learning with high-dimensional, continuous actions. Technical Report WL-TR-93-114, Wright Laboratory, Wright-Patterson Air Force Base, OH, 1993.

S. Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations integrales. *Fundamenta Mathematicae*, 3:133–181, 1922.

M. Bardi and I. C. Dolcetta. *Optimal control and viscosity solutions of Hamilton–Jacobi–Bellman equations*. Springer, 1997.

H. B. Barlow. Unsupervised learning. *Neural Computation*, 1(3):295–311, 1989.

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846, 1983.

J. Baxter and P. L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

R. Beard, G. Saridis, and J. Wen. Approximate solutions to the time-invariant Hamilton–Jacobi–Bellman equation. *Journal of Optimization theory and Applications*, 96(3):589–626, 1998. ISSN 0022-3239.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

H. Benbrahim and J. A. Franklin. Biped dynamic walking using reinforcement learning. *Robotics and Autonomous Systems*, 22(3-4):283–302, 1997. ISSN 0921-8890.

D. A. Berry and B. Fristedt. *Bandit Problems: sequential allocation of experiments*. Chapman and Hall, London/New York, 1985.

D. P. Bertsekas. *Dynamic Programming and Optimal Control, vol. II*. Athena Scientific, 2007.

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

D. P. Bertsekas, V. S. Borkar, and A. Nedic. Improved temporal difference methods with linear function approximation. *Handbook of Learning and Approximate Dynamic Programming*, pages 235–260, 2004.

D. Bertsimas, K. Natarajan, and C. P. Teo. Tight bounds on expected order statistics. *Probability in the Engineering and Informational Sciences*, 20 (04):667–686, 2006.

S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471–2482, 2009. ISSN 0005-1098.

C. M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, USA, 1995.

C. M. Bishop. *Pattern recognition and machine learning*. Springer New York:, 2006.

D. Black. *The theory of committees and elections*. Cambridge: At the University Press, 1958.

L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, L. D. Jackel, Y. LeCun, U. A. Muller, E. Sackinger, P. Simard, and V. Vapnik. Comparison of classifier methods: a case study in handwritten digit recognition. In *Proceedings of the 12th IAPR International Conference on Pattern Recognition, 1994. Vol. 2 - Conference B: Computer Vision & Image Processing.*, volume 2, pages 77–82, 1994.

C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11(1):94, 1999.

J. A. Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2):233–246, 2002. ISSN 0885-6125.

S. J. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57, 1996.

R. I. Brafman and M. Tennenholtz. R-max–a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2003.

S. J. Brams and P. C. Fishburn. Approval voting. *American Political Science Review*, 72:831–847, 1978.

L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47 (1-3):139–159, 1991.

A. Bryson and Y. Ho. *Applied Optimal Control*. Blaisdell Publishing Co., 1969.

N. J. Calkin. A curious binomial identity. *Discrete Mathematics*, 131(1-3): 335–337, 1994.

E. Capen, R. Clapp, and T. Campbell. Bidding in high risk situations. *Journal of Petroleum Technology*, 23:641–653, 1971.

A. R. Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Brown University, Providence, RI, May 1998.

C. E. Clark. The greatest of a finite set of random variables. *Operations Research*, 9(2):145–162, 1961.

K. Conn, C. Liu, N. Sarkar, W. Stone, and Z. Warren. *Towards affect-sensitive assistive intervention technologies for children with autism*, chapter 20, pages 365–390. RS/I-Tech Education and Publishing, 2008.

C. H. Coombs. A theory of data. *Psychological review*, 67(3):143–159, 1960.

R. Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge MA, 1996. MIT Press.

R. H. Crites and A. G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2/3):235–262, 1998.

H. A. David and H. N. Nagaraja. *Order Statistics*. John Wiley & Sons, 3 edition, 2003.

L. Davis. *Handbook of genetic algorithms*. Arden Shakespeare, 1991.

P. Dayan. The convergence of TD(λ) for general lambda. *Machine Learning*, 8:341–362, 1992.

P. Dayan and T. Sejnowski. TD(λ): Convergence with probability 1. *Machine Learning*, 14:295–301, 1994.

J. C. de Borda. Mémoire sur les élections au scrutin. *Histoire de l'académie royale des sciences année 1781*, 2:657–665, 1784.

