The State of Mind
Reinforcement Learning with Recurrent Neural Networks

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 29 januari 2004
klokke 14.15 uur

door

Pieter Bram Bakker

geboren te Leidschendam
in 1972

Promotiecommissie

Promotor:

Prof.dr. P.T.W. Hudson	Universiteit Maastricht/Universiteit Leiden

Referent:

Dr. J. Schmidhuber	IDSIA

Overige leden:

Prof.dr. G.A.M. Kempen	Universiteit Leiden
Prof.dr. J.N. Kok	Universiteit Leiden
Prof.dr. E.O. Postma	Universiteit Maastricht
Prof.dr. J.M.J. Murre	Universiteit Maastricht/Universiteit van Amsterdam
Dr. B.J.A. Kröse	Universiteit van Amsterdam
Dr. M.A. Wiering	Universiteit Utrecht
Dr. F. v. d. Velde	Universiteit Leiden
Dr. G. Wolters	Universiteit Leiden

Contents

List of Figures

List of Tables

Preface

In doing the research for this thesis, I have had the privilege to work in a group of people that have allowed me to largely pursue my own ideas and interests, while at the same time providing the resources I needed and providing many opportunities for stimulating discussions of my work and theirs. For this I am very grateful. In this context, a number of colleagues deserve special mention.

First of all, I wish to thank the various colleagues that I shared a room with over the years and that were always the first in sharing the joys and inevitable annoyances of PhD research: Arjan de Boer, with whom I enjoyed many hours of discussions about the good and bad in science, society, and soccer; Antonino Raffone, whose enthusiasm about neural networks was contageous; and Jiska Memelink, who often lent a sympathetic ear when I was complaining about the life of a PhD student.

Several colleagues discussed with me topics vital to my own research. Of all colleagues in Leiden, Fred Keijzer's work was probably most similar in spirit to my own work. The papers he let me read when I was still an undergraduate student were very influential in my choices of topics as a PhD student, and I appreciate greatly our many discussions about adaptive behavior and dynamical systems—even if usually we did not agree. Mark de Kamps taught me some valuable dynamical systems concepts and discussed with me a lot of work on neural networks. Bart Happel's work on combining genetic algorithms and neural networks was very inspiring to me as an undergraduate student, and I thoroughly enjoyed the regrettably few discussions about our work and the grander goals of AI later on.

In addition to discussions related directly to my own work, I appreciated the many scientific and non-scientific conversations I had during lunch and coffee breaks, and occasionally dinner or drinks, with many other colleagues, including my regular lunch partners Nomi Olsthoorn, Guido Band, Ineke Bloem, and Fenna Poletiek.

I had very pleasant and fruitful cooperations with three PhD students at other universities that led to publications, Paul den Dulk, Michiel de Jong, and Fredrik Linåker. The work with Paul and with Michiel did not end up in this thesis in explicit form, but many of the ideas developed in our cooperations are there implicitly, and both Paul and Michiel helped in clarifying many issues for me. The work done with Fredrik ended up in chapter 7, and is to me a good example of how cooperation between people working in different countries and relatively different fields can still be both pleasant and mutually beneficial.

I have had the good fortune to work with two very clever undergraduate students who did research with me, and I am very pleased that both went on to become PhD

students themselves. Gwendid van de Voort van de Kleij cooperated with me on the early work for chapter 4, executing and analyzing the first extensive simulations on internal state in MDPs ("handy state" as we liked to call it). Michel van Dartel cooperated with me on parts of the work described in chapter 5, analyzing the "cognitive map" capabilities of a recurrent neural network trained for navigation. Besides doing very useful work for me, both Gwendid and Michel made great company inside and outside the university.

As in every research group, in the Unit of Cognitive Psychology in Leiden a pivotal role is played by the secretaries. I wish to thank Karin Honsbeek, Marina Bouhuis, Romke Biagioni, and Albertien Olthoff for taking care of countless administrative issues that I couldn't keep track of, keeping me up to date of important news, protecting my spot in a good room, organizing trips and birthday parties, and lifting my spirits by once making me employee of the month.

I wish to thank those people who took the time and effort to read parts of the unfinished thesis and reports on which this thesis is based, and who provided many useful comments: Michiel de Jong, Edwin de Jong, Dagmar van der Neut, Gwendid van de Voort van de Kleij, Michel van Dartel, Fred Keijzer, Ingmar Visser, Viktor Zhumatiy, Jelle Kok, and Matthijs Spaan. I want to single out Edwin de Jong, who read a draft of the complete thesis, made encouraging remarks, and provided very to-the-point feedback both on a detailed level and on a conceptual, overall level.

I would like to thank the members and former members of our AI discussion club "Emergentsia", Edwin de Jong, Michiel de Jong, Dick de Ridder, David Tax, Rens Kortmann, Paul den Dulk, Martijn Brinkers, Hendrik-Jan Hoeve, Jelle Kok, Sjaak Verbeek, Matthijs Spaan, Sander Bohté, Martijn van Otterlo, and Marco Wiering, for the many interesting discussions about diverse topics in AI. I learned much from these meetings, thanks to the diversity of discussed papers and the diversity of expertise in the club; whatever general AI context I missed in my own research group I found in Emergentsia.

Edwin de Jong, Michiel de Jong, and Paul den Dulk were among the earliest members of Emergentsia, but more importantly, they have been good friends for many years, and their ideas about AI and cognitive science have influenced me greatly from when I was in the first years of my undergraduate studies. In fact, the enthusiasm invoked by our conversations played an important role in my decision to apply for a PhD student position in the first place, so I owe special thanks to them.

My life as a PhD student in Leiden revolved not about work alone, fortunately, and I would like to thank all my friends, who provided plenty of fun distractions from my research, who understood when to inquire carefully about my research and when not to ask anything, who pointed out the relativity of it all by making fun of my thesis ("scriptie") and my topic (pole balancing in particular is hilarious, apparently), but who also expressed serious interest, faith, and support during many rounds of tea, coffee, and still stronger beverages.

I wish to express immense thanks to my parents for their unconditional love, belief in me, and support through thick and thin.

Finally, the person I want to thank most of all is Dagmar, for her unfailing love, help, patience, and understanding, especially during the last and most difficult period of finishing this thesis.

Chapter 1

Introduction

1.1 Understanding and constructing intelligence

The general, long-term goals behind the research described in this thesis are to contribute to our understanding of biological intelligence and to the construction of artificial intelligence. These two goals are purposively stated together, because I believe that they are intimately connected. Understanding the complexities of biological intelligence requires the insights of an advanced science of artificial intelligence, and vice versa, artificial intelligence can benefit a great deal from insights into how nature solved the problems of building intelligent systems. That is not to say that true artificial intelligence must necessarily mimic biological intelligence precisely, or that biological intelligence will necessarily turn out to function exactly as the best artificially intelligent systems. However, I do believe that general principles can be found, general principles of intelligence, and in uncovering those principles, either from studies on biological intelligence or from studies on artificial intelligence, we actually learn about both biological and artificial intelligence.

The methodology of the work in this thesis reflects this idea. First and foremost, algorithms and simulation experiments are presented which aim to contribute to artificial intelligence. However, the algorithms and architectures that are used, reinforcement learning and neural networks, were chosen not only because of their promise for artificial intelligence per se, but also because of their relationship with biological learning and biological nervous systems. It is difficult to predict the insights into "general principles of intelligence" which may, in the long run, be yielded by this methodology, and in this thesis no specific "hypotheses" about those principles are formulated and tested in a rigorous way. However, the proposed architectures and algorithms could be viewed as implicit hypotheses about what might constitute certain principles of intelligence. Examples are the use of value function approximation for learning and control throughout the thesis, the use of a single system, a recurrent neural network, to construct a useful state signal by generalizing over perceptual information where possible and supplementing it with short-term memory where necessary, and the use of hierarchical control for difficult tasks. Occasionally, general principles of intelligence are more explicitly debated, for example in the discussion about the trade-off between

perception and internal state in chapter 4, and in the discussion about cognitive maps in chapter 5.

1.2 The main focus of this thesis

The work in this thesis is part of the general framework of adaptive behavior research. More specifically, it is part of the machine learning paradigm of reinforcement learning. The main focus of the thesis is to build and understand mechanisms and algorithms for reinforcement learning tasks where perception of the environment may or must be supplemented with internal state (short-term memory). Much of adaptive behavior research and reinforcement learning is concerned with "reactive" tasks, where good behavior can be accomplished using a direct mapping from perception of the environment to actions. This thesis considers more difficult but also more realistic tasks where that is not practical or possible, i.e. where immediate perception of the environment does not yield useful or sufficient information, and the system needs to learn to use internal state in combination with perception for good behavior. Such tasks can be called "representation-hungry" tasks (Clark, 1997), and the reinforcement learning equivalent is known as Partially Observable Markov Decision Processes (POMDPs). They pose important challenges for reinforcement learning, as well as for adaptive behavior research in general.

The specific learning tasks considered in different chapters are all tasks of this nature. They are basically variations of two types of tasks, navigation and pole balancing. However, it should be noted that this thesis is not about navigation and pole balancing as applications per se. They are only used to illustrate how policies can be learned, using the methods described in this thesis, in different types of partially observable domains and in domains with varying degrees of complexity. Besides the distinction between navigation and pole balancing, another dimension in which the investigated learning tasks can be distinguished is the following. One class consists of tasks which are conceptually simple and which look fairly "fabricated". They are designed such that we can easily determine optimal policies, easily vary specific difficulties of the task, and easily determine when and why the proposed methods succeed and fail. In some cases, these are tasks where one could fairly easily write a program that would solve the task; but that is beside the point. These are tasks that investigate particular difficulties for the learning algorithms, difficulties that are also important in more complex tasks, for which it would be much harder to program the solution by hand. The second category of tasks is more complex and realistic. They are used to demonstrate the power of the proposed methods and to show that the methods work in somewhat more realistic settings than the fabricated tasks of the first category, settings in which it is already more difficult to handcode the solution. The disadvantage of this class of tasks is that it is often harder to determine their optimal policies, harder to understand the specific difficulties they pose for the algorithms, and harder to analyze successful and failing agents.

1.3 The structure of this thesis

The chapters in this thesis can largely be read individually, with occasional references to previous chapters. At the same time, the chapters have a more or less logical ordering and I believe it is valuable to read them in their intended order. Each chapter builds on the previous chapters, in the sense that the previous chapters serve as a theoretical introduction, or in the sense that a chapter deals with problems or extends algorithms that were described in previous chapters. References to literature related to specific research questions can be found in the individual chapters; a general overview of the literature is presented in the next two chapters.

The next chapter, chapter 2, provides an introduction to adaptive behavior research, which provides the general research context or philosophy behind the work of this thesis. It explains, in much more detail than this introduction, how adaptive behavior research contributes to our understanding of biological and artificial intelligence. In this sense, it sets the stage for the next chapters, which zoom in to more specific, technical research questions.

Chapter 3 describes the specific paradigm used in this thesis, reinforcement learning. It focuses especially on problems and methods related to those investigated in this thesis. This helps in understanding the position that the methods used in this thesis take within the field of reinforcement learning, and in understanding the relevance of the specific research questions investigated in this thesis.

Chapter 4 is the first chapter in which a specific technical research question is addressed. It describes simulation experiments where the reinforcement learning algorithm known as Q-learning is combined with simple recurrent neural networks (Elman networks) to solve reactive tasks (MDPs). Such tasks could in principle be solved by learning a direct mapping from perception to actions, for instance using tables or feedforward neural networks. This chapter shows that even in this case, the internal state provided by the recurrent activations in the network can be useful, and in fact, a trade-off between the use of perception and internal state can be identified.

Chapter 5 considers the case where internal states are not just useful but necessary: POMDPs. It investigates the validity of the approach of using recurrent neural networks approximating value functions of a reinforcement learning algorithm for both continuous and discrete partially observable tasks. It focuses, in particular, on analyzing the resulting outward behavior of the agents and internal functioning of the networks. In this chapter, instead of Q-learning a reinforcement learning algorithm is used which can be viewed as an extension or transformation of Q-learning: Advantage(λ) learning.

Chapter 6 extends the work presented in chapters 4 and 5 by using a more sophisticated recurrent neural network architecture. Chapters 4 and 5 use Elman networks, chapter 6 uses Long Short-Term Memory (LSTM) networks. The LSTM network allows the reinforcement learning agent to solve significantly more difficult POMDPs, which have complex and long-term temporal dependencies between events that must be remembered and actions that depend on them. In addition, instead of straightforward undirected exploration as in the previous chapters, this chapter proposes a more sophisticated adaptive, directed exploration technique.

Chapter 7 extends the system described in chapter 6 by combining it with an unsupervised learning component, making the methods of the thesis more applicable to real robot applications. The overall system can be understood as a hierarchical control system: the unsupervised learning component preprocesses the robot's raw sensory inputs and provides a higher-level representation of the robot's environment to the reinforcement learning component, which learns to select low-level behaviors for execution.

Chapter 8, finally, presents the general conclusions. It summarizes the technical as well as conceptual contributions of this thesis, and it suggests some interesting directions for future research.

1.4 Notation and terminology

I have tried to make the terminology and mathematical notation in the various chapters maximally consistent with each other and with existing work on reinforcement learning and neural networks. A few exceptions are inevitable, but I believe they do not have to lead to confusion. For example, in this thesis parameters of function approximators adapted by reinforcement learning algorithms are sometimes referred to as w_i and sometimes as w_{im}. The latter usage is prevalent when we specifically consider neural networks as the function approximator, in which case such a parameter is always associated with a connection between a unit m and a unit i. Furthermore, time is always indicated by t, but sometimes as in $s(t)$ and sometimes as in s_t. Which one is used depends on the focus and readability of the corresponding equations, but it is always consistent at least within any one chapter. When we speak about the "state", without further qualifications, this always refers to the state of the environment. The state within an agent or neural network is referred to as "internal state". Similarly, the symbol s is, consistent with the reinforcement learning literature, reserved for indicating the state of the environment. For this reason, the CEC activation of an LSTM network's memory cell is not indicated by $s_{c_j^v}$, as is common in papers on LSTM, but by $z_{c_j^v}$.

Chapter 2

The adaptive behavior approach to cognitive science

Summary

This chapter discusses the field of adaptive behavior, a relatively new approach to cognitive science and artificial intelligence, that provides the general framework for the research described in this thesis. The goal of adaptive behavior research is to learn about intelligent behavior by constructing agents exhibiting that behavior, but it is different from traditional artificial intelligence. It is argued that rather than providing mere implementations of the abstract theories of mainstream cognitive science and artificial intelligence, the adaptive behavior approach yields many new insights, as well as indications that some of mainstream cognitive science's assumptions and theoretical ideas may be in need of revision. Specific examples of adaptive behavior research are presented to support these claims.

2.1 Introduction

Cognitive science is the scientific discipline that studies the basic mechanisms underlying human and animal intelligent behavior. Its ultimate goal is to understand how perception comes about, how memory works, how muscles are controlled, how language is produced and perceived, how emotions work, etc.; in short, how all the faculties function that make up the mind. Derived from that goal are the goals of understanding how those faculties may fail and how they behave in different circumstances, and applying the insights of cognitive science to clinical, educational, or industrial settings.

In recent years, forceful arguments have been put forward to the effect that cognitive science will and must become more intimately related to the neural sciences (Crick, 1988; Churchland, 1986). Among other developments, technological advances have made it possible to "look" inside the active brain, using techniques with acronym names such as PET, fMRI, ERP, and MEG. The new insights into brain mechanisms

that are likely to emerge from those developments will have to be taken into account by cognitive science.

The work described in this thesis is part of another rapidly evolving field with great relevance for cognitive science. It is the field of *adaptive behavior*, alternatively referred to as the field of autonomous agents, animats, behavior-based cognitive science, bottom-up cognitive science, or synthetic psychology. This chapter provides a general introduction to the key ideas and philosophy of the field of adaptive behavior and in this way sets the stage for the more technical chapters to follow.

The field of adaptive behavior takes an artificial intelligence approach to cognitive science (although it is very different from traditional artificial intelligence), in that systems are constructed that are capable of intelligent behavior. The idea is that by constructing an artificial system one can learn something about the biological system. In the process of construction one may encounter the same problems that the biological system encounters, and hypotheses about how the biological system overcomes these problems can be put to the test.

One might expect that this approach amounts to straightforward "implementation" of the abstract theories of mainstream cognitive science as concrete models. At best, this would lead to arbitration between competing abstract theories, and the filling in of some of the details left open by those abstract theories. In contrast, this chapter argues that the adaptive behavior approach yields insights that are far more revolutionary. The systems that are constructed and that work best are, in many cases, very different from any of the systems described in mainstream cognitive science literature. Where attempts have been made to implement existing abstract theories directly, insurmountable problems have often appeared—suggesting falsification of those theories rather than a mere filling in of the details.

To back up this argument it is necessary to describe the current standard view of human and animal information processing in cognitive science, and to present results from the adaptive behavior approach that are different from or even incompatible with that view. Section 2 contains a description of current mainstream cognitive science, together with some representative examples. Section 3 describes the adaptive behavior approach by contrasting it with mainstream cognitive science. Section 4 presents a number of examples of artificial systems or "agents" that are very different in architecture and functioning from what was or would be expected given the standard view. In that way they serve to back up the argument that some of the assumptions and ideas in mainstream cognitive science may be in need of revision, and that a closer relationship between cognitive science and adaptive behavior research is needed. Section 5, finally, contains a general discussion of the ideas presented in this chapter.

2.2 Mainstream cognitive science

2.2.1 Theoretical foundations

After the behaviorist era with its exclusive focus on behavior (Skinner, 1938, 1991), the cognitive revolution halfway through the 20th century brought renewed attention for the internal mechanisms underlying behavior (Chomsky, 1959; Tolman, 1948).

Theoretical ideas regarding those mechanisms were heavily influenced by another development of that time. The theory of computation showed that a very large class of mathematical functions could be performed by machines that mechanistically follow stored procedures, computers. In fact, it was made plausible that no physical machine could reliably perform functions beyond this class. Importantly, all computable functions can be performed by one and the same machine, the "hardware", only by changing the stored procedures, the "software" (e.g. Pylyshyn, 1984). Actual computer applications displayed capabilities such as arithmetic, pattern recognition, and game playing, which were previously thought to require human intelligence, and which in many cases outperformed human capabilities.

Ever since then, the internal mechanisms of the mind have been theorized to amount to a kind of computer. Combining several ideas from computation theory, Newell and Simon (1976) stated the *symbol system hypothesis*: for a system to be capable of human-level intelligence, it is both necessary and sufficient for it to constitute a symbol system, a computer. Just like a computer, the brain is viewed as a single piece of hardware capable of a wide variety of tasks. One idea of computation theory has been particularly important in that respect. That idea is that a computer can in principle be realized (implemented) in many different types of hardware, whether they are mechanical switches, electronic circuits, or even beer cans. Thus, an implementation in neural tissue is also possible. The end result of the computation is not dependent on the specific implementation, in the same sense in which Microsoft Word works equally well (or equally badly) on Windows computers and Macs. In that sense, the hardware is irrelevant. This has led to a widespread doctrine called *functionalism*, which states that for this reason cognitive science does not need to be concerned with the physical implementation, the brain, but only needs to focus on the functional level, the "software running on" the brain (e.g. Pylyshyn, 1984).

2.2.2 Functions and functional modules

The theoretical foundations described above should be considered in combination with a number of practical principles of scientific methodology to understand current cognitive science. These practical principles amount to ideas about how research on the mechanisms underlying behavior should proceed. It is clear that the entire human brain or mind is too complex and multi-faceted to oversee and assess for single researchers, or to describe all aspects and details of its functioning in terms of a single simple model. For this reason, and influenced by the computer analogy, the system is decomposed into different *functions*: perception (itself sorted by sensory modality), memory, language, attention, motor control, reasoning, etc. In practice, each function has its own, more or less separate community of researchers and a distinct body of literature. Figure 2.1 presents a fairly characteristic general overview of the human information processing system as it appears in many textbooks (e.g. Ashcraft, 1998).

Within each function, a similar decomposition into constituent functional components or modules is assumed. Each component is dedicated to some subfunction. Information is exchanged between components through communication lines although, in close analogy with computer systems, these communication lines are often assumed to have a "limited bandwidth", i.e. have a limited capacity. Exchanged information

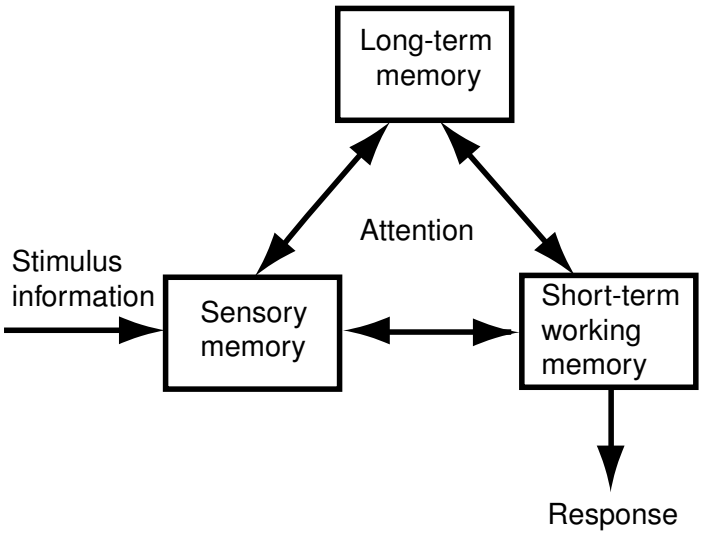

Figure 2.1: An overview of the human information processing system. Adapted from Ashcraft (1998). Similar overviews appear in many textbooks.

consists of the results computed by one module which are the prerequisite for another module. Many models assume strictly serial processing; an array of modules activated one after the other, each one using the final results of the previous module as input. Other models, recent ones in particular, allow parallel processing. Multiple modules may be active at the same time and possibly interact, and the results are integrated into a final, response module. Components are usually represented visually as boxes and communication lines as arrows, yielding what are known as box-arrow models. Figures 2.2 and 2.3 show two well-known models representing this approach, one model of language production (Levelt, Roelofs, & Meyer, 1999) and one model of working memory (Baddeley, 1990).

The idea of decomposition into functions and functional modules is not just based on practical considerations and the computer analogy. Even though functionalism dictates that the hardware of the brain is, in an important sense, irrelevant for the functioning of the mind, cognitive scientists have been quick to embrace findings from the neural sciences that seem to support the idea of decomposition into functions. It is known from studies where animal brains were purposely lesioned, as well as from brain imaging and cell recording studies, that different anatomical regions in brains are involved in different aspects of animal and human functioning. Furthermore, neuropsychological studies show that people with accidental brain lesions, e.g. caused by car accidents, strokes, or bullet wounds, often display fairly specific defects rather than an overall deterioration of functioning. For instance, a person may lose the ability to produce language but still comprehend language, or a person may be able to visually recognize all objects except faces.

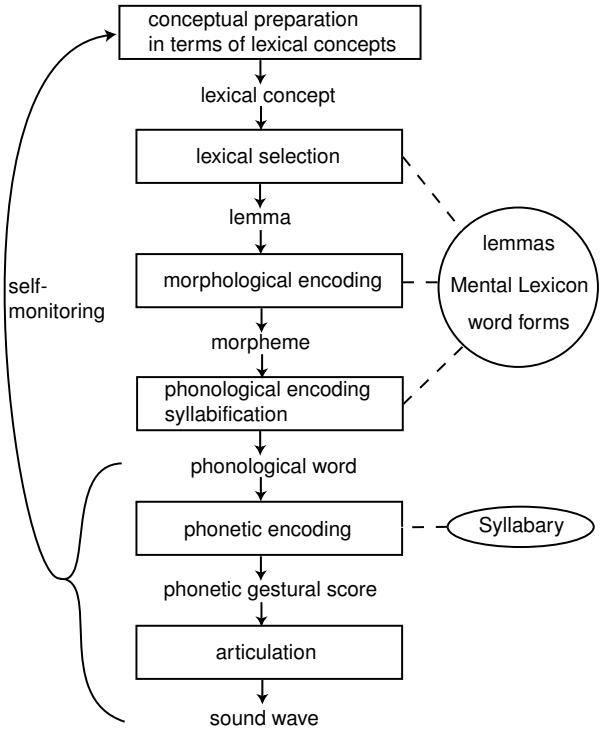

Figure 2.2: A model of language production. Adapted from Levelt, Roelofs, & Meyer (1999).

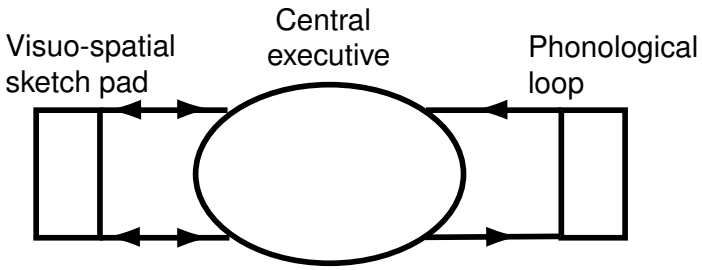

Figure 2.3: A model of working memory. Adapted from Baddeley (1990).

Many models in mainstream cognitive science assume as one of the functional modules some kind of central workspace or central processor where information from different sources comes together, where selections are being made, where information is temporarily stored and manipulated, and where decisions are made regarding actions. In figure 2.1, for instance, there is a short-term working memory, which, guided by attention, receives information from long-term memory and from sensory memory. In working memory this information is processed, and the appropriate response is given as output. Figure 2.3 depicts a central executive, which has, as slave systems, temporary stores for auditory information and visuo-spatial information, and which uses that information for its reasoning and decision processes. Figure 2.2 has a central executive in disguise, in the form of the self-monitoring process. Central processors and workspaces are based on the way computers function. There are hardly any data from the neural sciences which indicate that such central processors or workspaces actually exist in animal and human brains (Dennett, 1994, 1991), but in this case such objections are brushed aside by referring to functionalism.

2.2.3 Experiments and effects

Other practical considerations that have been important in establishing current cognitive science concern not the theoretical models themselves but the experiments that are being done to develop and support the models. Experimental data are the foundation for cognitive science's theories, but they have the problem of providing only *indirect* information on the architecture and functioning of internal mechanisms. This is true even for brain imaging data, because these may, for instance, give rough indications as to which brain area receives more blood than others, but not say in what way that brain area is involved exactly. However, the problem is even more acute for the kind of data on outward behavior that experimental cognitive science typically works with: reaction time, strength, frequency, and accuracy of responses. Interestingly, these measures have in large part remained the same since the dawn of experimental psychology in the late 19th century. To cope with the indirectness of those data, experiments are usually set up such that a manipulation of a single controlled variable results in a changed response from the subjects, a so-called *effect*. The type and direction of the effect may then be used to derive conclusions about the underlying mechanisms that are involved and affected by the experimental manipulation. The subtractive procedure or the method of additive factors, developed by Donders (1862) and still widely used, is a particular instance of this idea. An experimental manipulation results in a change of responses, in this case longer reaction times, and this is interpreted as indicating an additional stage of processing or an additional involved functional component.

However, drawing straightforward conclusions from such effects can be difficult. Even if we have a large, clear-cut effect, its implications for the underlying mechanism are often unclear. Consider a case from language production research based on picture-word interference experiments (e.g. Glaser & Düngelhoff, 1984; Levelt et al., 1999). These experiments are based on the Stroop task, an experimental paradigm that is more than 60 years old (Stroop, 1935). In picture-word interference experiments, the subject must name a picture (or classify it or respond in another way). Within the boundaries of the picture, a distracting word appears before, during, or after the

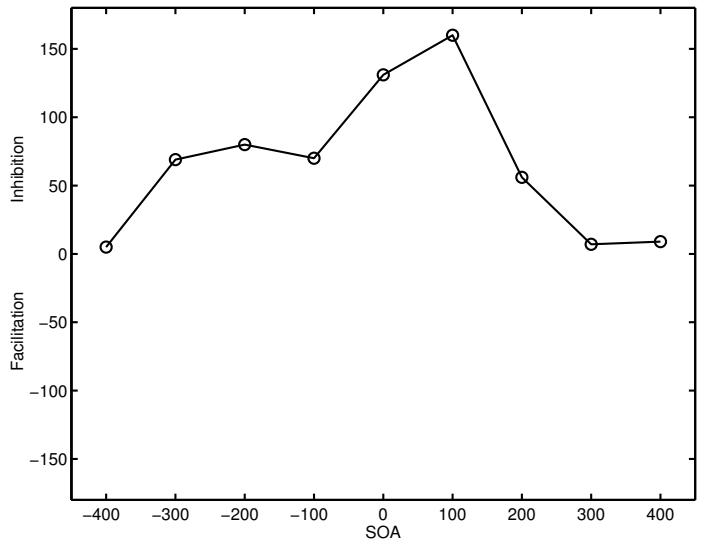

Figure 2.4: Semantic inhibition in the picture-word interference task where subjects have to name the picture, as a function of Stimulus Onset Asynchrony (SOA). Adapted from Glaser & Düngelhoff (1984).

appearance of the picture itself. A reliable finding is that if the so-called Stimulus Onset Asynchrony (SOA), the time between the presentation of the picture and the word, is roughly between -100 milliseconds and +100 milliseconds, and the distractor word is semantically related to the picture (such as the word "dog" is related to a picture of a cat), naming the picture takes significantly longer than in other conditions. This finding is displayed in figure 2.4.

What does this *semantic inhibition* effect mean for models of language production? In the interpretation of additive factors, it could mean that an additional stage of processing or functional module is involved. Alternatively, it could mean that semantically related words are stored closely together in memory, and "activation" of words "spreads out" to their neighborhood in a kind of blurring process, making it harder to distinguish semantically related words and pick out the correct one. Yet another interpretation is that the distractor word is processed faster than the picture and prepares the response component for saying the wrong word, which subsequently needs to be suppressed before the right word can be said.

The standard way to decide between the alternatives is to run new experiments. However, effects may disappear or even be reversed by slight changes to the experimental setup or by alternative manipulations. In the case of picture-word interference tasks, the effect is indeed reversed and becomes a *facilitation* effect when the pictures must be *categorized* rather than named (Levelt et al., 1999). In many cases research then tends to "zoom in" on details regarding those changes and manipulations, mapping out exactly when effects appear and how large they may become. In doing that,

researchers are following the seemingly reasonable principle that one must first come to grips with the subtleties of the effect before one can move on. In addition, in new experiments a completely new effect may appear, much to the delight of its discoverers. The details of this new effect are then explored in more detail, etcetera.

In effect (no pun intended), the end result is that much of cognitive science is more concerned with effects by themselves than with the implications of effects for theoretical models. Measurements and the effects derived from them take on a life of their own, with models being postponed to a later time "when we have more data". The argument that "there is not yet enough data" is similarly used to justify the usual state of affairs that if a model is proposed, it is not specified beyond the very abstract level of a small number of boxes and arrows, as described above and illustrated by figures 2.1–2.3.

2.2.4 Mathematical and computational models

In those rare cases where a cognitive science model is further specified and a mathematical or computational model is implemented, an observation can be made that illustrates the strong focus of mainstream cognitive science on raw experimental data and effects. Such models typically replicate the raw experimental data and effects very precisely, but they do not aim to replicate the general behavior that the research initially set out to investigate.

For example, the computational model of language production proposed by Levelt et al. (1999), which is an implementation of the box-arrow model depicted in figure 2.2, replicates the semantic inhibition effect very well, along with a number of other experimental effects, but it says little about language production in general: how the picture is "transformed" into processes dedicated to language, what those processes look like and how they are operated upon, how the complex structure of language and the very large number of words are dealt with, and how processes concerned with language are in turn transformed into muscular movements corresponding to saying a word. In other words, one cannot present an actual picture of a cat to the computational model and obtain an auditory response "cat"—let alone have the language production model produce language as it is normally produced, in the form of comprehensible sentences. Reaction times and errors are produced, rather than actual language. Again this is justified by referring to the intuitively reasonable arguments that there is not enough data and that one must stay close to the data that one has got, before one can go on and "generalize to the unknown". However, this results in the peculiar situation that data that were initially gathered to say something about the mechanisms underlying a general capability of behavior, become the primary focus of the model at the expense of those general mechanisms. Such models explain the errors in and speed of certain behavior, without explaining the behavior itself.

The (implicit) idea behind this is that successive models of raw data and effects will encompass more and more data and, in the long run, converge to models which will indeed be able to produce actual behavior. After all, if we keep doing experiments, more and more data will become available, and if all conceivable data are eventually taken into account, this should include data on actual behavior. However, we have seen that more data often amounts to more details regarding the existence, disappearance

and reversal of an effect under different experimental variations, and not necessarily to more data on actual behavior in natural circumstances. In practice, after many years of careful experimentation and corresponding theorizing, models have rarely, if ever, converged to models aiming more and more to replicate actual behavior in natural circumstances.

2.3 The adaptive behavior approach

Adaptive behavior research takes a very different approach to the study of the mechanisms underlying behavior. This approach is conveniently and effectively described by contrasting it with mainstream cognitive science on a number of issues.

2.3.1 General capabilities, as opposed to experimental effects

The previous section ended with a discussion of the kind of data that current models in cognitive science focus on, and ideas about how successive models will encompass more data. In general, this concerns the question of deciding where to start: which kind of data should be modeled first? Mainstream cognitive science usually opts for the kind of exact and quantitative but, as we saw, very indirect data that can be and have been obtained in carefully controlled experiments.

The adaptive behavior approach can be understood as choosing another kind of "data" to focus on first. It attempts to replicate general capabilities of behavior that humans or animals exhibit. A system, or *agent*, is constructed that is capable of, for instance, locomotion behavior, or navigation behavior, or cooperation behavior, etc. These capabilities are not data in the regular sense of that which is gathered in controlled experiments, but they are data in the sense that they are non-trivial phenomena exhibited by humans and animals and they should be accounted for by a model. The idea is that a good model of the mechanisms underlying behavior should first and foremost account for what those mechanisms are *for*, what makes them interesting in the first place: the generation of successful behavior.

Viewed from this perspective, it is clear that the adaptive behavior approach is not merely a branch of engineering. It is a branch of cognitive science, in that it attempts to explain known facts about the behavior of organisms, facts concerning the existence, complexity, and limitations of that behavior. The difference between mainstream cognitive science and the adaptive behavior approach is not that the first explains actual data on organisms and the latter does not; the difference is in the type of data that is being explained.

Obviously, the type of data described as "general capabilities" cannot be measured as rigorously and quantitatively as the typical data gathered in experimental cognitive science. In some cases, we may only be able to say that there is a qualitative fit between the constructed agent's behavior and an organism's behavior: the agent is capable of some behavior, but the degree to which it matches the capability of a particular organism is not entirely clear. In other cases, it may be possible to determine this fit more quantitatively, by taking measures of effectiveness and efficiency. However, in

many cases an agent's behavior is not even intended to model the behavior of a specific species, but rather a type of behavior exhibited by many species.

Mainstream cognitive science often stays close to quantitative data gathered in neatly controlled experiments at the expense of the actual behavior itself. The adaptive behavior approach does the opposite: sacrificing some quantitative measures in favor of "qualitative data" that are claimed to have at least as high a priority.

2.3.2 Basic behavior, as opposed to high-level behavior

Building a system capable of behavior is difficult. At the moment, complex behavior is out of reach. For this reason, the adaptive behavior approach simplifies by focusing on very basic behavior first (Keijzer, 2001). This can be contrasted with most of experimental cognitive science which can be characterized as simplifying by taking simple behavioral measures of mostly high-level cognition such as natural language, decision making, complex perception and attention, and intentional processes. It can similarly be contrasted to "traditional" artificial intelligence, which has focused on mimicking those higher-level cognitive processes. Traditional artificial intelligence has met with some success, but mainly when the domain to which the high-level cognitive process applies is very limited, such as chess or medical diagnosis. The idea of going back to basic behavior first, which adaptive behavior research adopts, is based in part on the brittleness of traditional artificial intelligence systems in real-world domains.

The adaptive behavior approach has the disadvantage that, initially, many of the most interesting types of behavior, especially the ones exhibited uniquely by humans and not by other animals, are not dealt with. On the other hand, this may not be such a bad idea given the history of research in biology, for instance. Taking an example from Cliff (1991), the very successful field of genetics has worked its way up from the relatively simple genomes of the fruit fly up to more and more complex genomes, with the human genome coming into the picture only recently. Similarly, adaptive behavior research starts with the very basic behaviors exhibited by virtually all animals and slowly works its way up: locomotion, orientation, approach to desirable stimuli and movement away from danger, collision detection and collision avoidance, simple navigation, prey following and fleeing, cooperation, communication, etc.

This initial focus on basic behavior, as opposed to a focus on high-level cognition, does not imply a return to behaviorism or a denial of the existence and importance of high-level cognition. It is mainly a pragmatic choice, based on perceived limitations of our understanding of the mechanisms underlying even very basic behavior. In sharp contrast with behaviorism, adaptive behavior research is very much concerned with the internal mechanisms behind behavior, and it does not view behavior as simple stimulus-response relationships. In the long run, the adaptive behavior approach hopes to tackle more high-level cognitive behavior. In fact, there are already some efforts in this direction, especially with regard to planning (Nolfi & Floreano, 1998; Mataric, 1997; Sutton, 1991), communication using higher-level concepts (de Jong, 1999), and even language (Steels, 1997).

2.3.3 Learning by constructing, as opposed to learning by measuring

By constructing an agent capable of a certain type of behavior, one can learn something about how biological systems, organisms, accomplish that behavior (Braitenberg, 1984). At the very least, one will learn about the *problems* that are involved in accomplishing that behavior, which are the same problems that the biological system must overcome. We can think of the evolved mechanisms underlying behavior as a *solution* to those problems. It seems reasonable that understanding that solution requires a sufficient understanding of the problems.

Interestingly, during construction one's intuitions about what will be and what will not be severe problems are often falsified. The history of the field of artificial intelligence is very illuminating in this regard. On the one hand, problems may turn out to be much more severe than was envisioned. In research on navigating robots it was initially thought that the process of transducing sensory information into a world model, as well as the process of transducing planned actions into motor commands was fairly trivial, and that all the hard work is done in sorting out a plan given the world model (Nilsson, 1984; Moravec, 1982). This turned out to be a gross underestimation of the difficulties of those "transduction" processes, and research on navigating robots stalled for many years as a result (Brooks, 1991a).

On the other hand, problems that seem to be very serious may turn out to be pretty easy. It was previously thought that transforming English verbs from the present tense into the past tense requires a complicated system of rules, containing the rules and how they are applied, as well as the exceptions to which they do not apply. However, it was shown that this task can be performed relatively successfully and even learned as a straightforward nonlinear input-output mapping, using a basic feedforward neural network (Rumelhart & McClelland, 1986a). This system learned the past tenses of both regular and irregular verbs, and generalized reasonably well to both regular and irregular verbs it had never seen before.

In general, the history of artificial intelligence shows that construction of an intelligent system is by no means a trivial enterprise. This suggests that it is not true that for any abstract theory on how to achieve some behavior, it will be easy or even possible to construct an implementation of that theory that works. The conclusion that the adaptive behavior approach draws from this observation is that the test of implementation is a much more serious one than is usually assumed by mainstream cognitive science. Implementation reveals genuine problems associated with accomplishing a particular type of behavior; and it reveals weaknesses (if any) of abstract theories in overcoming those problems.

Conversely, properties of a successful artificial system may suggest ideas, and even specific hypotheses, about how the biological system does the job. The engineered solution can be compared and contrasted with the biological solution. Having something to compare with may help in making sense of the data on biological mechanisms that are available from psychology and the neurosciences, data that are there in large quantities but that are often hard to interpret.