M. J. A. N. de Caritat, le marquis de Condorcet. Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix, 1785.

A. de Moivre. *The doctrine of chances, or, a method of calculating the probability of events in play*. Printed by W. Pearson, for the Author, London, 1718.

G. Deboeck and T. Kohonen. *Visual Explorations in Finance with self-organizing maps*. Springer New York, 1998.

D. C. Dennett. Cognitive wheels: The frame problem of AI. *Language and Thought*, page 217, 2005.

A. E. Eiben and J. E. Smith. *Introduction to evolutionary computing*. Springer Verlag, 2003.

D. Ernst, M. Glavic, and L. Wehenkel. Power systems stability control: Reinforcement learning framework. *IEEE transactions on power systems*, 19 (1):427–435, 2004.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(1):503–556, 2005.

E. Even-Dar and Y. Mansour. Learning rates for Q-learning. *Journal of Machine Learning Research*, 5:1–25, 2003.

E. Even-Dar, S. Mannor, and Y. Mansour. PAC bounds for multi-armed bandit and Markov decision processes. In *COLT: Proceedings of the Workshop on Computational Learning Theory, Morgan Kaufmann Publishers*, 2002.

D. M. Farrell. *Electoral systems: A comparative introduction*. Wiley Online Library, 2001.

C.-N. Fiechter. Efficient reinforcement learning. In *Proceedings of the seventh annual conference on Computational learning theory COLT '94*, pages 88–97, New York, NY, USA, 1994. ACM.

P. C. Fishburn. Condorcet social choice functions. *SIAM Journal on Applied Mathematics*, pages 469–489, 1977.

P. C. Fishburn. Monotonicity paradoxes in the theory of elections. *Discrete Applied Mathematics*, 4(2):119–134, 1982.

R. A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 222:309–368, 1922.

R. A. Fisher. *Statistical methods for research workers*. Oliver & Boyd, Edinburgh, 1925.

K. Främling. Replacing eligibility trace for action-value learning with function approximation. In *Proceedings of the 15th European Symposium on Artificial Neural Networks (ESANN-07)*, pages 313–318. d-side publishing, 2007.

S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. Müller, M. Wooldridge, and N. Jennings, editors, *Intelligent Agents III Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin / Heidelberg, 1997.

Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the thirteenth International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.

C. Gaskett, D. Wettergreen, and A. Zelinsky. Q-learning in continuous state and action spaces. *Advanced Topics in Artificial Intelligence*, pages 417–428, 1999.

A. Geramifard, M. Bowling, and R. S. Sutton. Incremental least-squares temporal difference learning. In *Proceedings of the 21st national conference on Artificial intelligence-Volume 1*, pages 356–361. AAAI Press, 2006.

T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber. Exponential natural evolution strategies. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 393–400. ACM, 2010.

G. J. Gordon. Stable function approximation in dynamic programming. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning (ICML-95)*, pages 261–268, San Francisco, CA, 1995. Morgan Kaufmann.

G. J. Gordon. *Approximate Solutions to Markov Decision Processes*. PhD thesis, Carnegie Mellon University, 1999.

A. D. Grazia. Mathematical derivation of an election system. *Isis*, 44(1/2): 42–51, 1953. ISSN 00211753.

E. Greensmith, P. L. Bartlett, and J. Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *The Journal of Machine Learning Research*, 5:1471–1530, 2004.

B. Grofman and S. L. Feld. If you like the alternative vote (a.k.a. the instant runoff), then you ought to know about the Coombs rule. *Electoral Studies*, 23(4):641–659, 2004.

S. S. Gupta and S. Panchapakesan. Order statistics arising from independent binomial populations. Technical report, Purdue University, september 1967.

A. Hans and S. Udluft. Ensembles of neural networks for robust reinforcement learning. In *Proceedings of the 9th IEEE International Conference on Machine Learning and Applications*, pages 401–406. IEEE, 2010.

A. Hans, D. Schneegaß, A. M. Schäfer, and S. Udluft. Safe exploration for reinforcement learning. In *Proceedings of the 16th European Symposium on Artificial Neural Networks, (ESANN 2008)*, pages 143–148, 2008.