2.3.4 Mild functionalism, as opposed to extreme functionalism

The degree to which constructed solutions, agents, tell us anything about nature's solutions, organisms, is a matter of some debate. The extreme form of functionalism that is usually adhered to in mainstream cognitive science tells us that the same function can be accomplished in a virtually unlimited number of ways. For this reason, investigating one possible implementation of a function in an engineered agent is not very relevant. The way the engineered agent performs the function may be completely different from the way the organism performs it, and what we are interested in, in the end, is only the latter. How much does an airplane really teach us about how birds fly? Perhaps surprisingly, the answer to that question is: quite a lot. One example is that even though airplanes do not flap their wings and the wings are (fairly) rigid, the principle by which their wings accomplish their task is very similar to birds. Airplane wings and bird wings have similar, though not exactly the same, curvatures that cause lower air pressure above the wing than below during horizontal movement, creating lift. This understanding of bird wings almost completely depends on the development of airplanes and the corresponding understanding of aerodynamics.

In general, there are not as many possible ways to accomplish a particular capability effectively as seems to be implied by extreme functionalism. The original idea from computation theory on which functionalism is based only says that the end result of a computation can be accomplished in multiple ways. Firstly, it says nothing about how long it will take. The archetypical computer, the Turing Machine, would be impractically slow for nearly all functions if it were implemented. In other words, different implementations are good for different functions.

Secondly, computation theory does not claim that *just any* idea about how to achieve a capability will work. In the enormous space of possibilities of different systems, or "design space", only a few regions amount to systems that are actually capable of accomplishing anything interesting. One can reduce the number of regions even more and zoom in to the region of interest by using self-imposed constraints on the "building blocks" of the artificial system. Sure enough, a wheeled robot does not tell us very much about legged locomotion. However, if the robot has legs and its artificial brain consists of artificial neurons, it becomes a different matter altogether. It then becomes more likely that one lands in more or less the same region of possibilities as the biological system, and the constructed solution has accordingly more similarities with the biological solution.

Thus, extreme functionalism's suggestion that the particular implementation of an abstract function is irrelevant is rejected. The implementation is neither irrelevant nor trivial; in contrast, in order to understand a system's functioning one must understand the implementation and in principle be able to construct one.

There seems to be a paradox here. If a system's functioning is so closely tied to its implementation, how can we ever expect to learn something about an organism's functioning, with its biological implementation, by studying an artificial implementation? To solve this apparent paradox, it is necessary to specify the functionalism dimension further. At one end of the spectrum, there is extreme functionalism, which says that a system's functioning has nothing to do with the system's implementation. In the extreme, this would imply that any random stack of bricks could be intelligent.

At the other end, there is extreme "implementationism", which says that a system's functioning is inseparably linked to all details of its implementation. Extreme implementationism implies that each and every protein used in the cell is critical in achieving a neuron's function, and an artificial neuron that leaves out one such protein will fail (this is sometimes called "carbon chauvinism"). The argument that was made here really argues for *mild functionalism*. Not *every* detail in an implementation is crucial for a system's functioning. It is possible and necessary to abstract away from many of the details (such as a particular protein) when constructing an artificial system and make meaningful comparisons with the biological system. In other words, within a single region of design space, there are still many systems whose details differ to some extent, but whose functioning is not affected significantly by those details. Of course, it is not known *a priori* which details are essential and which are not; this has to be found out.

In a way, the basic idea behind functionalism was always good: it is possible and in fact essential to discard irrelevant implementation details so as to arrive at abstract, scientific understanding of the functioning of complex systems (this is what I mean by mild functionalism here). However, in mainstream cognitive science this idea gradually transformed into a dogma that says that one should not consider concrete implementations because they are irrelevant for scientific understanding (radical functionalism); and this is wrong.

The adaptive behavior approach follows mild functionalism and attempts to construct successful implementations as a route to understanding intelligence. In the process, the constructed agent's capabilities, its limitations, and the problems it overcomes can be understood; and the agent can be compared to biological implementations. In most cases, adaptive behavior research takes inspiration from biology and attempts to use in its implementations the same type of building blocks as nature does, in order to facilitate meaningful comparisons. Sometimes it even becomes a matter of testing specific hypotheses about how the biological system works (e.g. Webb, 1994; Beer, 1990). As an extra advantage of the adaptive behavior approach, known biological features can be added to and removed from the agent at will, and the effects on the resultant behavior can be observed. In this way, it may become clear whether that particular feature is essential for the organism's functioning or not. These suggestions can subsequently be put to the test by the neuroscience and psychology communities (see Webb, 1994; Beer, 1990 for examples). These are important ways in which adaptive behavior research feeds back to neuroscience and psychology and fruitful interactions may take place.

2.3.5 Detailed models, as opposed to abstract models

In accordance with mild functionalism, the adaptive behavior approach proposes implemented agents that actually demonstrate some capability of behavior, and which are therefore specified in great detail. This is in contrast with the very abstract models typically proposed by mainstream cognitive science. Those abstract models usually consist of a few boxes and arrows (see figures 2.1–2.3) and they contain hardly any details, sometimes to the point where one may wonder how much information they contain at all. A box labeled "short-term memory" contains only the information

that humans are capable of remembering information presented to them a little while before; it says nothing about the underlying mechanisms.

Mainstream cognitive science often takes the position that details are bad, because the "principle of parsimony" says that models should be as simple as possible. The adaptive behavior approach takes the position that details are good or at least unavoidable, because actual behavior cannot be accomplished without them, neither in artificial systems nor in biological systems. As argued above in the context of "learning by constructing", it is very hard to predict in advance which details are arbitrary and which are crucial. Only by implementing them will this become clear. In any case, a constructed agent that is successful at its behavior can be seen as an "existence proof" that proposed mechanisms deemed necessary for functioning actually do the job, and this requires the details to be filled in. Such an existence proof can never be given by an abstract box-arrow model.

A related objection is that details make it hard to see the bigger picture: the general principles that govern the mechanisms behind the behavior. If there are too many details, one may not be able to see "the wood for the trees". First of all, if a detail turns out to be crucial in the implementation phase, it is apparently not part of the trees, but of the wood: without it, the bigger picture would not be complete, and without implementing it, the bigger picture could not have been obtained at all. On the other hand, one should not drastically go the other way toward extreme implementationism, reasoning that all details may be important, and concluding that therefore one should always start by modeling individual molecules, or atoms, or elementary particles. One should constrain oneself to details for which it is reasonable to suppose that they may have relevance for the system's functioning; the details by virtue of which the system may be doing what it is doing. Admittedly, finding this right level of detail is an art in itself. It is as easy to become too detailed as it is to become too abstract.

The agents that are constructed within the adaptive behavior approach are either actual robots or computer simulations. In both cases, the agent is investigated as a complete system of brain and body interacting with an environment, to make the behavioral task as realistic as possible. Both computer simulations and robots have advantages and disadvantages. An important advantage of computer simulations is that they are easy to work with and easy to modify. In addition, certain things are practically impossible to realize in robots but are possible to simulate in the computer, such as evolution, large populations of agents, muscles, a variety of environments, etc. On the other hand, it is very hard to simulate the full complexity of the real world. There is a danger of oversimplifying the problems that the agent is faced with. This is not a problem if one uses real robots. A real robot needs to confront the real world and show it is capable of successful behavior there—one may argue that only then there is a genuine existence proof that proposed mechanisms underlying behavior actually work. On the other hand, research on robots is very difficult and can get stuck in technical problems which are not interesting from an AI/cognitive science point of view. Usually, the types of behavior exhibited by robots are simpler than the ones in simulation, because it is so hard to achieve even the simple ones.

2.3.6 Perception to action loops, as opposed to functional modules

Mainstream cognitive science decomposes the animal and human information processing system into separate functions, and each function in turn into functional components. Each function is studied in isolation, with little attention for interactions with other functions or with the environment. The adaptive behavior approach uses another kind of decomposition, one that isolates all mechanisms that are involved in the specific, usually simple behavior of interest. Since the goal is to achieve complete behavior, all aspects from perception to action which are critical in achieving that particular type of behavior must be dealt with. This also includes interaction of the brain with the body and with the environment. Actions typically change the state of the body and the world and what is perceived next in an immediate feedback-like way, and the next actions should depend on these new percepts. This temporal aspect is central to virtually any behavioral task, and it must be taken into account if successful behavior is to be generated. Mainstream cognitive science does not usually acknowledge the importance of this fact, and it treats body and environment as "passive", independent receivers of actions and providers of sensations.

Dealing with all aspects from perception to action sounds like adding up all problems involved in different functions. If an individual function studied in isolation is so complex as to warrant a large, separate field of research, how can we expect to deal with all those functions at once? The trick is to study and solve only those problems of, for instance, perception or memory or motor control that are necessary to accomplish the single behavior of interest. Agreed, full-blown human perception is very complex and cannot be simply built into an agent. That is why the focus is on basic behavior and on the corresponding aspects of perception, memory, and motor control that by themselves are relatively simple when compared to the full complexity of human perception, memory, and motor control. However, much can be and has been learned from how these simple aspects of a task constrain each other, how they interact with each other, and how interaction with the environment constrains the whole system (Brooks, 1989, 1991a; Mataric, 1991).

Often, adaptive behavior researchers take a "minimalist" approach to constructing perception to action loops capable of a certain behavior (Wilson, 1991). That is, they attempt to accomplish as much intelligence as possible with as simple an architecture as possible. An architecture is only extended when absolutely necessary. The idea is that in this way one can better understand what part of an overall architecture the capability can be attributed to and one can better understand the power as well as limitations of different architectures.

As we shall see in the next section as well as in the rest of the thesis, one of the most important insights resulting from adaptive behavior research is that successful agents sometimes resist a clear-cut decomposition into functions and functional modules. Memory may be distributed across the whole agent rather than located in a separate component (Rumelhart & McClelland, 1986b; Dennett, 1994). Perception and action may be intricately linked to the point where they are no longer usefully thought of as two separate components (Brooks, 1989, 1991a; Beer, 1990; Beer & Gallagher, 1992). Agents may behave "as if" they pay selective attention to certain sensory information,

without having an explicit, separate mechanism or component for attention (Werner, 1994; McCallum, 1996). For these agents, the intuitively natural decomposition into functions and functional components is not the most fruitful one. This in turn suggests that it is not necessarily the most fruitful one when one thinks about biological systems.

What about those findings from the neural sciences, described above, that seem to support the idea of decomposition into functions and functional modules? First of all, it should be noted that many of those findings do not directly implicate such clear-cut functional isolation, but rather show something much weaker: there is some level of specialization, *not all* brain areas are directly involved in a particular task, but only parts of the brain. The same is usually true of constructed agents: there is some level of specialization, some parts of the artificial brain are involved in some tasks, and other parts in other tasks.

Within biological brain regions that are involved, it is usually unclear how to further assign subtasks to brain areas. However, what is suggestive is that there are typically many connections back and forth to brain areas; a finding that argues against strict functional isolation. Furthermore, one should be careful with deriving conclusions from failing systems. If a radio starts to make a howling sound if one particular transistor is broken, this does not mean that this transistor is the "howl inhibitor" (Arbib, 1989). In general, the phenomenon that only a specific capability fails if one of the physical building block is broken, is a characteristic of many systems, and not only systems with clear-cut, isolated functional modules.

Nevertheless, there are good reasons for assuming some level of modularity (e.g. Simon, 1962), in the sense that one element in a system does not and should not always depend heavily on (all) other elements; and as described above, the brain certainly seems to exhibit a certain level of modularity. However, the extent and the nature of this modularity should be found out rather than assumed and a priori defined.

2.3.7 Decentralized control, as opposed to centralized control

Mainstream cognitive science usually proposes as one of the functional modules a central workspace where information is temporarily stored for processing and organization by a central controller. There is a danger of attributing all capabilities that are not yet understood to this central controller, which is not specified in more detail. Sometimes this central controller can rightly be called a "homunculus", a little man in the head, which takes over all the hard work that the overall system is supposed to do. Such a model does not explain the capabilities of the system, but only "pushes back" the problems deeper into the system, into an unspecified functional component named "central controller" (Dennett, 1991).

By its very nature, the adaptive behavior approach cannot resort to this strategy because it forces itself to replicate the capabilities. If a central controller were to be proposed, it would have to be implemented for behavior to be accomplished. In the process, its mechanisms have to be made explicit, and its weaknesses, if any, are revealed.

Almost all of traditional artificial intelligence has used central controllers and workspaces in its systems. This has met with very limited success, and it is therefore an example of a specific cognitive science idea about underlying mechanisms that

seems to be falsified whenever put to the test of implementation. Among the problems is the creation of a serious bottleneck if everything has to pass through the central workspace, slowing down performance tremendously. There is also the problem of information access (Dennett, 1994; Lenat & Guha, 1990). One can have a long-term memory storing huge amounts of information, but how does one get the relevant piece of information into working memory in time? Furthermore, for systems that interact with the world, it has proven to be very hard to maintain and update the central model of the world that the central processor is operating upon. Using sensory information to decide what has to be changed in the central world model, as well as predicting what will change in the world model if some action is executed, is notoriously difficult (e.g. Brooks, 1991a; Moravec, 1982; Nilsson, 1984; Dennett, 1991; Krotkov & Simmons, 1996; see Hutter, 2003 for the theoretically optimal but computationally intractable approach to this and related problems).

For these reasons, the adaptive behavior approach typically attempts to accomplish behavior without using a central controller or workspace. Control is decentralized, distributed among local controllers operating in parallel. Each local controller performs a simple task, such as moving a single leg when its own sensors say so, or activating another local unit when its sensors detect a specific feature in the world. Through the interaction of these local controllers between themselves and with the environment, the behavior as a whole "emerges", without a need for a central guiding or monitoring mechanism. This is sometimes called *self-organization*, because there is no central system actively organizing the behavior.

2.3.8 Distributed, continuous representation, as opposed to symbolic representation

Based on the computer analogy, the content of the central workspace (as well as long-term memory) is usually theorized to amount to so-called symbolic representations. These are language-like structures consisting of arrays of symbols, manipulated by logical operations. Symbols are the atoms of knowledge that "stand for" something in the outside or inside world, such as "Mary" or "love". An important, powerful property of symbolic representations is their combinatorial structure. Symbols can be combined into large symbol structures in many different ways, representing many different meanings. Certain operations on these representations may depend on the combination of the symbols, rather than the meaning of the individual symbols themselves. This is called structure-sensitive processing, and it allows a single type of operation to generalize to many different contexts.

Just as there are no indications that there is a central workspace in the brain, it is not obvious at all where and how the symbolic representations are encoded. That is not to say that there are no symbolic representations in the brain. However, the structure of and processes in the brain suggest other types of encoding that are used next to, or perhaps even instead of symbolic encodings. These other types of encodings are investigated in depth in the field of artificial neural networks or "connectionism" (see Rumelhart & McClelland, 1986b). Artificial neural networks are simplified models of biological neurons and biological neuron interactions. They are inherently based on decentralized control. Furthermore, no clear distinction can be made between the

controlling part and the information used by the controller, which is very different from standard computers and models in mainstream cognitive science. As it turns out, the type of continuous-valued, distributed internal states encoded by artificial neural networks, and therefore—it is hypothesized—by biological nervous systems, affords a type of processing well suited for many parts of intelligent behavior.

Symbolic representations are best suited for logical, all-or-nothing types of reasoning and applications of strict rules. Much of intelligent behavior is handled better and perceived more fruitfully as recognition and classification of patterns, completion of partial information, association between related pieces of information, and decisions based on incomplete information and on the satisfaction of multiple, "soft" constraints. Those types of tasks are done more naturally and effectively in neural networks than in symbol systems. This is particularly acute for a system interacting with the environment, which is an important focus of adaptive behavior research. As described above, the transduction process from sensory information to a central world model, which is supposed to consist of symbolic representations, is very hard. The type of continuous, distributed internal states of neural networks lends itself much better to connections and interactions with sensory and motor apparatus than discrete, symbolic representations. As for structure-sensitive processing, it was shown that this is not limited to symbolic representations but can also be done in neural networks (e.g. Chalmers, 1990; Elman, 1990).

For these reasons, a lot of adaptive behavior research uses artificial neural networks, or some other type of system based on distributed internal states. However, in contrast with the modeling approach in cognitive science known as "connectionism", no absolute commitment is made to distributed internal states. If it turns out that for some types of behavior (e.g. high-level cognition) symbolic representations are necessary or very useful, the adaptive behavior approach, with its focus on making systems work, will use them. There are more differences with connectionism, so it is certainly a mistake simply to equate the two approaches. In a way, connectionism is situated in between mainstream cognitive science and adaptive behavior research. Like the adaptive behavior approach, it emphasizes implementations and decentralized control. However, unlike the adaptive behavior approach, connectionist models are often intended as straightforward implementations of the functional modules of mainstream cognitive science theories, and they are often used to fit typical experimental cognitive science data directly on reaction times and errors. The adaptive behavior approach emphasizes much more than connectionism the importance of building complete agents interacting with realistic environments, exhibiting general capabilities of behavior.

2.3.9 Bottom-up engineering, as opposed to top-down engineering

The decomposition into functions and functional modules that mainstream cognitive science assumes reflects—and is probably inspired by—standard engineering principles. In fact, it is basically how traditional artificial intelligence constructs an artificially intelligent system.

First, it is determined what the system is supposed to do, what its overall task is. Next, this task is divided into subtasks which are handled by dedicated components.

Each component's subtask is relatively independent and well-defined, such that the component can be individually built and tested. To avoid the notorious problem of unwanted side effects and complications caused by interactions between components, each component's functioning is isolated as much as possible from other components. Finally, all components are put together. This can be called top-down engineering, because one starts with the abstract idea of the overall task, working one's way down to more and more concrete subtasks and finally physical realization of the components and combination into a complete system.

However, this is not the way nature constructs systems. Nature does not start with an abstract idea about what the final system should do. It does not neatly figure out subtasks and assign these to functionally isolated subsystems. It does not care about whether or not any clear-cut decomposition into subfunctions is possible at all, or whether the end result is easy to comprehend for scientists; and it does not build and test the subsystems individually before they are recombined. In contrast, nature blindly tries out many kinds of systems without any foresight on what the system should do or how it should do it. It builds on systems that were successful before, varies them randomly and selects the lucky ones that happened to work one way or another. It selects a system as a whole and does not develop subsystems in isolation, opportunistically allowing side effects and complex interactions between subsystems as well as multiple functions within a subsystem if they happen to be beneficial.

To contrast this with top-down engineering, it may be called bottom-up engineering (e.g. Dennett, 1994): starting with building blocks from previous generations, those building blocks are varied and more or less randomly combined into a new system, allowing strong interdependencies and interactions between them, without a preconceived and neatly worked out plan on the overall design; but the system is selected on the basis of success in its environment, such that unsuccessful systems (most systems, in fact) are filtered out and we are left with successful ones. Only with hindsight, then, can one say that a successful system is "designed to perform a particular task".

It seems plausible that top-down engineering often leads to different types of systems than bottom-up engineering. Thinking again in terms of the metaphorical space of possibilities of systems, design space, top-down engineering is constrained to certain regions of design space; in particular, regions where systems are easily decomposable into functional modules (these may be very good systems: airplanes are an example). Bottom-up engineering does not have those constraints. Of course, it has other constraints, such as the constraint that a design is always heavily based on a previous design. Because of these differences in constraints, bottom-up engineering may end up in very different regions of design space. Those regions may yield systems from the easily imaginable to the bizarre—the only criterion used by bottom-up engineering is success of the system.

In order to learn more about the systems designed by nature and to have access to the same regions in design space that nature uses, a lot of adaptive behavior research attempts to mimic nature's bottom-up engineering processes. Agents are developed over many generations using a simulated evolution process, employing so-called evolutionary algorithms; or they are developed using learning, nature's way of developing an organism during its lifetime. As it turns out, in many cases the artificial systems developed in this way are indeed very different from what was expected given

the standard top-down view in traditional artificial intelligence and cognitive science (e.g. Beer, 1995; Beer & Gallagher, 1992; Nolfi & Floreano, 1998). This challenges the (usually implicit) assumption in cognitive science that the top-down approach is the most logical and fruitful one, when it comes to understanding and reconstructing biological intelligence.