N. Hansen, S. D. Müller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation*, 11(1):1–18, 2003. ISSN 1063-6560.

T. Hare. *The election of representatives, parliamentary and municipal: A treatise*. Longmans, Green, Reader, and Dyer, 1873.

J. A. Hartigan. *Clustering algorithms*. John Wiley & Sons, Inc. New York, NY, USA, 1975.

T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, pages 97–109, 1970.

S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1994.

V. Heidrich-Meisner and C. Igel. Evolution strategies for direct policy search. *Parallel Problem Solving from Nature–PPSN X*, pages 428–437, 2008.

T. K. Ho, J. J. Hull, and S. N. Srihari. Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(1):66–75, 1994.

W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

J. H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, 1992. ISBN 0262581116.

R. A. Howard. *Dynamic programming and Markov processes*. MIT Press, 1960.

T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

T. Jaakkola, S. P. Singh, and M. I. Jordan. Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 345–352. MIT Press, Cambridge MA, 1995.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

J. L. W. V. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Journal Acta Mathematica*, 30(1):175–193, 1906.

F. Jiang, H. Berry, and M. Schoenauer. Supervised and evolutionary learning of echo state networks. *Parallel Problem Solving from Nature–PPSN X*, pages 215–224, 2008.

J. Jiang and M. S. Kamel. Aggregation of reinforcement learning algorithms. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN 2006)*, pages 68–72, 2006.

L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains, 1995. Unpublished report.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

S. Kakade. A natural policy gradient. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14 (NIPS-01)*, pages 1531–1538. MIT Press, 2001.

M. J. Kearns and S. P. Singh. Finite-sample convergence rates for Q-learning and indirect algorithms. In *Neural Information Processing Systems 12*, pages 996–1002. MIT Press, 1999.

M. J. Kearns and S. P. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2):209–232, 2002.

J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings of IEEE international conference on neural networks*, volume 4, pages 1942–1948. Perth, Australia, 1995.

S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5):975–986, 1984.

V. R. Konda and V. Borkar. Actor-critic type learning algorithms for Markov decision processes. *SIAM Journal on Control and Optimization*, 38(1):94–123, 1999.

V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, 2003.

D. Kortenkamp, R. P. Bonasso, and R. Murphy. *Artificial intelligence and mobile robots: case studies of successful robot systems*. MIT Press, 1998.

P. S. Laplace. Mémoire sur les approximations des formules qui sont fonctions de très grands nombres et sur leur application aux probabilités. *Mémoires de l'Académie Royale des Sciences de Paris*, 10:301–347, 1810.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Back-propagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

A. M. Liapunov. Nouvelle forme du théorème sur la limite des probabilités. *Mémoires de l'Académie imp'eriale de Saint-Pétersbourg*, 12:1–24, 1901.

C. S. Lin and H. Kim. CMAC-based adaptive critic self-learning control. *IEEE Transactions on Neural Networks*, 2(5):530–533, 1991.

L.-J. Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, January 1993.

M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 157–163. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

M. L. Littman and C. Szepesvári. A generalized reinforcement-learning model: Convergence and applications. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning (ICML-96)*, pages 310–318, Bari, Italy, 1996. Morgan Kaufmann.

C. Liu, K. Conn, N. Sarkar, and W. Stone. Online affect detection and robot behavior adaptation for intervention of children with autism. *IEEE Transactions on Robotics*, 24(4):883–896, 2008.

F. J. Lobo, C. F. Lima, and Z. Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*. Springer, 2007.

W. S. Lovejoy. A survey of algorithms methods for partially observable Markov decision processes. *Annals of Operations Research*, 28:47–66, 1991.

H. R. Maei and R. S. Sutton. GQ (λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference On Artificial General Intelligence (AGI-10)*, pages 91–96, Lugano, Switserland, 2010. Atlantis Press.

H. R. Maei, C. Szepesvári, S. Bhatnagar, D. Precup, D. Silver, and R. Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. *Advances in Neural Information Processing Systems 22 (NIPS-09)*, 22, 2009.

H. R. Maei, C. Szepesvári, S. Bhatnagar, and R. S. Sutton. Toward off-policy learning control with function approximation. In *Proceedings of the 27th Annual International Conference on Machine Learning (ICML-10)*, New York, NY, USA, 2010. ACM.

S. Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159–196, 1996.

S. Mannor and J. N. Tsitsiklis. The sample complexity of exploration in the multi-armed bandit problem. *Journal of Machine Learning Research*, 5: 623–648, 2004.

S. Mannor, D. Simester, P. Sun, and J. N. Tsitsiklis. Bias and variance approximation in value function estimates. *Management Science*, 53(2):308–322, 2007.

B. P. Marron. One person, one vote, several elections: Instant runoff voting and the constitution. *Vermont Law Review*, 28:343, 2003.

J. McCarthy and P. Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.

J. Michels, A. Saxena, and A. Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, page 600. ACM, 2005.

D. Michie and R. Chambers. BOXES: An experiment in adaptive control. *Machine intelligence*, 2(2):137–152, 1968.

T. M. Mitchell. *Machine learning*. McGraw Hill, New York, US, 1996.

G. E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.

D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11: 241–276, 1999.

J. J. Murray, C. J. Cox, G. G. Lendaris, and R. Saeks. Adaptive dynamic programming. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 32(2):140–153, 2002. ISSN 1094-6977.

M. B. Naghibi-Sistani, M. R. Akbarzadeh-Tootoonchi, M. H. J.-D. Bayaz, and H. Rajabi-Mashhadi. Application of Q-learning with temperature variation for bidding strategies in market based power systems. *Energy Conversion and Management*, 47(11/12):1529–1538, 2006. ISSN 0196-8904.

K. S. Narendra and M. A. L. Thathachar. Learning automata - a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323–334, 1974.

K. S. Narendra and M. A. L. Thathachar. *Learning automata: an introduction*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1989.

T. W. Nattkemper and A. Wismüller. Tumor feature visualization with unsupervised learning. *Medical Image Analysis*, 9(4):344–351, 2005.

A. Nedić and D. P. Bertsekas. Least squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems*, 13 (1-2):79–110, 2003.

J. Neyman and E. S. Pearson. On the use and interpretation of certain test criteria for purposes of statistical inference part i. *Biometrika*, 20(1):175–240, 1928.

R. G. Niemi. The problem of strategic behavior under approval voting. *American Political Science Review*, pages 952–958, 1984.

M. J. Osborne and A. Rubinstein. *A course in game theory*. The MIT press, 1994.

S. Pandey, D. Chakrabarti, and D. Agarwal. Multi-armed bandit problems with dependent arms. In *Proceedings of the 24th international conference on Machine learning*, pages 721–728. ACM, 2007.

N. Papadatos. Maximum variance of order statistics. *Annals of the Institute of Statistical Mathematics*, 47:185–193, 1995.

R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1088–1094. Morgan Kaufmann, 1995.

J. Pazis and M. G. Lagoudakis. Binary action search for learning continuous-action control policies. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 793–800. ACM, 2009.

B. A. Pearlmutter and G. E. Hinton. G-maximization: An unsupervised learning procedure for discovering regularities. In J. S. Denker, editor, *Neural Networks for Computing: American Institute of Physics Conference Proceedings 151*, volume 2, pages 333–338, 1986.

J. Peng. *Efficient dynamic programming-based learning for control*. PhD thesis, Northeastern University, 1993.

J. Peng and R. J. Williams. Incremental multi-step Q-learning. *Machine Learning*, 22:283–290, 1996.

J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008a.

J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–697, 2008b.

J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *IEEE-RAS international conference on humanoid robots (Humanoids2003)*. IEEE Press, 2003.

E. L. Porteus. Some bounds for discounted sequential decision processes. *Management Science*, 18(1):7–11, 1971.

W. B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley-Blackwell, 2007.

D. Precup and R. S. Sutton. Off-policy temporal-difference learning with function approximation. In *Machine learning: proceedings of the eighteenth International Conference (ICML 2001)*, pages 417–424, Williams College, Williamstown, MA, USA, 2001. Morgan Kaufmann.

D. Precup, R. S. Sutton, and S. P. Singh. Eligibility traces for off-policy policy evaluation. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, pages 766–773, Stanford University, Stanford, CA, USA, 2000. Morgan Kaufmann.

D. Prokhorov and D. Wunsch. Adaptive critic designs. *IEEE Transactions on Neural Networks*, 8(5):997–1007, 2002.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Machine Learning, Proceedings of the Fifteenth International Conference (ICML '98)*, pages 463–471. Morgan Kaufmann, San Francisco, CA, 1998.

C. R. Rao and S. J. Poti. On locally most powerful tests when alternatives are one sided. *Sankhyā: The Indian Journal of Statistics*, pages 439–439, 1946.

P. Ray. Independence of irrelevant alternatives. *Econometrica: Journal of the Econometric Society*, 41(5):987–991, 1973.

I. Rechenberg. *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog, 1971.

R. Richie, C. Kleppner, and T. Bouricius. Instant runoffs: A cheaper, fairer, better way to conduct elections. *National Civic Review*, 89(1):95–110, 2000.

M. Riedmiller. Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*, pages 317–328. Springer, 2005.

M. Riedmiller, J. Peters, and S. Schaal. Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 254–261. IEEE, 2007. ISBN 1424407060.

B. D. Ripley. *Pattern recognition and neural networks*. Cambridge University Press, 2008.

H. Robbins. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc*, 58(5):527–535, 1952.

H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

A. Ross. Computing bounds on the expected maximum of correlated normal variables. *Methodology and Computing in Applied Probability*, 12:111–138, 2010.

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

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist sytems. Technical Report CUED/F-INFENG-TR 166, Cambridge University, UK, 1994.

S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2009.

D. G. Saari and J. Newenhizen. The problem of indeterminacy in approval, multiple, and truncated voting systems. *Public Choice*, 59(2):101–120, 1988.

J. C. Santamaria, R. S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive behavior*, 6(2):163–217, 1997. ISSN 1059-7123.

C. Schaffer. A conservation law for generalization performance. In *Proceedings of the Eleventh International Conference on Machine Learning (ECML 1994)*, pages 259–265, 1994.

L. Schomaker. Using stroke-or character-based self-organizing maps in the recognition of on-line, connected cursive script. *Pattern Recognition*, 26(3): 443–450, 1993. ISSN 0031-3203.

A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Machine Learning: Proceedings of the Tenth International Conference*, pages 298–305. Morgan Kaufmann, Amherst, MA, 1993.

H. P. Schwefel. *Numerische Optimierung von Computer-Modellen*. Birkhäuser, Basel, 1977. Volume 26 of Interdisciplinary Systems Research.

F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010. ISSN 0893-6080.

L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39(10):1095–1100, 1953.

A. A. Sherstov and P. Stone. Function approximation via tile coding: Automating parameter choice. In J.-D. Zucker and L. Saitta, editors, *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK*, volume 3607 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 2005. ISBN 3-540-27872-9.

S. P. Singh. The efficient learning of multiple task sequences. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*, pages 251–258, San Mateo, CA, 1992. Morgan Kaufmann.

S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

S. P. Singh and R. C. Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16, 1994.

S. P. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

R. D. Smallwood and E. J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, pages 1071–1088, 1973.

W. D. Smart and L. P. Kaelbling. Effective reinforcement learning for mobile robots. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation (ICRA 2002)*, pages 3404–3410, Washington, DC, USA, 2002.

J. E. Smith and R. L. Winkler. The optimizer's curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, 52(3):311–322, 2006.

J. H. Smith. Aggregation of preferences with variable electorate. *Econometrica: Journal of the Econometric Society*, pages 1027–1041, 1973.

W. D. Smith. Range voting. Technical report, NEC Research Institute, 2000.

E. J. Sondik. *The Optimal Control of Partially Observable Markov Decision Processes*. PhD thesis, Standford, California, 1971.

P. D. Straffin Jr. *Topics in the Theory of Voting*. Birkhaeuser Verlag, 1980.

A. L. Strehl and M. L. Littman. A theoretical analysis of model-based interval estimation. In *Proceedings of the 22nd international conference on Machine learning*, pages 857–864. ACM, 2005.

A. L. Strehl, L. Li, E. Wiewiora, J. Langford, and M. L. Littman. PAC model-free reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 881–888. ACM, 2006.

A. L. Strehl, L. Li, and M. L. Littman. Reinforcement learning in finite MDPs: PAC analysis. *Journal of Machine Learning Research*, 10:2413–2444, 2009.