2.3.10 A posteriori analysis, as opposed to a priori analysis

The overall emphasis on constructing working systems, the fact that bottom-up engineering does not work from an abstract theory, and the explicit rejection of extreme functionalism, all seem to suggest that the adaptive behavior approach is not very interested in abstract theories. The main concern seems to be with building an agent that is specified in detail and that is successful at its particular task, and not so much with finding more generalized, high-level theories of behavior and of the mechanisms behind behavior. This is not, however, necessarily true.

First of all, it *is* true that the adaptive behavior approach has a somewhat different attitude towards abstract theories. Adaptive behavior research looks at organisms from an engineering perspective. This leads to a general view of cognitive science as an enterprise that is more like reverse engineering than like physics. A "reverse engineer" attempts to understand complex machinery made by someone else, with the goal of being able to build a similar device himself (Dennett, 1991). He does not come up with a single "theory of video cassette recorders" or a single "theory of cars", in the same sense as there is a theory of elementary particles. What he is looking for is insights in the workings of the machine. This requires descriptions of many parts and their interactions, and a number of general principles; but not a single theory. In the same way, it may be an idle hope to find a relatively simple "theory of behavior" for complex, bottom-up engineered systems such as organisms.

Having said that, the goal is still to uncover those general principles of the mechanisms underlying behavior, and taken together, the general principles can be said to constitute the abstract theory. It is important to note that constructing detailed implementations does not preclude an adaptive behavior researcher from thinking about the abstract theory, both in the process of constructing and in the analysis of the completed agent. In the end, abstract theories (general principles) are what the whole enterprise is about; not single, detailed agents, built for one specific environment. The adaptive behavior researcher is, in fact, in one of the *best* positions to understand the general principles, because he has constructed a complete agent himself and knows, like no one else, about the problems that have been overcome and how the solution works.

The doctrine called mild functionalism does not say that one should not be concerned with the abstract theory; it says that one should build and look at implementations if one wants to find the abstract theory. Compared to cognitive science based on extreme functionalism, the order of doing things can be said to be reversed. Cognitive science based on extreme functionalism starts out, *a priori*, with an abstract theory, based on an analysis of the demands of the task and on preconceived ideas on how to deal with the demands, and expects that an implementation can easily be found which realizes this theory in a physical machine. Adaptive behavior research, in contrast,

focuses first and foremost on building a successful machine and only then on analyzing it, *a posteriori*, so as to arrive at the abstract theory. If one employs bottom-up engineering to construct the agent, this is the only possible methodology.

The idea is that it does not make sense to establish the abstract theory until one has extensively investigated the feasibility of different ideas on how to accomplish a certain capability. In other words, implementation is part of the process of theorizing right from the start. Once there are successful implementations, it becomes possible to say which ideas worked and which failed and, with hindsight, it may become easier to see *why* certain ideas worked and others failed. This then yields the more generalized, abstract theories.

As one example, implementations of multilayer feedforward neural networks in detailed computer simulations have yielded the general theoretical insight that a lot of complex rule-like behavior can be accomplished with, and understood as a *nonlinear mapping of vectors* (e.g. Rumelhart & McClelland, 1986b). This is the abstract theory behind multilayer feedforward networks, but it was not and probably could not have been conceived a priori by functionalist cognitive science. The whole concept of "nonlinear mappings of vectors" emerged from and depended on investigations of detailed implementations of neural networks.

An example that is more typical of adaptive behavior research is Beer's (1995) analysis of an evolved locomotion controller, described in more detail in the next section. The high-level abstract theory describing the evolved controller is stated in terms of dynamical systems, and again it is very different from anything that was or even could have been conceived a priori.

2.4 Examples of adaptive behavior research

In this section a number of examples of adaptive behavior research are presented that are intended to illustrate the issues described in the previous section. At the same time, they are examples of some of the theoretical insights that have been gained using the adaptive behavior approach. A number of these insights directly oppose ideas in mainstream cognitive science. This suggests that those classical concepts should not be taken for granted, but may be in need of revision.

2.4.1 Locomotion

Locomotion, in particular legged locomotion, was one of the first types of behavior investigated using the adaptive behavior approach. In part, this resulted from dissatisfaction about traditional artificial intelligence efforts on locomotion. Those efforts had resulted in large robots using a central controller which carefully planned and executed each single step or change of position before anything else could happen, yielding very slow and unnatural locomotion (e.g. Krotkov & Simmons, 1996).

Taking inspiration from biology, researchers, most notably Brooks (1989, 1991b, 1991a), decided to use simple decentralized controllers in small, insect-like legged robots. Each leg is controlled in a local, reflexive way, aided by individual timers: if the leg is down and forward, swing backward; if it is down and backward, lift the

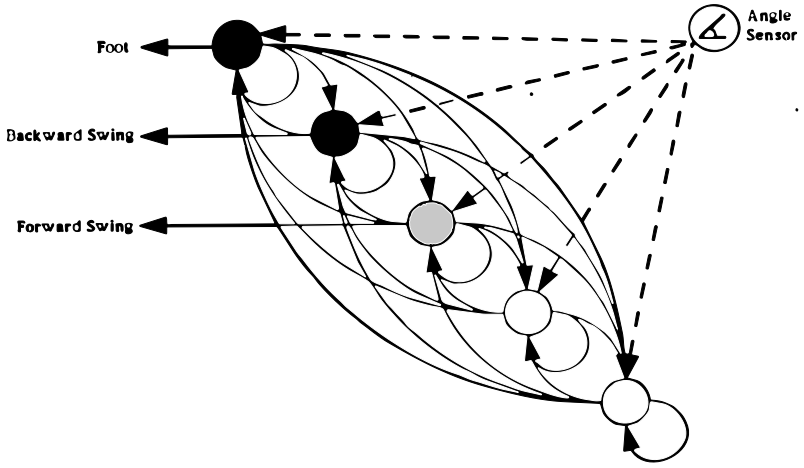

Figure 2.5: Part of the neural locomotion controller connected to a single leg. From Beer & Gallagher (1992).

leg and swing forward, etc. Coordinating these six moving legs such that successful locomotion is accomplished seems like a complex problem. In contrast, this can be achieved by simply letting the robot move about in the world and exploiting the sensors on the legs, using simple inhibition between the legs. If a leg is swinging forward, the other legs that stand on the ground are told to swing backward a little, etc. When put into action, the robot quickly settles into an efficient, lifelike gate of locomotion and is even able to negotiate somewhat rough terrain. It can easily be supplemented with similar reflex-like mechanisms that allow it to detect and deal with collisions. This work was an important "step" because it showed how complex coordinated movements can arise from parallel, distributed control, without a central coordinating mechanism and without using explicit central representations of the task and the environment. Thus, it is one of the first successful examples of exploiting self-organization to achieve complex behavior.

Following up on this work, Beer and co-workers (Beer, 1990; Beer, Chiel, Quinn, Espenschied, & Larsson, 1992) used a neural network as locomotion controller for their simulated insects and real robots. This artificial neural network was a simplified model of the nervous system of the cockroach. Previously, locomotion of this type of animals was thought to be controlled by a central locomotion system carrying out a fixed motor program and sending commands to the legs at precisely timed moments (see Simmons & Young, 1999): a typical model within mainstream cognitive science. In contrast, the neural network model uses highly distributed control and, again, inhibition between legs, but this time only local inhibition between neighboring legs.

The cockroach is known to have different gaits, which it uses at different speeds. To explain this, classical models have to assume that the central locomotion system contains different motor programs, one for each gait. In the neural network model, however, these different gaits arise spontaneously when the speed is varied—another

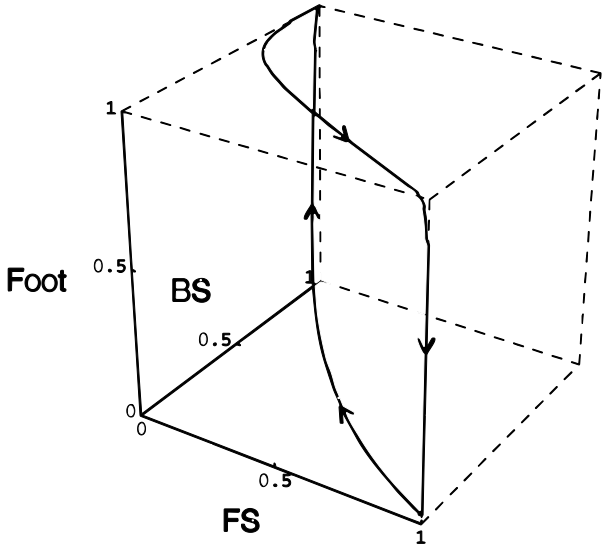

Figure 2.6: Phase space plot of the limit cycle of the leg controller depicted in figure 5. The output of the Foot, Backward Swing (BS), and Forward Swing (FS) motor neurons are plotted. From Beer (1995).

instance of self-organization. If one is to understand how this works, one cannot fruitfully think in terms of the classical concepts of motor programs, functional components, and the like, but one has to view the neural network in interaction with body and environment as a single dynamical system in which different periodic attractors are stable at different speeds.

Later work by Beer and colleagues (Beer, 1995; Beer & Gallagher, 1992) included the development of neural network locomotion controllers using evolutionary algorithms, thus mimicking nature's bottom-up engineering style. This provided additional support for the idea that, at least for organisms exhibiting this type of behavior, dynamical systems notions may provide better explanations than cognitive science's classical notions. A part of the overall neural network that is connected to a single leg is shown in figure 2.5 (Beer & Gallagher, 1992). It cannot be meaningfully decomposed into different functional modules, nor is it functionally isolated from other parts of the network; and there is no central control. Rather, each neuron continuously affects and is affected by the other neurons as well as the body and the environment, and the overall capability emerges from the interactions. To illustrate the difference in the type of explanations of the mechanisms underlying behavior, figure 2.6 depicts a so-called phase plot of the limit cycle of this single leg's local controller interacting with the environment (Beer, 1995). What is important here is that this new, dynamical systems type of abstract theory (and the diagram illustrating the theory) is very different from mainstream cognitive science theories, and this insight depended on the implementation of agents.

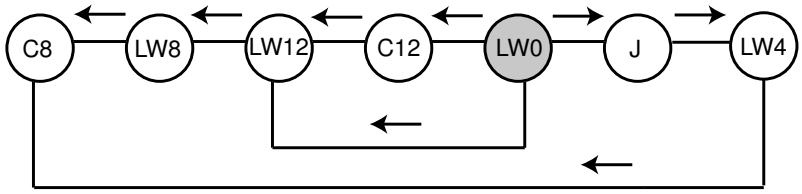

Figure 2.7: Cognitive map learned by a robot in a cluttered office environment. LW8 means Left Wall heading south, for instance. Arrows denote the spreading of activation from the goal (gray node). Adapted from Mataric (1991).

2.4.2 Navigation

Locomotion can be used by an organism just to wander around at random, until food is encountered. Locomotion can be employed much more efficiently, however, if the animal knows where it is and where to go to achieve its goals. This is called navigation, and all but the simplest animals use it. The animal exploits cues in the environment available through its sensors, or a memory of what it has done and experienced since it left "the nest", or a combination of these options, to decide where to go next.

It was navigation which prompted Tolman (1948) to suggest that pure stimulus-response behavior was insufficient to account for certain behavior, in one of the studies leading up to the cognitive revolution. Tolman argued that rats trained in a maze learn a cognitive map, a map encoded in the brain, representing their environment and their current position. Early efforts to implement such cognitive maps by traditional artificial intelligence were not very successful. A robot developed at Stanford failed dramatically when the angle of the sun changed over time, changing the shadows in an otherwise static environment (Moravec, 1982; Brooks, 1991a). Shakey the robot at SRI had some success at navigation, but it stood still for long periods of time to "think" (in the meantime shaking a bit, hence its name), and it operated in a highly simplified, small world of a few rooms and brightly colored boxes (Nilsson, 1984; Dennett, 1991; Brooks, 1991a). These examples illustrate the difficulty of using sensory information and planned actions to maintain a central world model, in this case a central cognitive map module, even when the layout of the cognitive map is carefully programmed in beforehand.

More recent attempts by adaptive behavior researchers have taken another route. Rather than using a central functional module containing the cognitive map, separate from sensory and motor apparatus and operated upon by a central controller, they use systems based on distributed control and close interaction with the environment. In addition, they use learning, one of nature's bottom-up engineering methods, rather than a preprogrammed cognitive map.

One example is Mataric (1991). A robot was constructed capable of navigating successfully in a cluttered office environment. It applies the perception-action loop philosophy described earlier. One perception-action loop (or "layer") is used to avoid obstacles and follow walls. Building upon the first loop, another loop detects and

registers landmarks in combination with its own concurrent movements. A third loop uses that information to develop a kind of cognitive map (see figure 2.7).

However, this cognitive map is different from traditional conceptions of the cognitive map. It consists of a network of nodes, each representing a registered landmark and corresponding movement. The current location of the robot is represented by one of the nodes being active. Activation spreads to other nodes, thus generating "expectations" about what will be perceived when the robot performs the corresponding movement. A goal location can also become active, and activation will similarly spread out to neighboring nodes. This spreading of activation depends on the physical distance between landmarks. Consequently, locally at each landmark suggestions can be made as to which direction to go to reach the goal most rapidly. Overall, this results in the robot choosing the globally shortest path to the goal. Determination of the current position, map building, and action selection are not separated into distinct functional components, but they are all combined in this single map. There is no central planning mechanism figuring out the optimal path to the goal; the navigation behavior emerges as the result of interacting local units. Chapter 5 discusses the issue of navigation and cognitive maps again and in more detail.

2.4.3 Collective behavior

As a final example of a line of research in the adaptive behavior community, let us have a look at work that is concerned not just with one agent, but with multiple agents interacting with each other. Interestingly, complex collective behavior can arise from the interaction of simple agents. This can be viewed as the previously discussed principle of self-organization in systems with decentralized control, but applied on a larger scale.

As early as 1950, Grey Walter experimented with a pair of very simple robots interacting with each other. He noted that the resultant behavior from the robots could become surprisingly complex: "Crude though they are, they give an eerie impression of purposefulness, independence, and spontaneity" (Walter, 1950). Reynolds (1987) showed how the adaptive and well-coordinated behavior exhibited by flocking birds could be replicated by agents, "boids", that follow very simple rules based on the distance to their immediate neighbors (see figure 2.8). The resultant behavior is very smooth, adaptive, and life-like.

Work by Steels and co-workers demonstrates cooperation in learning agents. In one study (Steels, 1995), there is potentially mutual benefit for robots to cooperate, because they are hindered by parasites in obtaining energy. Even though in principle the robots compete for the same energy source, and cooperation behavior is neither programmed in beforehand nor suggested by explicit instruction, cooperation emerges spontaneously. Even stronger forms of altruism can arise. Brinkers and den Dulk (1999) investigated groups of evolving agents changing over generations because of simulated evolution. The experiment was set up such that some members of a group of agents will do much better if some other members sacrifice their "lives". Even though agents are selected on an individual basis, as is the case in natural selection, and therefore selfish behavior is the default expectation, such radically altruistic agents evolve and the altruistic behavior is evolutionarily stable.

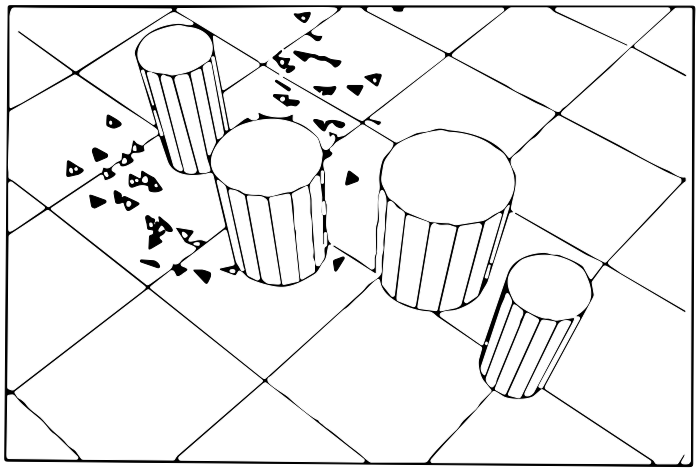

Figure 2.8: Flocking boids. The flock has split up to avoid the obstacles in the flight path, and will reassemble after the obstacles. Reprinted with permission from Craig W. Reynolds.

In many animals, cooperation is accomplished with the help of communication. In a simulation study by de Jong (1999), agents can potentially benefit from warnings by other agents that a certain type of predator is present. Each type of predator makes a specific location unsafe, e.g. the presence of a snake makes it unsafe to stay on the ground. The warning signals are learned in a bottom-up way. The agents start out using different, random warning signals for situations where different predators are present. Eventually they converge to common, reliable signals, to the point where they learn to rely on the other agents' warnings and avoid an unsafe location even if their own perception indicates that this location is safe.

These were all examples of cooperative behavior. The darker side of nature can be replicated as well. Nolfi and Floreano (1998) show, both in computer simulations and in robots, how co-evolution of predator and prey may yield relatively quick bottom-up development of complex behavioral strategies that oppose each other. The evolution of a slightly smarter strategy on one side pushes the other side to evolve a strategy that can cope with that, which in turn pushes the first side to develop an even smarter strategy, etc. This can be described as an "arms race". The predator species developed intricate pursuit and ambush strategies, while the prey species developed effective escape and avoidance strategies. These strategies, which betray sophisticated anticipation capabilities, are encoded as distributed information in artificial neural networks. Once again, the bottom-up engineering approach leads to a system without distinct or central functional components for storing the strategies, for planning the behavior according to the strategy, or for explicit anticipation representation.

2.5 Discussion

The arguments and examples presented in this chapter showed that the field of adaptive behavior does not provide mere implementations of the abstract theories of mainstream cognitive science. In contrast, in many cases successful agents are controlled by mechanisms that are very different in architecture and functioning from what was or would be expected given the abstract theories. Understanding those mechanisms often required completely new types of explanation, rather than explanations derived from those abstract theories. On the other hand, direct attempts of straightforward implementation of ideas from the abstract theories turned out to be problematic or unnecessary.

What does this mean? It was argued that implementation is neither trivial nor irrelevant for cognitive science. In contrast, if the implementation tells a different story than the abstract theory, suggesting that the abstract theory cannot be implemented or that some behavior is best achieved using other mechanisms than was anticipated, this has implications for the abstract theory. In this case, implementation of agents suggests that certain standard ideas in cognitive science, such as centralized control, cognitive maps, separation between working memory and long-term memory, and in general decomposition into isolated functional components, are in need of revision or, at the very least, should not be taken for granted as much as they are now.

Many of these standard ideas in cognitive science were arrived at and supported using the standard methodology of collecting data in controlled experiments. The fact that now some of those standard ideas are rejected prompts rethinking of that methodology. Experiments are important, but perhaps not for the same purpose as much of mainstream cognitive science has it. In some cases, there is an overly strong preoccupation with finding effects, without worrying what the effects mean. It seems to me that certain types of data—reaction time, strength, frequency, and accuracy of responses—are measures that are usually too indirect to straightforwardly derive from them reliable conclusions about the underlying mechanisms, and collecting additional indirect data does not help (they may have much practical significance, though, if they can be applied to industrial, clinical, or educational settings).

The most interesting data from cognitive science experiments, in terms of relevance for models of underlying mechanisms, may be data that indicate a capability unknown beforehand that humans or animals have, that map out what the capabilities are, or that disprove a previously assumed capability. Such data have direct implications for models of the mechanisms underlying behavior, in that they say what those models should be able to do and what they need not do. In that way, cognitive science heavily constrains adaptive behavior research. The exact speed and accuracy of the behavior are constraints that become important only later on, once we have successful models that account for the behavior itself. Adaptive behavior research, in turn, constrains cognitive science in the kinds of systems that are proposed as models. Certain kinds of systems work very well and other kinds of systems cannot be made to work at all. The test of implementation is crucial and should be brought into theorizing as early as possible. In addition, from successful agents the adaptive behavior approach can derive hypotheses about the biological mechanisms, which can subsequently be tested by experimentalists.

It was shown in adaptive behavior research that multiple agents may benefit from cooperating with each other. The same may be true for the two approaches discussed here. A stronger relationship between mainstream cognitive science and adaptive behavior research may be mutually beneficial.