R. Sun and T. Peterson. Multi-agent reinforcement learning: weighting and partitioning. *Neural Networks*, 12(4-5):727–753, 1999.

Y. Sun, D. Wierstra, T. Schaul, and J. Schmidhuber. Efficient natural evolution strategies. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO-09)*, pages 539–546. ACM, 2009.

R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Dept. of Comp. and Inf. Sci., 1984.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

R. S. Sutton. Integrated architectures for learning, planning and reacting based on dynamic programming. In *Machine Learning: Proceedings of the Seventh International Workshop*, 1990.

R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1045. MIT Press, Cambridge MA, 1996.

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

R. S. Sutton and S. P. Singh. On step-size and bias in temporal-difference learning. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pages 91–96. Citeseer, 1994.

R. S. Sutton, D. Precup, and S. P. Singh. Intra-Option Learning about Temporally Abstract Actions. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML-98*, pages 556–564. Morgan Kaufmann Publishers Inc., 1998.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems 13 (NIPS-00)*, 12:1057–1063, 2000.

R. S. Sutton, C. Szepesvári, and H. R. Maei. A convergent O(n) algorithm for off-policy temporal-difference learning with linear function approximation. *Advances in Neural Information Processing Systems 21 (NIPS-08)*, 21: 1609–1616, 2008.

R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML-09)*, pages 993–1000. ACM, 2009.

C. Szepesvári. The asymptotic convergence-rate of Q-learning. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 1064–1070, Cambridge, MA, USA, 1998. MIT Press. ISBN 0-262-10076-2.

C. Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.

C. Szepesvári and M. L. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation*, 11(8):2017–2059, 1999.

C. Szepesvári and W. D. Smart. Interpolation-based Q-learning. In *Proceedings of the twenty-first international conference on Machine learning (ICML-04)*, page 100. ACM, 2004.

D. M. J. Tax, M. Van Breukelen, and R. P. W. Duin. Combining multiple classifiers by averaging or by multiplying? *Pattern recognition*, 33(9):1475–1485, 2000.

M. E. Taylor, S. Whiteson, and P. Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, page 1328. ACM, 2006.

G. Tesauro. Practical issues in temporal difference learning. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 4*, pages 259–266. San Mateo, CA: Morgan Kaufmann, 1992.

G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

G. J. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

R. H. Thaler. Anomalies: The winner's curse. *Journal of Economic Perspectives*, 2(1):191–202, Winter 1988.

C. K. Tham. Reinforcement learning of multiple tasks using a hierarchical CMAC architecture. *Robotics and Autonomous Systems*, 15(4):247–274, 1995.

N. Tideman. The single transferable vote. *Journal of Economic Perspectives*, 9(1):27–38, 1995.

T. N. Tideman. Independence of clones as a criterion for voting rules. *Social Choice and Welfare*, 4(3):185–206, 1987.

C. F. Touzet. Neural reinforcement learning for behaviour synthesis. *Robotics and Autonomous Systems*, 22(3/4):251–281, 1997.

J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16:185–202, 1994.

J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, Cambridge,MA: MIT Laboratory for Information and Decision Systems, 1996.

J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42 (5):674–690, 1997.

A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR-91)*, pages 586–591. IEEE, 2002.

E. Van den Steen. Rational overoptimism (and other biases). *American Economic Review*, 94(4):1141–1151, September 2004.

J. van der Wal. Discounted Markov games: generalized policy iteration method. *Journal of Optimization Theory and Applications*, 25(1):125–138, 1978.

M. Van Erp and L. Schomaker. Variants of the borda count method for combining ranked classifier hypotheses. In *Proceedings of the Seventh International Workshop on Frontiers in Handwriting Recognition*, pages 443–452, 2000.

H. P. van Hasselt. Double Q-Learning. In *Advances in Neural Information Processing Systems*, volume 23. The MIT Press, 2010.

H. P. van Hasselt and M. A. Wiering. Reinforcement learning in continuous action spaces. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-07)*, pages 272–279, 2007a.

H. P. van Hasselt and M. A. Wiering. Convergence of model-based temporal difference learning for control. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-07)*, pages 60–67, 2007b.