Chapter 3

Reinforcement learning

Summary

This chapter provides a broad overview of the general concepts, methods, and important problems of reinforcement learning. The chapter discusses how reinforcement learning is situated within the broader context of other machine learning techniques and the adaptive behavior approach. Special attention is given to the distinction between Markovian and non-Markovian tasks, and how this has implications for the necessary mechanisms of agents dealing with these different classes of tasks.

3.1 Introduction

The adaptive behavior approach described in the previous chapter provides the general framework for this thesis' research, but the specific paradigm that is used is the machine learning paradigm called reinforcement learning. This chapter provides an introduction to reinforcement learning. Even though it is by no means a complete review of the field (see Sutton & Barto, 1998; Kaelbling, Littman, & Moore, 1996 for reviews), it attempts to describe many of the basic concepts, methods, and important problems associated with reinforcement learning. Special emphasis is given to those issues and methods that are the main focus of this thesis.

The next section describes the basic idea of reinforcement learning. It explains how reinforcement learning fits in with the general framework of the adaptive behavior approach, and how it relates to other machine learning paradigms and to psychology and biology. Section 3 formalizes the basic notions, and describes the distinction between Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs), which is an important distinction in reinforcement learning, and especially for this thesis. Section 4 describes the basic ideas behind different classes of solution techniques. Section 5 then discusses in more detail solution techniques for MDPs. Here, and in the rest of this thesis, a "solution technique" means any method that computes exact or approximate solutions. Section 5 also describes how the important problems of exploration and generalization can be dealt with. Section 6

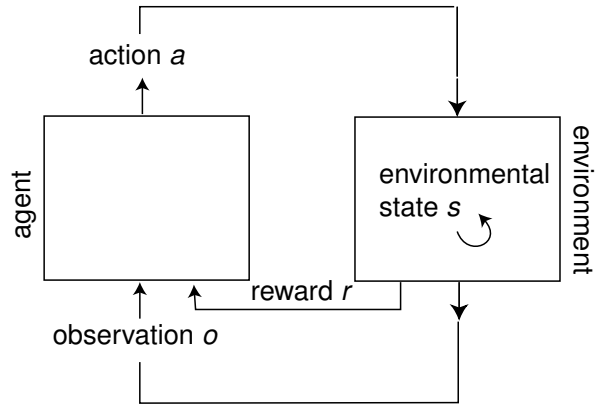

Figure 3.1: Schematic representation of the situation considered by reinforcement learning.

discusses in more detail solution techniques for POMDPs. The final section contains a general discussion, focusing again on the basic ideas and distinctions between different solution techniques.

3.2 Elementary concepts of reinforcement learning

3.2.1 The basic reinforcement learning problem

Reinforcement learning corresponds to a broad class of machine learning methods that allow an agent to learn how to behave in an environment based on scalar reward signals. The methods are so diverse that it is best to define the field by the problem they attempt to solve: the reinforcement learning problem.

The reinforcement learning problem considers an *agent* interacting with an *environment* (see figure 3.1). In control theory, the agent is called the *controller* and the environment the *plant*, but they mean virtually the same. The agent emits *actions* a that may change the *state* s of the environment, for example the position of a robot in its world, or the orientation of a robot arm; and the agent makes observations o, which provide the agent with information about the current state.

The agent's behavior is described by its *policy* π, which defines the actions it chooses in different situations. The goal of reinforcement learning is to learn a policy, based on a specific type of feedback from the environment to the agent called *rewards* r. Rewards are scalar values which indicate how good a particular situation is. The agent may focus on only maximizing the *direct* reward obtained after one action, but most of reinforcement learning is concerned with the case of *delayed* rewards, in which actions change the state and rewards in the future should also be taken into account. The objective, then, is to learn a policy which maximizes some measure of long-term reward.

It is widely accepted in artificial intelligence that the more complex desired behavior becomes, the more difficult it becomes to program the behavior. Thus, one of the holy grails of artificial intelligence is a system that can develop itself, more or less autonomously, on the basis of its own experiences with the world (Turing, 1950). Reinforcement learning, no matter how simple and limited it is at this point, may contribute to this long-term goal of artificial intelligence. It does not require a person to program the desired behavior by hand, or even show the agent desired actions in different situations. Instead, it potentially allows the agent to learn virtually autonomously, based only on reward signals if the agent reaches particular goals.

Reinforcement learning realizes many of the commitments of the general adaptive behavior framework: it emphasizes the close interaction of an agent with its environment, it focuses on perception to action cycles and complete behaviors rather than separate functions and functional modules, it uses bottom-up development (learning) of intelligence, it is not based on symbolic representations, etc. Because of this, studying reinforcement learning is a way of working on the basis of adaptive behavior principles, while at the same time having the benefit of a somewhat more solid theoretical and mathematical basis than much of adaptive behavior research. We will see in the remainder of this chapter, as well as in other chapters, how reinforcement learning provides a way to study, in a more or less formal way, specific challenges identified in adaptive behavior research, such as scaling up to more complex tasks and going from "reactive" to "representation-hungry" problems.

3.2.2 Relationship with human and animal learning

This basic picture of learning on the basis of rewards is shared with an influential branch of psychology, called *behaviorism*. Behaviorist psychology has studied learning on the basis of rewards in animals and humans since the early 20th century, usually under the name "operant conditioning" (Skinner, 1938, 1991). The central idea was already formulated by Thorndike (1911) as the Law of Effect (p. 244):

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.

Operant conditioning has proven to be a very powerful and useful teaching method. Animals have been taught surprisingly complex tasks in this way; for instance, pigeons learned how to play table tennis. Children with severe mental disabilities have got rid of self-destructive behaviors and have acquired social skills no one thought possible (Malott, Whaley, & Malott, 1993). In these studies the rewards were controlled explicitly by the experimenter. However, the success of the behaviorist techniques suggests that reinforcement learning is an important learning method for animals in

their natural surroundings as well. This is possible because the world "contains natural rewards", e.g. food and sex, as well as natural penalties, e.g. pain. Of course, the world does not really "contain" rewards and penalties as such, independent of organisms who seek and avoid them. In fact, evolution has shaped organisms so that certain stimuli function as rewards or penalties—and this can be very different for different organisms.

Thus, like a lot of other adaptive behavior research, the formalized models of reinforcement learning are inspired by and have some similarity with biological systems. And as in other adaptive behavior research, the influence goes both ways. The early developments on temporal difference learning (see below), for instance, were based in part on psychological ideas about "secondary reinforcers", which correspond to situations that reliably precede reward. Conversely, animal learning data have been modeled using formal reinforcement learning models (Sutton & Barto, 1981; Hallam, 1999), and recently there have been neuroscience studies that suggest that several important concepts in reinforcement learning, such as temporal difference errors and exploration, have fairly direct neural correlates in the brain (Schultz, Dayan, & Montague, 1997; Kakade & Dayan, 2000).

Even though the work of this thesis is not directly concerned with behavioral or neuroscience experiments, it may, therefore, still contribute to explaining them in the long run, by contributing to formal reinforcement learning. The relevance for biology is, I believe, increased by this thesis' commitment to neural networks. Nevertheless, whenever one takes an artificial intelligence approach and attempts to build machines that can learn interesting things, at some points one has to make fairly arbitrary choices that have nothing to do with biology, and that in some cases can immediately be seen to be biologically unrealistic. That is the price that is paid for making things work, and that is what is done in this thesis.

3.2.3 Reinforcement learning versus supervised and unsupervised learning

In order to understand reinforcement learning's place within the entire field of machine learning, it is instructive to classify learning approaches based on the quality of the training information provided by the environment of an agent (Hinton, 1987; Gullapalli, 1991).

In supervised learning tasks, the environment tells the agent exactly what the output should be for each input. The agent must learn the mapping from inputs to outputs, based on a limited number of input-output examples, and it must learn to generalize correctly to new cases. In unsupervised learning tasks, the environment only presents inputs to the network, without providing any information about the output. Learning consists of autonomously clustering the inputs, based on some measure of similarity between inputs.

Reinforcement learning tasks are in between supervised and unsupervised learning tasks. Unlike unsupervised learning, the agent receives *some* information from its environment on how well the output is doing in the environment. However, this information from the environment is *evaluative* rather than *instructive*. In supervised learning, if the output consists of a vector, the training information consists of a vector

of the same size. In reinforcement learning the training information is only a scalar, a single number evaluating the success or failure of an entire sequence of outputs. As a consequence of this, reinforcement learning will usually take more time than supervised learning to learn the same input-output mapping, if the correct outputs are known such that supervised learning can be used. For obvious reasons, reinforcement learning is sometimes called *semi-supervised* learning (Arbib, 1995).

Even if one wishes to use supervised learning, determining a desired output vector may be difficult; for example, when the output vector serves as a control signal for a robotic arm with many degrees of freedom. The difficulty lies in the fact that the control signal has a very complex relationship with the end result in the environment. Therefore, errors in the end result cannot be translated straightforwardly into errors in the output vector. This is called the *distal error problem* (Jordan & Rumelhart, 1992; Barto, 1995). One solution is to first learn a model of what happens between control signal and end result. This model can then be used to compute supervised errors for the output vector (Jordan & Rumelhart, 1992, see also section 3.5.1.3). Another way to approach this is to use the distal error as the basis for a scalar reward signal, such that the task can be considered a reinforcement learning task.

Reinforcement learning can, in principle, be more flexible than supervised learning. The system "keeps an open mind" to any solution for the problem at hand, instead of being constrained to the one solution the supervisor has decided on and which may not even be the best one. It can always keep striving for better performance rather than hold on to a fixed or pre-determined solution; and whereas the supervised solution may work in one, stable environment, once the environment changes, that solution becomes worthless. The supervisor would have to specify a new solution. When a reinforcement learner notices that the success of its behavior deteriorates, perhaps as a result of a changing environment, it can automatically start exploring again, looking for outputs that do well in this new environment.

Unsupervised learning has been suggested as a natural alternative to supervised learning for many tasks (Kohonen, 1995; Murre, 1992). Some tasks indeed lend themselves to unsupervised learning. However, such tasks can, by definition, not be directed toward the satisfaction of constraints imposed by the task, since the agent receives no environmental feedback about its outputs. An example will clarify this. Suppose a robot's world contains two distinct landmarks L and R. One task for the robot may require it to turn left at landmark L and turn right at landmark R; but another task may require it to turn *right* at landmark L and *left* at landmark R. Clearly, it does not suffice to just learn to see the difference between landmarks L and R, which is all that unsupervised learning can accomplish. Landmarks L and R each have to be associated with either action "left" or action "right", depending on what works best in the particular task. The only way to learn how to behave in an initially unknown environment is by being told what to do (supervised learning) or by taking into account what the consequences are of actions tried out in the environment (reinforcement learning).

3.2.4 Exploration versus exploitation

The reinforcement learning agent is rewarded for "good" sequences of actions. However, it is not *told* what the correct actions are. It has to learn this by trial and error.

The process of trying out actions in order to determine how good they are is called *exploration*. Once the agent has found good actions, it can instead do *exploitation* of the learned policy. The agent is faced here with a dilemma. Once it has found a policy that appears to do well, should it be content with its performance and look no further? On the one hand, there may be much better policies. On the other hand, more exploration usually means sacrificing current performance without a guarantee of finding a better policy. This is the *exploration/exploitation dilemma*.

3.2.5 Structural and temporal credit assignment

Consider the case where an agent does not yet have a very good policy, and it receives a reward after a long sequence of states and actions. Now it has to decide which of the many actions were responsible for that reward. It seems natural to at least attribute responsibility to the last action. In addition, many actions before the last action must have also contributed. But which ones? It is likely that not all actions contributed equally; in fact, it is likely that many actions were in fact bad, and the reward was obtained *despite* these bad actions. The challenge is to determine which actions in an action sequence are good and which are bad, and similarly, which states are good because they precede reward and which are bad. This is the problem of *temporal credit assignment*.

Given that we have identified which states and actions in a sequence deserve credit for a delayed reward, there is still the problem of *structural credit assignment*. This is the problem of assigning credit to the individual adjustable parameters that define the policy. Suppose the policy is represented by a neural network. Then an individual weight change may influence the probability of not just one action but many of the possible actions. The challenge is to determine how to change the weights such that actions that deserve credit are indeed made more likely. For example, the backpropagation procedure may be used to propagate errors on the output (action) side of the network to individual weights. Backpropagation and variations of backpropagation are the solution to the structural credit assignment problem in most of the reinforcement learning agents described in this thesis. Note that in direct reward problems we have only the structural credit assignment problem and not the temporal credit assignment problem.

3.2.6 Discrete versus continuous tasks

So far we have talked about *discrete* timesteps, actions, and states, without making this assumption of discrete problems explicit. Most of the theory and practice in reinforcement learning is concerned with discrete tasks, but it is possible to extend many of the concepts and methods to continuous tasks. In many cases, and similar to other fields, this boils down to "discretizing" or "quantizing" the continuous variables, and then applying the same or somewhat adapted methods as we use for discrete tasks (Doya, 2000). For instance, continuous time may simply be sampled at a fixed frequency such that we obtain discrete timesteps. However, at the same time it is important to realize that direct application of the methods developed for discrete methods is not always practical, even if it is theoretically possible. For example,

discretization of a continuous state space may lead to a prohibitively large number of states. Chapters 4, 5, 6, and 7 all contain elements dealing with (parts of) this problem. In the remainder of this chapter, however, we will focus mainly on the discrete case, for reasons of ease of presentation.

3.2.7 Online versus offline learning

It is useful to distinguish between *online* reinforcement learning and *offline* reinforcement learning (see also Wyatt, 1995). These really correspond to different goals of reinforcement learning research.

Let's say the final application that we have in mind is robot control. In online reinforcement learning, the goal is to build a learning robot: we want to construct a robot which, when put in various unknown environments, can learn to perform certain tasks. The value of this is that we could then use a single type of robot in various tasks and various environments, and use the robot for unknown environments (for example, enemy terrain or other planets). In this case, one of the most important constraints is the number of learning experiences needed to learn the task: if it takes millions of learning experiences before the task is learned the robot would usually be useless. Furthermore, the exploration/eploitation dilemma is very important here, because we want the robot to start to exploit its knowledge as soon as possible. Finally, safety is important: above all, we want to prevent the robot from harming others or itself.

In offline reinforcement learning, on the other hand, the goal is to develop a good controller for some task, and to develop the controller using reinforcement learning. Note that in this case we do not aim for a learning robot as a goal in itself, we only want a robot that can perform some task(s), and we happen to use learning as the means to get there. The value of this approach is that it is in general very difficult to program robot controllers, and learning may yield a controller that is better than a handcoded one. In this case, the main constraint is the final performance of the robot controller. We do not care so much about the number of learning experiences needed to train the controller. Learning can be done in a simulated version of the robot, making learning experiences much cheaper than if all learning was done on the real robot. For the same reason, the exploration/exploitation dilemma and safety issues are much less important than in online learning. Balancing exploration and exploitation is only important to allow reinforcement learning to reach satisfactory solutions within a reasonable time, and the simulated robot may kill itself as often as it likes.

Of course, online and offline reinforcement learning are not necessarily completely independent enterprises. It may be a good approach to combine the two. We could first develop, offline in simulation, a controller for the robot. The controller developed in this way is likely to be imperfect for the real robot, e.g. because of errors in the simulation. We could then "finetune" the controller on the real robot, using online reinforcement learning. This is then feasible within a reasonable time, because at this point we already have a fairly good controller.

3.3 Reinforcement learning formalized

3.3.1 Formal model of the environment

In the finite, discrete case, a reinforcement learning problem contains:

- A time counter $t = 0, 1, 2, 3, \ldots$.
- A finite set of environmental states $S = \{S_1, S_2, S_3, \ldots, S_N\}$. The state at time t is denoted by s_t. To simplify notation, $s \in S$ and $s' \in S$ are also used to refer to states from this set.
- A finite set of actions $A = \{A_1, A_2, A_3, \ldots, A_M\}$. The action at time t is denoted by a_t. To simplify notation, $a \in A$ and $a' \in A$ are also used to refer to actions from this set.
- A finite set of observations $O = \{O_1, O_2, O_3, \ldots, O_L\}$. The observation at time t is denoted by o_t. To simplify notation, $o \in O$ and $o' \in O$ are also used to refer to observations from this set.
- A finite set of reward values $K = \{K_1, K_2, K_3, \ldots, K_P\}$, where each $K_i \in \mathbb{R}$. The reward received after executing action a_t in state s_t is denoted by r_{t+1}. To simplify notation, $r \in K$ and $r' \in K$ are also used to refer to reward values from this set.

In the most general case, the dynamics of the environment may be specified by the probability distribution

$$\Pr\{s_{t+1} = s, r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} \tag{3.1}$$

for all s', r, and all possible values of past events $s_t, a_t, r_t, \ldots, r_1, s_0, a_0$. If the environment has the *Markov property*, then the environment's response at time $t+1$ depends only on the events at time t, and the following holds:

$$\begin{aligned} &\Pr\{s_{t+1} = s, r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} = \\ &\Pr\{s_{t+1} = s, r_{t+1} = r \mid s_t, a_t\}. \end{aligned} \tag{3.2}$$

Thus, the direct reward and the next state depend only on the current state and the current action, and not on events before that time. This is a reasonable assumption, and one that is made also in other fields such as physics and systems theory (Ashby, 1960). It basically says that the state summarizes all that has happened in the world and that is relevant for knowing how the world will develop from now on (given the same actions). If the state variables identified by the experimenter do not have this property, it means that the state is not really identified, and other information must be incorporated into the state variables to arrive at the true state (Ashby, 1960).

Given that the environment has the Markov property, the environment is defined by:

- A *state transition function* $f_s : S \times A \rightarrow \Pr(S)$, which defines a probability distribution over states. $\mathcal{P}^a_{ss'}$ denotes the probability of making a transition from state s to state s' given action a. At every timestep t, the state is set to s_t.

- An *observation function* $f_o : S \rightarrow \Pr(O)$, which defines a probability distribution over observations. $\mathcal{P}_o^s$ denotes the probability of making observation o given state s. At every timestep t, the observation is set to o_t.
- A *reward function* $f_r : S \times A \times S \rightarrow \Pr(K)$, which defines a probability distribution over direct rewards. $\mathcal{R}_{ss'}^a$ denotes the expected direct reward given action a in state s and a transition to state s'. At every timestep t, the reward is set to r_t.

3.3.2 Formal model of the agent

The agent dealing with the reinforcement learning problem is defined by:

- An input or observation space, which corresponds to the set of observations O.
- An output or action space, which corresponds to the set of actions A.
- An *internal state* space Z.
- A *parameter* space W.
- An *internal state transition function* $f_z : Z \times O \times W \rightarrow \Pr(Z)$, which defines a probability distribution over internal states. At every timestep t, the internal state is set to z_t.
- An *action function* $f_a : Z \times O \times W \rightarrow \Pr(A)$, which defines a probability distribution over actions. At every timestep t, the action is set to a_t.
- A *parameter transition function* $f_w : Z \times O \times W \times A \times K \rightarrow W$ which is defined by the reinforcement learning algorithm and which computes w_t.

The combination of the internal state transition function and the action function comprises the agent's policy π, and this policy is completely defined by the parameters w. Note that, by definition, the internal state has the Markov property.

The internal state and the parameters are both internal variables of the agent, and in principle we could merge them into one set of variables. Nevertheless, in many cases it is practical to distinguish them. The parameters are the variables that are modified by the learning algorithm, and they typically change on a long timescale. They constitute the agent's long-term memory. For example, the weights in a neural network are parameters. The internal state, on the other hand, consists of those internal variables that are not part of either the observation space or the action space, that may change over time even without learning, and that together with the current observation determines the action using the action function, which is parameterized by the parameters. Internal states typically change on a short timescale, and they constitute the agent's short-term memory. For instance, the state of a Finite State Automaton (FSA) constitutes an internal state, and the recurrent activations in a recurrent neural network constitute an internal state. However, note that the distinction between parameters and internal states can be blurry. For instance, there are studies where the weights of a neural network, traditionally thought of as the

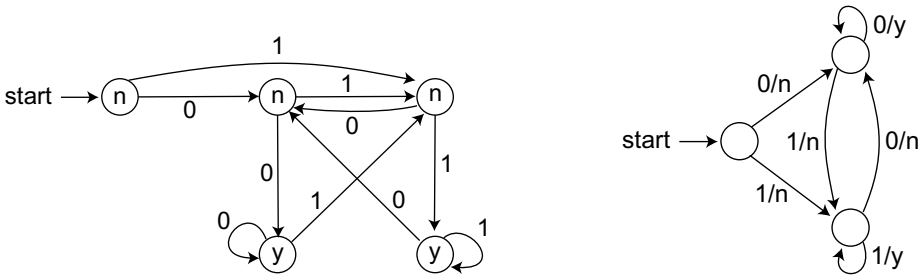

Figure 3.2: Fig. a (left). Moore input-output FSA. There are two inputs, 0 or 1, and two outputs, y (yes) or n (no). Outputs are associated with states. Fig. b (right). Mealy input-output FSA. Outputs are associated with edges. This Mealy machine is equivalent to the Moore machine depicted on the left. Adapted from Hopcroft & Ullman (1979).

parameters, can change on a short timescale, providing the system with short-term memory (Schmidhuber, 1992b; Urzelai & Floreano, 2001).

Even with the distinction between parameters and internal state in place, for one and the same agent there are still multiple, but equivalent ways to "divide the work" between internal state transition function and action function. This corresponds to different choices of what the internal states correspond with in the system under study. The action function can be written as a function of only the internal state and the parameters: $a_t = f_a(z_t, w_t)$. In the language of input-output FSAs, that corresponds to the Moore machine: the action is associated with *states* of the FSA (Hopcroft & Ullman, 1979, see figure 3.2a). Alternatively, the action function may, as above, be written as a function of internal state (different from before) and parameters *and the observation*: $a_t = f_a(z_t, o_t, w_t)$. In the language of input-output FSAs, this corresponds to the Mealy machine: the action is associated with *edges* of the FSA (see figure 3.2b). As we know from discrete automata theory (Hopcroft & Ullman, 1979), the Moore machine and Mealy machine interpretations are equivalent and can always be rewritten into each other (see figure 3.2). Rewriting changes the internal state transition function and action function, and it also changes the number of internal states and what the internal states correspond with in the agent.

If the agent does not use observations for its action selection at all, i.e. the action function can be written as $a_t = f_a(z_t, w_t)$ and the internal state transition function as $z_t = f_z(z_{t-1}, w_t)$, in the language of control theory it is said to perform *open loop* or *feedforward* control. In that case, it basically emits a fixed series of actions (or probability distributions over actions), independent of sensory feedback. Feedforward control is rarely studied in reinforcement learning (but see Hansen, Barto, & Zilberstein, 1996), because it is not very flexible, and not very robust in the face of noise and uncertainty about the state of the environment and the results of actions. When the agent does use observations for its action selection, it performs *closed loop* or *feedback* control. This thesis, like virtually all reinforcement learning work, is only concerned with feedback control.

Within feedback control, we can make another distinction. If the action function can be written as a function of only the observation and the parameters, i.e. $a_t = f_a(o_t, w_t)$, then we say the agent uses a *perception-based* or *reactive* or *memoryless* policy. In this case, the agent makes no use of internal state or short-term memory for its action selection. If, on the other hand, the action function must include internal state to be an accurate description of the action selection, i.e. $z_t = f_z(z_{t-1}, o_t, w_t)$ and $a_t = f_a(z_t, o_t, w_t)$ (Mealy interpretation) or $z_t = f_z(z_{t-1}, o_t, w_t)$ and $a_t = f_a(z_t, w_t)$ (Moore interpretation), the agent uses an *internal state-based* policy.

3.3.3 Measures of long-term reward

The reinforcement learning agent's job is to maximize some measure of long-term reward. The rewards received after timestep t are denoted by r_{t+1}, r_{t+2}, etc., and the measure of long-term reward is called the *return* R_t, which is defined as some function of this sequence of rewards. The objective may be to maximize the sum of future rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \tag{3.3}$$

where T is the final timestep. This definition of the return makes sense if the task of the agent is composed of separate episodes which end at time T, after which a new episode starts. For example, a robot must accomplish a certain goal, such as cleaning a room, and then start again (in a different room). These are called *episodic* tasks. If there are no such episodes, we are dealing with *continuing* tasks. In the case of continuing and possibly infinite tasks the definition of R_t as the sum of future rewards does not make much sense because it may become infinite. In that case, we may use the following definition of the return, the *discounted* return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.4}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*. If $\gamma < 1$, this discount rate basically makes rewards that are further in the future less valuable. This corresponds to the intuition, also found in economic theory, that the same reward is worth more if it is available immediately than if it is available only after a long time. Mathematically, one desirable property of this definition is that the infinite sum always has a finite value (as long as rewards are bounded). One might argue that for real-world agents truly continuing tasks do not exist because all agents have a finite life. However, independent of whether we consider episodic or continuing tasks, another desirable property of the discounted return definition is that it automatically favors policies that get to rewards faster. This is practically always what we want, regardless of whether we consider continuing or episodic tasks. If $\gamma = 0$, the agent does not care at all about delayed rewards, only about direct rewards. If $\gamma = 1$, on the other hand, this definition reduces to definition 3.3. R_t as defined by 3.4 is used throughout this thesis.

3.3.4 MDPs versus POMDPs

In a Markov Decision Process (MDP, Markovian task), the observation probabilities as defined by the observation function are not really probabilities but reduce to *one*

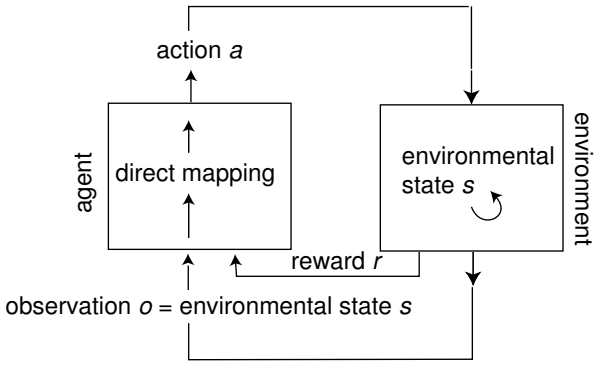

Figure 3.3: Schematic representation of a Markov Decision Process. The agent's observation is equivalent to the state of the environment, and it can simply learn a direct mapping from observations (states) to actions to accomplish an optimal policy.

particular observation for *each* particular state. In other words, the observation simply *is* the environmental state, and the concepts observation and state are interchangeable (see figure 3.3). The environment's response at time $t+1$ (defined by 3.2) can then be said to depend only on the current *observation* o_t and the current action a_t:

$$\begin{aligned} &\Pr\{s_{t+1} = s, r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} = \\ &\Pr\{s_{t+1} = s, r_{t+1} = r \mid o_t, a_t\}. \end{aligned} \tag{3.5}$$

This means that optimal prediction of how the environment will develop is a function only of o_t and a_t, and optimal control can be accomplished by an agent that simply maps current observations to actions. For this reason, virtually all work on MDPs employs agents that learn direct mappings from observations (states) to actions: memoryless policies, that do not make use of internal state. Chapter 4, however, challenges this conventional wisdom for certain cases. Note that an MDP can be viewed as a Markov Chain with a distinct state transition matrix for each action.

In a Partially Observable Markov Decision Process (POMDP, or non-Markovian task[1]), the observation function is a more complex function than the simple one observation per state mapping of an MDP. Different states may have similar probability distributions over observations. Thus, different states may look the same to the agent, and the same state may look different at different times. For this reason, POMDPs are said to have *hidden state* or *perceptual aliasing*. See figure 3.4 for a visual representation of this idea. A POMDP can be viewed as a Hidden Markov Model (HMM) with a distinct state transition matrix for each action.

[1]Technically speaking, non-Markovian tasks can also be interpreted as including those tasks where the underlying process itself does not have the Markov property, and the "state", even if it were known, does not depend only on the last state and the last action. For the sake of simplicity, when we speak about non-Markovian tasks in this thesis, we always mean POMDPs, where the underlying process is an MDP, such that if we reconstruct the state perfectly, we again have an MDP.

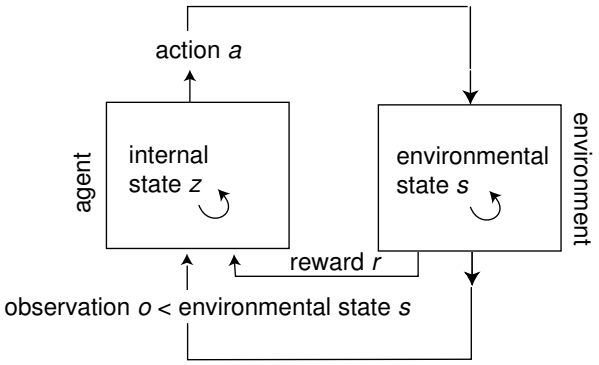

Figure 3.4: Schematic representation of a Partially Observable Markov Decision Process. The observation provides some information about the environmental state but not complete information; this is indicated by the $<$ symbol. For optimal performance the agent may have to use some form of internal state.

In a POMDP, the environment's response is no longer a function of only the current observation and action, i.e. equation 3.5 no longer holds. This means that a memoryless policy may no longer be optimal, compared to what is theoretically possible if one remembered the entire history of observations, actions, and rewards. For instance, two hallways in a building may look the same for a robot's sensors, but in the first hallway the optimal action is "go left" and in the second hallway the optimal action is "go right". A memoryless policy cannot deal with this situation, but an internal state-based policy can. For instance, it may be possible to disambiguate two hallways that look identical by remembering that the robot has just passed the men's room.

Note that many realistic problems are partially observable in this sense. An agent's sensors will in general be noisy, and limited in the amount of detail they can pick up. Even with very good sensors, it is often the case that different places in the world look the same (such as hallways). An alternative to internal state for solving partial observability is to add or improve sensors. For example, a mobile robot may be fitted with specific sensors helping it to disambiguate hallways. To the extent that that is possible or affordable or desirable, this provides an alternative to internal state. However, note that sensors will *always* have some noise and associated uncertainty; and adding sensors will not help in cases where an agent must act differently in the same perceptual situation depending on a context that cannot be perceived at that moment. This is the case, for example, when an agent first has to move to location A and then to location B in the same world, *passing through the same states*, or when a certain action depends on instructions or observations the agent has received earlier on. For example, consider the case where a robot observes that a certain door is locked, blocking the path to its goal. It turns back to take a different route. When it now arrives at a junction, from where it earlier took the route towards the locked door, it should not have forgotten that the door was locked and again take the previous route to the locked door. Instead, it should use short-term memory to decide that now this

junction, *which the robot can perceive perfectly and tells the robot exactly where it is*, should no longer be associated with the action corresponding to the old path, but to another path. In any case, biological intelligence can certainly deal with the problem of partial observability to some extent, so if we wish to understand how that works, we need to consider partially observable problems.

POMDPs are the reinforcement learning equivalent of so-called "representation-hungry" (Clark, 1997), "anticipatory behavior" (Keijzer, 2001), or "minimally cognitive" (Slocum, Downey, & Beer, 2000) tasks in the adaptive behavior literature. Like POMDPs, these are tasks where good behavior cannot be accomplished by a reactive agent that responds directly and only to environmental inputs. In adaptive behavior research, this has spawned a lot of debate about whether or not this creates a need to return to the full-blown world models that adaptive behavior researchers have rejected from the outset. Specifically, the question has been raised whether the representations (which can be interpreted as internal states) that must supplement direct sensory information have to be symbolic in nature, signifying a return to symbolic reasoning systems (Clark, 1997; Keijzer, 2001; van Gelder, 1998). POMDPs allow us to investigate these questions within the reinforcement learning framework, and solution techniques for POMDPs may therefore shed some light on these general adaptive behavior debates. Most of the work in this thesis is concerned with POMDPs and the issues that are associated with this difficult, but also interesting and realistic class of problems.

3.4 Solution techniques

3.4.1 Model-based versus model-free techniques

One dimension in which solution techniques for MDPs and POMDPs can be distinguished is whether or not they use a model of the environment. Here a model of the environment means an explicit representation of the environment's state transition function, observation function, and reward function. In forward models, this is more or less literally so, and the model behaves as if it is the environment. In inverse models, really the inverse of the environment's state transition function, observation function, and reward function are modeled (given that the inverses are well-defined), such that it takes as inputs observations and rewards and gives as outputs actions that lead to those observations and rewards. The advantage of an inverse model is that it may be used directly for control: the inverse model is given "high reward" as input, and it provides an action as output that accomplishes a high reward, because this is an action that in this environment is apparently associated with high reward. Note that this only works in the case of direct reward. Forward models are more general in that they can be used in the delayed reward case, but they can only be used in a less straightforward way. Several such ways are described below. In the remainder of this chapter and this thesis, when models are discussed they refer to forward models.

Model-based or indirect reinforcement learning methods use a model as the basis for their search for a good policy. Model-free or direct reinforcement learning methods, on the other hand, do not use a model, and instead learn directly on the basis of

online interactions with the environment. In model-free work, the environment may of course be simulated in a computer, but the agent does not have access to an explicit representation of the environment's state transition function, observation function, and reward function. It is possible, in many cases, to go from the model-free case to the model-based case by first *learning* a model, using the interaction with the environment. After a sufficiently correct model has been learned, model-based solution techniques can be applied to it. This is reminiscent of adaptive control theory ideas where a system identification phase is followed by an actual control design phase.

This thesis' technical chapters only discuss model-free solution techniques, but in the current chapter model-based techniques are discussed as well. This is done not only to provide a more complete survey of the field, but also because in many cases model-free methods can be viewed as approximations to model-based techniques. Moreover, understanding exact model-based methods may help us to better understand the nature of reinforcement learning problems in general, and to better understand what the objective should be for model-free methods. Furthermore, a description of model-based methods makes clear what their limitations are, especially in terms of computational complexity, and therefore helps in understanding some of the reasons for turning to model-free methods. Model-based methods may inspire model-free methods or help delineate constraints which model-free methods have to take into account or can exploit. Conversely, model-free methods may inspire novel model-based techniques. These novel model-based techniques may, for instance, be useful approximations to exact model-based techniques, which are computationally intractable for large problems.

3.4.2 Direct policy search versus value functions

Another, more or less independent dimension in which solution techniques can be distinguished is whether they take a *direct policy search* approach or a *value function* approach. In its simplest form, direct policy search searches directly in the space of policies to find one that works well. That is, it evaluates policies as a whole, and adjusts the parameters that define a policy based on these global evaluations. For instance, an evolutionary algorithm can be used to search for the parameters of a policy, and the fitness measure may correspond to the average long term reward measure obtained by the policy.

In the other approach, based on value functions, the basic idea is to associate individual states or state-action pairs with values. Those values are typically updated based on the direct rewards and the values of the next states or state-action pairs, such that states or state-action pairs that precede high rewards get high values themselves. Policies are derived more or less indirectly, by determining which action has or leads to the highest value in the learned value function. Most of the reinforcement learning methods in the literature use value functions in one way or another, and so do the methods investigated in this thesis.

Even though the distinction between direct policy search and value functions may seem quite clear-cut, in practice it really is not. As we shall discuss in more detail below, approaches that are described by many researchers as direct policy search methods may use something like a value function to aid the mechanism that adjusts the policy parameters directly (Sutton, McAllester, Singh, & Mansour, 2000; Baird

& Moore, 1998; Meuleau, Peshkin, Kim, & Kaelbling, 1999). Such methods are still considered direct policy search methods because the policy has a representation independent of the value function, and the value function could be replaced by another function that guides the policy search toward good policies. On the other hand, value function methods may be adjusted such that values are transformed toward something like "preferences" for different actions, which are there for policy determination purposes only and which have little to do with values in the traditional sense (Harmon & Baird, 1996; Baird, 1999). Despite this blurriness, the distinction between direct policy search and value functions remains useful in aiding our thinking about different methods.

3.4.3 Value functions and the Bellman equation

As noted above, the work of this thesis and most of the work in the reinforcement learning literature uses value functions. States or state-action pairs are associated with values that indicate how "good" the states or state-action pairs are. A value associated with a state or state-action pairs represents the expected return from that particular state or state-action pair, given a certain policy. Formally, the state-value $V^\pi(s)$ of a state s given policy π is defined as

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\Big\} \tag{3.6}$$

where E_π denotes the expected value given that the agent follows policy π. Similarly, the action-value $Q^\pi(s,a)$ of action a in state s given policy π is defined as

$$Q^\pi(s,a) = E_\pi\{R_t \mid s_t = s, a_t = a\} = E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\Big\}. \tag{3.7}$$

The nice thing about values as defined in this way is that they can be written recursively, that is, in terms of the values of possible successor states or state-action pairs. For state-values, this corresponds to

$$\begin{aligned} V^\pi(s) &= E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\Big\} \\ &= E_\pi\Big\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\Big\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}\Big[\mathcal{R}^a_{ss'} + \gamma E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\Big\}\Big] \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \end{aligned} \tag{3.8}$$

where $\pi(s,a)$ is the probability of choosing action a in state s under policy π. This is called the *Bellman equation* for $V^\pi(s)$. $V^\pi(s)$ is the unique solution to its Bellman

equation. For $Q^\pi(s,a)$, we have a similar Bellman equation. Bellman equations form the basis for many MDP and POMDP solution techniques.

What we were interested in, in the end, is finding a good or preferably the best policy. There may be more than one best policy, but there is always at least one. For convenience, all optimal policies are denoted by π^*. They share the same optimal value function. For states, this is defined as

$$V^*(s) = \max_\pi V^\pi(s) \tag{3.9}$$

and for state-action pairs, it is defined as

$$Q^*(s,a) = \max_\pi Q^\pi(s,a). \tag{3.10}$$

The corresponding Bellman equation, known as the Bellman optimality equation, for $V^*(s)$ is

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ &= \max_a E_{\pi^*}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \Big\} \\ &= \max_a E_{\pi^*}\Big\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a \Big\} \\ &= \max_a E\{ r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \end{aligned} \tag{3.11}$$

and for $Q^*(s,a)$ the Bellman optimality equation is

$$\begin{aligned} Q^*(s,a) &= E\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \} \\ &= \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')]. \end{aligned} \tag{3.12}$$

The goal of value function-based solution techniques is to find $V^*(s)$ or $Q^*(s,a)$. Once that is accomplished, the optimal policy simply consists of taking that action a in each encountered state s for which $Q^*(s,a)$ is highest or, equivalently, that action that leads to the highest expected direct reward plus discounted $V^*(s')$ in the next state s'. After all, that corresponds to the action that leads to the highest expected return. Thus, in this way *globally* optimal behavior can be accomplished by examining values that are available *locally* at the current state.

3.5 MDP solution techniques

3.5.1 Model-based MDP solution techniques

3.5.1.1 Policy iteration

The classical solution techniques for MDPs given an exact model of the environment are collectively known as dynamic programming. These techniques iteratively compute the exact value function using the model. They do not actually interact with the environment, so they do not really do trial and error learning, and they do not need to concern themselves with the exploration/exploitation dilemma, for instance. For these reasons, most authors do not regard dynamic programming algorithms as "genuine" reinforcement learning algorithms. In any case, these techniques do consider the reinforcement learning problem, in the general sense, and they are important to understand if one wants to understand genuine reinforcement learning techniques.

One algorithm developed and studied in dynamic programming is policy iteration. Policy iteration consists of two phases that are performed consecutively and repeatedly: policy evaluation and policy improvement. Starting with a random current policy, policy evaluation computes, using an iterative algorithm, the value function of the current policy. Policy improvement then improves the current policy, using this value function. Policy evaluation next computes the value function of this new policy, etcetera. This process is guaranteed to converge to the optimal value function and corresponding optimal policy.

Policy evaluation evaluates the current policy by transforming the Bellman equation for V^π (equation 3.9) into the following update rule:

$$V^\pi_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^\pi_k(s')] \tag{3.13}$$

for all s. After an entire sweep through the set of states is completed, the procedure starts again, using the new estimations of V^π. Each such iteration brings the value function closer to the real value function for policy π, and it is guaranteed to converge to V^π as $k \to \infty$.