H. P. van Hasselt and M. A. Wiering. Using continuous action spaces to solve discrete problems. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN 2009)*, pages 1149–1156, 2009.

J. Van Nunen. A set of successive approximation methods for discounted Markovian decision problems. *Mathematical Methods of Operations Research*, 20(5):203–208, 1976.

H. van Seijen, H. P. van Hasselt, S. Whiteson, and M. A. Wiering. A theoretical and empirical analysis of Expected Sarsa. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 177–184, 2009.

V. N. Vapnik. *The nature of statistical learning theory*. Springer Verlag, 1995.

D. Vrabie, O. Pastravanu, M. Abu-Khalaf, and F. Lewis. Adaptive optimal control for continuous-time linear systems based on policy iteration. *Automatica*, 45(2):477–484, 2009. ISSN 0005-1098.

F. Y. Wang, H. Zhang, and D. Liu. Adaptive dynamic programming: An introduction. *Computational Intelligence Magazine, IEEE*, 4(2):39–47, 2009.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, England, 1989.

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

B. L. Welch. The generalization of 'Student's' problem when several different population variances are involved. *Biometrika*, 34:28–35, 1947. ISSN 0006-3444.

P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

P. J. Werbos. Advanced forecasting methods for global crisis warning and models of intelligence. In *General Systems*, volume XXII, pages 25–38, 1977.

P. J. Werbos. Neural networks for control and system identification. In *Proceedings of IEEE/CDC Tampa, Florida*, 1989a.

P. J. Werbos. Backpropagation and neurocontrol: A review and prospectus. In *IEEE/INNS International Joint Conference on Neural Networks, Washington, D. C.*, volume 1, pages 209–216, 1989b.

P. J. Werbos. Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 2:179–189, 1990.

P. J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 2002. ISSN 0018-9219.

J. Westra, H. P. van Hasselt, F. Dignum, and V. Dignum. Adaptive serious games using agent organizations. In *Agents for Games and Simulations, Trends in Techniques, Concepts and Design, Proceedings of the First International Workshop on Agents for Games and Simulations (AGS-2009)*, pages 206–220, 2009a.

J. Westra, H. P. van Hasselt, V. Dignum, and F. Dignum. On-line adapting games using agent organizations. In *Proceedings of the IEEE Symposium On Computational Intelligence and Games (CIG-08)*, pages 243–250. IEEE, 2009b.

S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917, 2006.

S. Whiteson, M. E. Taylor, and P. Stone. Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, 15(1):33, 2007.

D. Whitley, S. Dominic, R. Das, and C. W. Anderson. Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13(2):259–284, 1993.

B. Widrow and M. E. Hoff. Adaptive switching circuits. *1960 IRE WESCON Convention Record*, 4:96–104, 1960. New York: IRE. Reprinted in Anderson and Rosenfeld [1988].

A. P. Wieland. Evolving neural network controllers for unstable systems. In *International Joint Conference on Neural Networks*, volume 2, pages 667–673, Seattle, 1991. IEEE, New York.

M. A. Wiering. QV(λ)-learning: A new on-policy reinforcement learning algorithm. In D. Leone, editor, *Proceedings of the 7th European Workshop on Reinforcement Learning*, pages 17–18, 2005.

M. A. Wiering and J. H. Schmidhuber. Fast online Q(λ). *Machine Learning*, 33(1):105–116, 1998.

M. A. Wiering and H. P. van Hasselt. Two novel on-policy reinforcement learning algorithms based on TD(λ)-methods. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-07)*, pages 280–287, 2007.

M. A. Wiering and H. P. van Hasselt. Ensemble algorithms in reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38 (4):930–936, 2008.

M. A. Wiering and H. P. van Hasselt. The QV family compared to other reinforcement learning algorithms. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 101–108, 2009.

D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989. ISSN 0899-7667.

D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

D. Wingate, C. Diuk, L. Li, M. Taylor, and J. Frank. Workshop summary: Results of the 2009 reinforcement learning competition. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML-09)*, page 1, New York, NY, USA, 2009. ACM.

D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, The Santa Fe Institute, Santa Fe, N. M., 1995.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

L. E. Zachrisson. Markov games. *Advances in Game Theory*, pages 211–253, 1964.