After a sufficiently close approximation to V^π has been found using this policy evaluation procedure, e.g. when the largest change in $V^\pi_{k+1}(s)$ is below a certain threshold, the policy improvement phase starts. Policy improvement simply chooses those actions for the new policy[2] which seem best according to the current value function:

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a E\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \end{aligned} \tag{3.14}$$

for all s. The policy improvement theorem (see Sutton & Barto, 1998) shows that the new policy π' must be as good as or better than the old policy π. Next, the value of

[2]For convenience, only deterministic policies are considered here. The ideas extend straightforwardly to stochastic policies, however. Then different actions of maximal but equal value must each get some probability of being selected, and all suboptimal actions must get zero probability.

the new policy is computed by applying policy evaluation again, etcetera. If the new policy is as good as the old policy, that is $V^{\pi'} = V^{\pi}$, then from 3.14 it follows that for all s:

$$\begin{aligned} V^{\pi'}(s) &= \max_a E\{r_{t+1} + \gamma V^{\pi'}(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^{\pi'}(s')]. \end{aligned} \tag{3.15}$$

This is the same as the Bellman optimality equation (3.11). Therefore, V^{π} and $V^{\pi'}$ must equal V^*, and π and π' must be optimal policies π^*. Thus, the consecutive phases of policy evaluation and policy improvement will eventually lead to the optimal value function and optimal policy. However, note that in general, both the policy evaluation and policy improvement phases require many sweeps through the entire state and action set. With large numbers of states and actions, this can become a computationally very expensive procedure.

3.5.1.2 Value iteration

The policy evaluation phase of policy iteration consists, by itself, of many iterations until the value of the current, intermediate policy is computed. It turns out to be possible to simply truncate policy evaluation after one sweep through the whole state set, and then immediately do policy improvement with respect to the imperfect value function thus obtained. By repeatedly doing this, we are again guaranteed to finally end up with the optimal value function and optimal policy.

This algorithm is known as value iteration, and it corresponds to the following backup operation:

$$\begin{aligned} V_{k+1}(s) &= \max_a E\{r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')] \end{aligned} \tag{3.16}$$

for all s. Note that value iteration can be interpreted as simply transforming the Bellman optimality equation (3.11) into an update rule. As with the policy evaluation phase of policy iteration, the process is stopped once the value function changes by only a small amount.

Having understood both policy iteration and value iteration as particular combinations of policy evaluation and policy improvement, it is easy to envision other, intermediate schemes (Sutton & Barto, 1998). For example, we may do some fixed number of policy evaluation sweeps through the state set before doing policy improvement. Or we may do policy evaluation for only a subset of the total number of states before doing policy improvement (asynchronous dynamic programming). This may be advantageous in cases with large numbers of states to reduce the computational load.

3.5.1.3 Backpropagation through a model

A model of the environment does not have to be used to compute value functions, like the dynamic programming techniques described above. It can also be used for direct

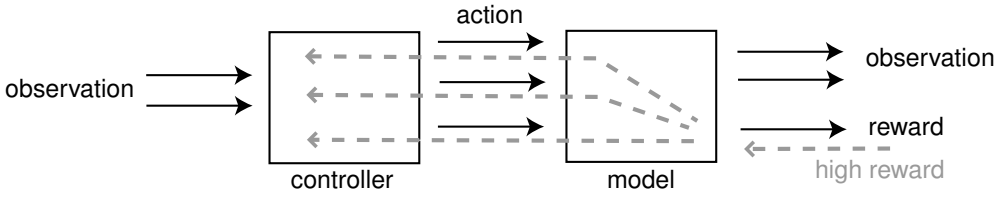

Figure 3.5: Schematic representation of backpropagation through a model. The gray, dotted arrows illustrate how the errors, based on the difference between desired, high rewards and actual rewards, are backpropagated through the model to the controller.

policy search. One way of doing this is by extending the idea of backpropagation through a model (Werbos, 1990; Jordan & Rumelhart, 1992; Kawato, 1990; Nguyen & Widrow, 1990) to the reinforcement learning case (Schmidhuber, 1991c).

Backpropagation through a model is based on the idea that if you have a model, consisting of differentiable functions, that maps inputs to resulting outputs, then it is possible to determine, by backpropagation, how the inputs should change so as to change the outputs in some desired direction. This same principle is used, in backpropagation learning in multilayer neural networks (Werbos, 1974; Rumelhart, Hinton, & Williams, 1986), to determine how hidden layer activations should change so as to change the outputs of the network in some desired reaction. There it is used to determine how to change the weights in the network. In backpropagation through a model, the weights of the model are fixed, and the errors that are backpropagated through the model reflect not prediction errors (the model is assumed to be accurate), but differences between *desired* outcomes in the environment and *actual* outcomes. These errors are backpropagated to the input side of the model, where they represent how the actions should change so as to improve the outcomes in the environment. A separate controller, connected to the input side of the model, can then learn the correct actions on the basis of these errors in a supervised learning way, again using backpropagation (see figure 3.5).

In the supervised learning version of backpropagation through a model (Werbos, 1990; Jordan & Rumelhart, 1992; Kawato, 1990; Nguyen & Widrow, 1990), desired outputs of the model (desired environmental states) are used to determine the errors that are to be backpropagated to the controller. In the reinforcement learning case (Schmidhuber, 1991c), one output of the model is reward (and possibly one more for "pain"). The "desired output" then simply corresponds to "high reward" on this output line.

In the general, delayed reward case, the environment has non-trivial state (see figure 3.1). Therefore, the model of the environment must have internal state. For example, the model may consist of a recurrent neural network (Schmidhuber, 1991c) or time-delay neural network (Jordan & Rumelhart, 1992). This allows the system to compute the influence of past actions on current rewards. The procedure used to accomplish this is the extension of backpropagation to recurrent neural networks,

called backpropagation through time (BPTT, Rumelhart et al., 1986).[3] BPTT through a model effectively propagates errors at the current timestep backwards to inputs of previous timesteps. The controller can then use the errors propagated backwards in time as supervised errors for actions at those previous timesteps, and thus learn the correct actions at those previous timesteps.

The model of the environment must consist of differentiable functions, and in most studies using this technique, both in supervised and reinforcement learning contexts, the model is learned, and it corresponds to a neural network. An elegant property of this approach is that both the model and the controller can be neural networks, and a single procedure, backpropagation, can be used to adjust the weights of the model, to compute the errors for the controller, and to adjust the weights of the controller. A disadvantage, on the other hand, is that this approach can be sensitive to small errors in the model, which may lead to large errors in the signals propagated back to the actions (Jordan & Rumelhart, 1992; Werbos, 1990; Barto, 1990).

The outputs (actions) of the controller, which corresponds to the inputs to the model, can be vectors in the backpropagation through a model approach. The errors backpropagated to the controller are vectors indicating in which direction the action vector should change. For this reason, in contrast to many other approaches, this approach can be used in tasks with high-dimensional, continuous actions. An example of such a task is robot arm control where many motors of the robot arm must be controlled concurrently using real-valued signals.

An interesting variation of the backpropagation through a model approach adds value functions. The "model" of the environment may be trained to output values of state-action pairs (Schmidhuber, 1990; Werbos, 1990), using techniques similar to other value function approaches described in this chapter. Since a state-action value already represents long-term reward as a function of the current state and action, we can use simple backpropagation to the model's input side rather than backpropagation through time. In other words, the value function takes care of temporal credit assignment to actions in the past, rather than backpropagation through time. However, we still have the advantage of being able to backpropagate error vectors toward action vectors. This approach is sometimes called backpropagation through a critic (Werbos, 1990).

3.5.1.4 Learning a model

When the agent is not provided with a model of the environment, it may first *learn* a model of the environment, and then apply any of the methods described above. Learning a model can be done using supervised learning techniques, which simply attempt to learn the mapping from current observations (states) and actions to observations (states) and rewards obtained at the next timestep.

[3] Or one may use a functionally equivalent procedure, such as Real-Time Recurrent Learning (RTRL, Williams & Zipser, 1989; Schmidhuber, 1991c.)

3.5.2 Model-free MDP solution techniques

3.5.2.1 Learning without a model

One approach to the absence of a model is to learn a model, as described above. A different approach is to attempt to learn value functions and/or policies directly, without using a model.

3.5.2.2 Q-learning and Sarsa

Probably the most widely used algorithm for learning value functions without using a model is Q-learning (Watkins, 1989; Watkins & Dayan, 1992). This thesis also uses Q-learning and variations of Q-learning. The basic idea is to incrementally estimate values of state-action pairs, Q-values, based on experienced rewards in the environment and the agent's own estimated Q-values. The update rule of Q-learning in its simplest form is

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)] \tag{3.17}$$

where α is a learning rate parameter. Thus, the Q-value of a state-action pair that the agent has just "visited " in the environment is updated on the basis of the direct reward and the maximally attainable Q-value in the new state. The difference between square brackets is computed based on two consecutive timesteps, and is therefore called the *temporal difference error* (Sutton, 1988). As in dynamic programming, an update of the value of a state-action pair does not have to wait until all subsequent rewards are in, but can be done "locally", based on the value of the next state-action pair.

Interestingly, recent neuroscience findings indicate that certain dopaminergic neurons in the mammal brain, which have long known been known to be involved in learning on the basis of rewards, actually fire correlated with temporal difference errors rather than rewards per se (Schultz et al., 1997). That is, they fire strongly when an *unpredicted* reward comes in, but they fire only at baseline level when a *predicted* reward comes in. On the other hand, when a predicted reward fails to come in, these neurons' firing rate drops below baseline level, indicating a negative temporal difference error. These findings suggest that the mammal brain uses something like temporal difference-based reinforcement learning. In general, it is an excellent example of how computational research may guide empirical research, and how artificial intelligence may lead to better theories about biological intelligence.

Similar to value iteration, Q-learning simply transforms the Bellman optimality equation (3.12) into an update rule. For this reason, Q-learning and similar reinforcement learning algorithms are sometimes called approximate or heuristic dynamic programming techniques (e.g. Watkins, 1989; Werbos, 1992; Bertsekas & Tsitsiklis, 1996). However, unlike value iteration, updating in Q-learning is not done by sweeping through the whole state set and explicitly using the state transition probabilities and expected rewards from a model, but instead by "sampling" the environment. This sampling depends on the current policy and the exploration around the current policy. A possible computational advantage of this sampling technique over sweeping through the whole state set, as happens in value iteration, is that only possible and reasonable

trajectories through the state space are sampled, which may lead to more efficient learning, especially when the state set is large (Sutton & Barto, 1998; Meuleau et al., 1999).

A common exploration strategy is to usually take the action with the highest currently estimated Q-value in that state, but to sometimes take a random action. Convergence to the optimal values $Q^*(s, a)$ is guaranteed when all state-action pairs continue to be updated, when the value of each state-action pair has its own, tabular representation, and when certain standard conditions of decreasing learning rates are fulfilled (Watkins, 1989; Watkins & Dayan, 1992; Sutton & Barto, 1998). Once a sufficiently close approximation to the optimal Q-values has been found, the optimal policy simply consists of taking that action in a state that has the highest Q-value.

Q-learning learns about the optimal policy π^* by following a policy that is not optimal (yet). This is called off-policy learning. One can also modify Q-learning such that the policy that the agent learns about is the same as the one that the agent is following. This is on-policy learning, and a corresponding learning algorithm is called Sarsa (for State-Action-Reward-State-Action), which has the following update rule (Sutton & Barto, 1998):

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]. \tag{3.18}$$

This rule will still converge to the optimal values and optimal policy as long as the policy converges in the limit to the greedy policy, that is the policy that does not do any exploration any more.

Sarsa can have an advantage over Q-learning when the agent must always keep exploring during its lifetime, for instance because it should never stop learning. In that case, we want to learn about that policy that keeps exploring, and not about the hypothetical greedy policy that the agent will never use anyway. Another advantage is related to the use of function approximators instead of tables to represent the Q-function (see below): in some cases Q-learning can diverge due to its off-policy nature, where Sarsa will not. However, usually it seems to make more sense to learn the value of a state based on the estimated best action that the agent can take in the next state (Q-learning), rather than the action that the agent actually takes, which may well be an exploring action and far from the best one (Sarsa).

3.5.2.3 Actor-critic architectures

Historically, the idea of learning values based on temporal difference errors obtained in the interaction with the environment precedes Q-learning and Sarsa. Witten (1977), Barto, Sutton, and Anderson (1983), Sutton (1984), and (Anderson, 1987) proposed architectures where one part of the system, the *critic*, learns state-value functions $V(s)$, and another part, the *actor*, learns the policy $\pi(s)$ on the basis of signals from the critic. Both the actor and the critic learn on the basis of temporal difference errors. The formal theory behind temporal difference (TD) learning was in large part developed in this context, by Sutton (1988).

The critic learns the value of the policy of the actor, for instance using the following temporal difference update rule:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)]. \tag{3.19}$$

The actor explores different actions, for instance with a probability proportional to "preference" $p(s, a)$. When these exploring actions lead to a positive temporal difference error (a positive value between the square brackets, corresponding to a better than expected result), the actor receives this positive temporal difference error as an internal reward. This then makes it more likely that the responsible action is tried again in the same situation. Conversely, a negative temporal difference error makes the responsible action less likely. One update rule accomplishing this can be:

$$p_{t+1}(s_t, a_t) = p_t(s_t, a_t) + \beta[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)] \tag{3.20}$$

where β is another learning rate parameter. Since at the same time the temporal difference error is used to update the critic, the critic will come to expect better results and the actor must do better still to get internal rewards.

For successful performance, the critic should not learn too quickly, for that would prevent the actor from getting sufficient internal reward to learn good actions. On the other hand, the critic should not learn too slowly, otherwise the actor would get too much internal reward for better than chance, but still suboptimal performance, and the actor might get stuck in that suboptimal performance. In practice, getting this balance right has often been found difficult. This is one of the reasons for the subsequent prevalence of Q-learning and Sarsa, which, in a way, combine the actor and the critic in one, simpler system.

Nevertheless, actor-critic architectures have certain advantages, which have caused a recent revival in interest. Like backpropagation through a model, actor-critic systems can be used in cases with high-dimensional and continuous actions. The corresponding action selection is quick; unlike Q-learning and Sarsa, which must search through the possibly large set of state-action values to find the one action with the highest Q-value. Furthermore, actor-critic architectures can learn explicitly stochastic policies, which can be important for certain tasks (e.g. Singh, Jaakkola, & Jordan, 1994). An advantage related to formal reinforcement learning theory is that actor-critic architectures may not have the same convergence problems as Q-learning when combined with function approximators (Sutton et al., 2000). Finally, the actor-critic architecture may be more realistic biologically: (Houk, Adams, & Barto, 1995) propose a model of mammal reinforcement learning based on the actor-critic architecture, tying the various components to circuits in the mammal brain's basal ganglia.

A special variation of the actor-critic architecture is the REINFORCE architecture of Williams (1992). In the delayed reward case, temporal credit assignment is not done using a value function learned by the critic. Instead, the "critic" may, for example, learn the expected *direct* reward at each timestep. The difference between the actually obtained direct reward and the expected direct reward, which can be called the *reinforcement comparison* error (Sutton, 1984), is the internal reward fed to the actor. This internal reward is fed to the actor's weights using yet another variation of backpropagation through time, which in this way takes care of temporal credit assignment. Thus, the actor must consist of differentiable functions and internal feedback loops; e.g. a recurrent neural network. Note the similarities with the backpropagation through a model approach and backpropagation through a critic approach described above.

REINFORCE clearly is a direct policy search method, since it does not use value functions. Some authors (e.g. Murphy, 2000; Sutton et al., 2000) also consider actor-critic architectures in general as direct policy search methods, because of several differences with "clear-cut" value function learning methods such as Q-learning and Sarsa. First, the policy (the actor) has its own parameters, independent of the value function parameters. Second, the policy is not directly derived from the value function (e.g. by taking the maximum over state-action values), but is instead determined by computing the output of the actor. Third, the critic does not have to learn actual values $V(s)$ or $Q(s, a)$. Instead, it may suffice to learn the *relative* values of actions in each state (Sutton et al., 2000).

3.5.2.4 Eligibility traces

The approaches discussed above based on value functions and temporal difference learning perform, in their simplest form, back-ups in the direction of the direct reward plus a discounted value in the next state. This is called the 1-step return, or $R_t^{(1)}$. For instance, in Q-learning it is defined as

$$R_t^{(1)} = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a). \tag{3.21}$$

Now consider a task where rewards are sparse. In the beginning of learning, only state-action pairs just preceding the rewards will have significant updates. The value "contained" in the rewards will only disperse to other state-action pairs very slowly, as the state-action pairs immediately preceding the rewards take on significant value. It is natural to wonder if we may speed up learning by updating state-action values not just on value information from one timestep later, but also from multiple timesteps later. In this way, state-action pairs a number of actions away from a sparse reward may start doing significant value updates right from the start. In other words, we want to base updates not only on the 1-step return, but also on multiple-step returns. For example, the update may be done toward the average of the 1-step return and the 2-step return $R_t^{(2)}$, where $R_t^{(2)}$ is defined as

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \max_a Q_t(s_{t+2}, a). \tag{3.22}$$

Or the update may be done toward the average of the 1-step, 2-step, and 3-step return, etc.

It may seem that the only way to do this updating toward multiple-step returns is to postpone updates until those multiple timesteps later, and in the meantime remember exactly which state-action pairs were visited in which order. However, it turns out that there is a simple, elegant way to do updates toward multiple-step returns that is still local in time. Algorithms doing this are members of the TD(λ) family of algorithms (Sutton, 1988). In the case of Q(λ)-learning (Watkins, 1989; Sutton & Barto, 1998; Wiering & Schmidhuber, 1998), updates are now done towards the λ-return, defined in the following way:

$$R_t^{\lambda} = (1 - \Lambda_{t+1})R_t^{(1)} + \Lambda_{t+1}(1 - \Lambda_{t+2})R_t^{(2)} + \Lambda_{t+1}\Lambda_{t+2}(1 - \Lambda_{t+3})R_t^{(3)} + \dots \tag{3.23}$$

where

$$\Lambda_t = \begin{cases} \lambda & a_t = \arg\max_a Q(s_t, a) \\ 0 & \text{otherwise} \end{cases} \tag{3.24}$$

where $0 \leq \lambda \leq 1$ is a constant that weighs the importance of long-term rewards as opposed to short-term rewards. If $\lambda = 0$ (Q(0)-learning), this reduces to standard Q-learning's 1-step return. If $\lambda = 1$ (Q(1)-learning), on the other hand, the update is done based only on actual rewards obtained during the episode. Thus, in that case estimating Q-values is not based on other estimated Q-values; in other words, there is no *bootstrapping*. In the intermediate cases, $0 < \lambda < 1$, an average of multiple-step returns is taken, weighted in a particular way.

The λ-return is truncated at the point where an exploring action is chosen: $a_t \neq \arg\max_a Q(s_t, a)$. This corresponds to the notion that rewards obtained after that point no longer reflect the value of the currently estimated best policy. After all, in Q-learning we wish to learn about the best policy, and not about a policy with exploring, probably suboptimal actions.

Now, the beautiful property of this approach is that this update of a value toward the λ-return can be accomplished by maintaining a simple scalar measure, called the *eligibility trace*, for each parameter of the reinforcement learning agent. In tabular Q-learning, this corresponds to one eligibility trace per state-action pair. The eligibility trace $e_t(s, a)$ (Watkins, 1989) is then defined as

$$e_t(s, a) = \begin{cases} \gamma\Lambda_t e_{t-1}(s, a) + 1 & s_t = s, a_t = a \\ \gamma\Lambda_t e_{t-1}(s, a) & \text{otherwise.} \end{cases} \tag{3.25}$$

The eligibility trace of a state-action pair can be viewed as information that says to what extent this state-action pair can be held "responsible" for rewards obtained later on. If the state-action pair is not visited, it decays exponentially. If it is visited, it is increased to denote its responsibility for possible later rewards. If an exploring action is taken, $\Lambda_t = 0$: the eligibility traces are reset to 0.[4] $e_0(s, a)$ is also 0 for all state-action pairs. The update rule for Q(λ)-learning, implemented using eligibility traces, then becomes

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha[r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)]e_t(s, a) \tag{3.26}$$

for all s and a. For Sarsa and actor-critic systems we have similar versions of eligibility traces (Sutton, 1988; Sutton & Barto, 1998), and even for Q-learning there are several variations on this version (Peng & Williams, 1996; Sutton & Barto, 1998). It is fairly easy to show that eligibility traces as defined in this way lead, when we consider an entire episode (or lifetime), to value updates toward the λ-return (Sutton, 1988; Watkins, 1989; Sutton & Barto, 1998).

In other words, Q-learning using the λ-return and Q-learning with eligibility traces can be viewed as two equivalent views on the same process. The first view, Q-learning using the λ-return, is called the forward view because from a state this view looks

[4] At least in off-policy methods such as Q-learning. In Sarsa, an on-policy method, the eligibility traces do not have to be reset with an exploring action, because the policy that the agent learns about is the policy actually followed, including the exploring actions.

forward in time for rewards and values in the future. The second view, Q-learning with eligibility traces, is called the backward view because from a state it looks back to previous states and actions to see which ones led to the current situation.

The backward or mechanistic view is important because it has an elegant and straightforward implementation, used throughout this thesis. It allows for learning that is local in time, doing value updates as each new experience comes in. This makes it both desirable as an artificial intelligence tool and increases its relevance for understanding biological intelligence (and as stated in the Introduction chapter, this thesis is interested in both). Furthermore, the backward view can give us an intuitive understanding of how an action in a given state can be held "responsible" for outcomes later in time.

The forward or theoretical view corresponds to a learning algorithm that is not local in time: value updates at a particular time depend on rewards that are available only later. However, it gives a different, more formal perspective that shows what exactly learning with eligibility traces accomplishes. In other words, it shows what error the value function system actually attempts to minimize.

A natural question to ask is why we would use bootstrapping methods, i.e. $\lambda < 1$, at all. After all, non-bootstrapping methods, $\lambda = 1$, can be implemented just as easily with eligibility traces, and they do value updates based on actually obtained rewards rather than on other, estimated values which may be unreliable. The simple answer is that in practice, it turns out that bootstrapping methods often outperform non-bootstrapping methods (Sutton & Barto, 1998). Formal reasons for this are not known. An intuitive reason may be that to the extent that estimated values go some way toward the correct values, they provide useful "subgoal" information, even when they themselves are still some distance away from actual rewards. Sequences of actions reaching such high-value subgoal states could then be learned more efficiently than when the actual rewards must have been reached. After all, between reaching the high-value subgoal state and the actual rewards one still needs to do a number of actions, each of which can be done incorrectly.

3.5.2.5 Evolutionary algorithms

An alternative to value functions is direct policy search. One direct policy search approach uses evolutionary algorithms to search directly in the space of policies (Moriarty, Schultz, & Grefenstette, 1999; Moriarty & Miikkulainen, 1996; Whitley, Dominic, Das, & Anderson, 1993; Yamauchi & Beer, 1994). A *population* of individual policies, encoded by initially random *genomes*, is subjected to "artificial evolution". Individuals that have a higher *fitness* (in this case, performance of the policy) than others have a higher chance to survive and reproduce than individuals with a low fitness. During reproduction genomes are changed in a more or less random way, changing the resulting individuals' policies, and possibly improving fitness, the policy's performance. The idea is that thanks to this process of variation and selection, over time the fitness of individuals in the population will improve, hopefully resulting in good or even optimal policies.

In the simplest case, the genome consists of a string of genes (parameters), each of which represents the action taken in a given state (Moriarty et al., 1999). The fitness

of each genome in the population can simply be the average return R_t obtained by the policy encoded by the genome. The variation during reproduction can be accomplished by applying typical *operators* used in evolutionary algorithms, such as mutation and crossover.

Mutation changes a gene with some small probability, in the above example changing the action to be taken in a given state. Crossover accomplishes recombination of different genomes. In the example where each gene represents the action taken in a given state, this corresponds to combination of different subpolicies into new policies. The idea here is that subpolicies that contribute to good performance in entire policies, and that do that more or less independently from other subpolicies, will become more prevalent in the population. Such successful subpolicies, or "building blocks" in the language of evolutionary algorithms, may be combined with other such successful subpolicies, leading to overall successful policies. This of course depends heavily on these subpolicies having a more or less independent positive contribution to the overall policy's success. In some cases this may be a justifiable assumption. However, in many cases, the success of a subpolicy depends strongly on other subpolicies. Consider an example where there is one subpolicy that takes the agent close to an exit door of a maze, and another which takes the last few actions and goes through the exit door, at which point the agent gets a reward. Each subpolicy's contribution to overall success is heavily dependent on whether the other subpolicy does its job.

Genes do not have to literally represent actions taken in a given state. One alternative is to use the genome to represent the weights or other parameters of a neural network that represents the policy (e.g. Whitley et al., 1993; Yamauchi & Beer, 1994). Again, the success of this approach, at least when one uses recombination operators such as crossover, depends on there being building blocks, in this case individual weights or sets of weights that contribute to high fitness, more or less independently from other weights. Given the distributed nature of neural networks and typically strong interdependence of weights, this is often doubtful. However, the advantage of this approach is that it can be used with arbitrary neural network architectures. A related approach, which to the best of my knowledge is not tested in genuine reinforcement learning environments, is to evolve only the general architecture of the neural network and use another learning algorithm to learn the weights (Whitley, Gruau, & Pyeatt, 1995; Happel & Murre, 1994).

The problems with finding building blocks when evolving neural networks have inspired variations of the evolutionary approach. Moriarty and Miikkulainen (1996) propose an approach based on *symbiosis*, called SANE. In this approach, individuals do not represent complete neural networks (complete policies), but only parts of neural networks. Such a part may be a hidden unit of a feedforward neural network, together with connections from a particular input and output unit and corresponding weights (SANE, Moriarty & Miikkulainen, 1996). In the fitness computation phase, a subset of the individuals is selected stochastically and combined into one neural network. The neural network represents the policy, and the policy is tested in the environment for some time to compute the policy's fitness. This is done a number of times for different combinations of individuals, and each individual's fitness is computed as the average fitness of the different complete neural networks in which they participated. The idea is that different individuals, i.e. different parts of neural networks, will take on different,

complementary roles; and thus will serve as building blocks which can be evolved more or less independently and combined into overall successful neural networks.

3.5.2.6 Classifier systems

Classifier systems (Holland, 1975; Goldberg, 1989; Wilson, 1994; Baum, 1999; Kwee, Hutter, & Schmidhuber, 2001) are a mixture of evolutionary algorithms, which are prototypical direct policy search methods, and a kind of value functions. The agent's policy is represented by a set of *classifiers*, which correspond to rules mapping particular observations to actions. A classifier has an associated *strength*, which has certain similarities to a state-action value. When an observation comes in, the classifiers whose observation conditions match this observation become active. Each active classifier makes a *bid* to be executed which is equivalent to its strength. One of the bidding classifiers is executed, with a probability proportional to its bid. When a direct reward is obtained, the classifier that led to this reward has its strength increased. However, this classifier also "pays" part of its own strength to classifiers that preceded it; those classifiers do the same, etc. This is the way classifier systems attempt to take care of temporal credit assignment.

(Dorigo & Bersini, 1994) showed that in certain restricted cases, classifier systems are equivalent to Q-learning. In recent years classifier systems have moved even more towards other reinforcement learning algorithms, by using Q-learning like update rules (Wilson, 1998). The main difference with algorithms such as Q-learning, then, lies in the way the policy's rules, the classifiers, are modified. Classifiers are modified as in other evolutionary algorithms (see above), by mutation and crossover. Mutation, in this case, randomly changes a classifier's observation condition or its action. Crossover can combine the observation conditions (which correspond to observable "features" in the environment) of different classifiers into a new classifier, as well as change associated actions. Again, to the extent that building blocks can be found in the genome representation, the evolutionary process can develop these building blocks more or less independently and combine them into overall successful classifiers using crossover.

3.5.2.7 Direct policy search based on the success story algorithm and metalearning

An entirely different class of direct policy search methods was developed by Schmidhuber, Zhao, and Wiering (1996), Schmidhuber, Zhao, and Schraudolph (1997), Zhao and Schmidhuber (1998), and Schmidhuber and Zhao (1999). A key idea in this approach is to blur the distinction between actions, internal state transitions, and learning. The policy's "actions" may be "normal", outward actions in the environment, but they may also be actions that change the agent's internal state, or actions that change the policy itself (thus, here we have an instance where the distinction between internal state and parameters becomes unclear). Thus, the system may execute "learning actions" that correspond to policy modifications. What the learning actions amount to, in terms of how and what they actually change internally, can also be part of the policy.

This means that if learning actions change those parts of the policy, they change the learning algorithm itself! In this way, the system may "learn to learn": metalearning.

At certain so-called "checkpoints", the *success story algorithm* is invoked. This algorithm evaluates policy modifications that have been done since the last checkpoint. If these policy modifications have not led to an increase in average reward per timestep, they are undone (this of course requires that policy modifications and previous policies have been stored). Furthermore, the previous checkpoint is removed. This means that now the policy modifications made just before that previous checkpoint are evaluated by the success story algorithm, and they in turn can be removed if they turn out to be bad in light of the new evidence. In this sense, no previous policy modification is sacred. If the success story algorithm reveals that in the long run a policy modification made long ago does not contribute to performance improvement, either because the policy modification was detrimental in terms of long-term reward, or simply because the environment has changed, it can be removed. Interestingly, the decisions when to invoke the success story algorithm, i.e. the checkpoints, can be part of the policy and can be learned too. For example, the system may be able to learn that in very stochastic environments it is better to wait a longer time to the next checkpoint, so as to get a more reliable estimate of the policy improvement.

An important advantage of this approach is that this approach makes very few assumptions about the environment, about the type of policy, or even about the moments at which the policy should be evaluated. This means it can be applied in situations where the assumptions made by other, less general methods are violated. Furthermore, the metalearning capacity has the potential, in principle, to let an agent develop learning algorithms itself that are well-suited to the tasks typically faced by the agent. This amounts to creating learning algorithms that have just the right bias for the task at hand.

3.5.2.8 Exploration issues

All these model-free methods try to improve the policy by deviating from the current policy (temporarily or not) and evaluating the consequences of these deviations. This is the process of exploration. In most cases, small changes are made to the policy, such that not all that has been learned so far is thrown away, but instead exploration helps to improve the current policy. The challenge is to explore sufficiently but not too much.

One simple and common exploration strategy is to normally choose the action that currently seems best in a given state ("greedy" action selection), but to choose a random different action with some small probability ϵ. This is called ϵ-greedy exploration. Another common, slightly more sophisticated strategy is to make the probabilities of different actions dependent on their computed "preferences" or values in the given state, e.g. using the Boltzmann (or Gibbs) distribution. In Q-learning, for instance, the probability $p(s, a)$ of each action a in state s is then computed according to

$$p(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{m=1}^{M} e^{Q(s,a_m)/\tau}} \tag{3.27}$$

where M is the number of possible actions and τ is the so-called temperature parameter. The higher the temperature, the flatter this distribution, and the more exploration the method does. In the limit, as $\tau \to 0$, action selection becomes greedy. Even with constant τ, as learning proceeds and the Q-values become increasingly different from each other, exploration gradually decreases and exploitation of the learned policy gradually increases. Boltzmann exploration and variations are used throughout this thesis.

The amount of exploration may also be decreased explicitly with time. For instance, ϵ in ϵ-greedy exploration may gradually be made smaller, or τ in Boltzmann exploration may gradually be made smaller. This reflects the intuition that the learned policy will likely improve significantly over time, such that much exploration is no longer necessary. Furthermore, decreasing exploration with time is usually necessary for proofs of convergence of reinforcement learning algorithms.

More sophisticated still is to let the amount of exploration depend on statistics associated with states or actions. The idea behind this is that you want to focus exploration on those states and actions about which you are uncertain or that are somewhat promising with respect to reward. Exploration mechanisms that do this do *directed* exploration, as opposed to the *undirected* exploration techniques described above.

For example, exploration of an action may depend on how often an action has been tried out in the same situation in the past (Sutton, 1990; de Jong, 1997). Or exploration may depend on a confidence interval associated with the expected returns from a state, a technique known as exploration based on Interval Estimation (Kaelbling, 1990). The directed exploration mechanism may itself be a learning method, in which case it is called *adaptive* exploration. For instance, it may learn to predict the next environmental state, and correlate exploration with the prediction error (Schmidhuber, 1991b; Thrun & Möller, 1992).

3.5.2.9 Generalization

In most realistic tasks, there are many states and possibly many actions. States and actions may even be continuous. In these cases, the agent cannot possibly gain sufficient experience with every single state-action pair in a limited amount of time. We want the agent to *generalize*, that is, to use its experience with a limited number of experienced states and actions in new states and actions. In most cases, generalization over states boils down to assuming that states that *look* similar, i.e. in terms of observations, are often similar in value. Likewise, generalization over actions usually assumes that similar actions (say, similar action vectors) have similar effects. Generalization over actions has not been studied extensively in the reinforcement learning literature, but since it presumes high-dimensional and/or continuous action vectors, actor-critic architectures (Gullapalli, 1991) or backpropagation through a model (Werbos, 1990) are more natural choices than, for instance, Q-learning and Sarsa (but see Baird & Klopf, 1993). In the remainder of this chapter and this thesis, we will limit ourselves to generalization over states, and not consider generalization over actions.

Of course, generalization is not a problem only for reinforcement learning. It has been studied extensively in the context of supervised and unsupervised learning. In

fact, many of the approaches to generalization in reinforcement learning simply apply methods from supervised or unsupervised learning to reinforcement learning.

A simple, classical approach is to discretize the state space into (hyper-)cubes of equal size, so-called BOXES, and treat all observations within a box as a single state (Michie & Chambers, 1968), after which we can apply a standard reinforcement learning algorithm based on limited sets of states. This is a *state aggregation* method. Classifier systems use a variation of state aggregation: the classifier's observation condition, which checks a number of observation features, can contain so-called "wildcards", meaning that the associated observation features can take on any value and still satisfy the observation condition.

State aggregation methods decompose the state space into discrete regions, usually with boundaries perpendicular to the observation dimensions. Each discrete region corresponds to a separate discrete state. For this reason, state aggregation methods cannot learn a *smooth* function over the state space, whereas in many cases such a smooth function may be preferable. For instance, it will in many problems be reasonable to assume that a state intermediate between two other states has an intermediate value, rather than the exact same value as either one or the other state.

An approach that can learn such smooth functions over the state space, studied by many researchers, uses some kind of parameterized function approximator to approximate the policy or value function. Usually, the function approximator's parameters w_i are learned using a *gradient descent* method. This is also the approach taken in this thesis. The input to the function approximator is an observation vector (and possibly an action vector), and the output represents policy actions, state-action values (Q-learning, Sarsa), or state values (critics).

The function approximator will generalize to new, unseen states. However, there is no guarantee that the assumption is correct that similar-looking states have similar values or should lead to the same actions. Furthermore, function approximators can sometimes generalize in unpredictable ways, and the more powerful and complex ones can get stuck in local optima with gradient descent learning. For these reasons, there is rarely a guarantee of finding the optimal policy or the optimal value function over the entire state space. Nevertheless, in many cases "satisficing" performance can be achieved in these cases where exact solutions are out of the question anyway. Bertsekas and Tsitsiklis (1996) discuss the issues of reinforcement learning combined with function approximation at length, especially from a formal perspective.

In the case of Q(0)-learning with a function approximator trained using gradient descent, the update for each parameter w_i of the function approximator becomes

$$w_{i,t+1} = w_{i,t} + \alpha[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)] \frac{\partial Q_t(s_t, a_t)}{\partial w_{i,t}} \tag{3.28}$$

where α is again a learning rate parameter. If the function approximator has learned a sufficiently close approximation to the optimal values $Q^*(s, a)$, we can simply compute, using the function approximator, $Q(s, a)$ for each of the possible actions in the current state and greedily select the action that has the highest $Q(s, a)$. Note that standard table-based Q-learning can be viewed as a special case of this equation, if we consider the Q-values stored in the table as the parameters w_i.

What $\frac{\partial Q_t(s_t,a_t)}{\partial w_{i,t}}$ looks like and the complexity of its computation depend of course on the function approximator. Simple linear function approximators have been used, where the value function is just a weighted sum of independent observation features, all of which can be either "on" or "off" (Sutton & Barto, 1998). Somewhat more complex variations of linear function approximators use overlapping tilings ("receptive fields") covering the state space, such that each point in the state space gives rise to a distributed, "coarse coded" feature representation. To this class belong CMACs, e.g. Watkins (1989), Sutton (1996), Kretchmar and Anderson (1997), and RBFs, e.g. Kretchmar and Anderson (1997), Acharyya (2000)).

Just as in supervised learning, there is a limit to the complexity of the function associated with the reinforcement learning algorithm that linear functions can represent. For this reason, many authors have used more complex function approximators, such as multilayer feedforward neural networks. Multilayer feedforward neural networks have been used to represent the value function of Q-learning (Lin, 1992; Anderson, 1993; Crites & Barto, 1996; Humphrys, 1997; Abul, Polat, & Alhaji, 2000; ten Hagen & Kröse, 1998), the value function of Sarsa (Rummery & Niranjan, 1994), the state value function of a critic (Tesauro, 1992, 1994), and to represent both actor and critic (Barto et al., 1983; Anderson, 1987). Tesauro's (1992, 1994) work, a backgammon player known as TD-gammon, is one of the most impressive and well-known applications in reinforcement learning. TD-gammon consists of a multilayer feedforward neural network whose input is a representation of the board position and whose output represents the value of that board position. A function approximator is needed because of the huge number of possible board positions in backgammon. The value is learned with standard TD(λ)-learning, using experience gathered by playing many games against itself. In this way, TD-gammon learned to play backgammon at a world class level.

As discussed above, neural networks have also been used as the policy representation in direct policy search methods, such as backpropagation through a model (Schmidhuber, 1991c), and in certain approaches based on evolutionary algorithms (Whitley et al., 1993; Yamauchi & Beer, 1994; Moriarty & Miikkulainen, 1996). In those direct policy search methods, too, one of the main reasons for turning to neural networks was generalization.

Even with simple, linear function approximation, it is possible to construct reinforcement learning tasks where TD-learning and Q-learning diverge (Baird, 1995; Tsitsiklis & Roy, 1997; Boyan & Moore, 1995). That is not particularly promising with respect to value-function based reinforcement learning with function approximators in general. However, as of yet it is completely unclear how serious the convergence problems are in practice. It may be that the constructed problematic tasks are not typical for most reinforcement learning tasks, or that different or more sophisticated types of reinforcement learning algorithms or function approximators do not suffer as much from these problems (Baird, 1995; Sutton & Barto, 1998; Sutton et al., 2000; Anderson, 2000). In any case, these are important issues to resolve if one wants to use reinforcement learning reliably in large tasks.

It is also possible to use eligibility traces with function approximators, for the same reasons as in the discrete, table-based case. We then need a separate eligibility trace e_i for each parameter w_i (Sutton, 1989). For instance, a parameter update for

Q(λ)-learning using a function approximator trained by gradient descent is

$$w_{i,t+1} = w_{i,t} + \alpha[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)]e_{i,t} \tag{3.29}$$

where

$$e_{i,t} = \gamma\Lambda_t e_{i,t-1} + \frac{\partial Q_t(s_t, a_t)}{\partial w_{i,t-1}}. \tag{3.30}$$

Thus, in this case the eligibility trace e_i says to what extent the corresponding parameter w_i is responsible for possible rewards later on. The function approximator still does gradient descent, but the error surface is now defined by the λ-return rather than the 1-step return.

Ideally, we want strong generalization in areas of the state space where that is possible, while making finer distinctions between states where that is necessary. Methods designed to accomplish that are called *adaptive resolution* methods. A class of adaptive resolution methods based on state aggregation, proposed by several researchers (Chapman & Kaelbling, 1991; McCallum, 1995; de Jong, 1999), keeps statistics about returns obtained from different state-action pairs. These methods start out by assuming that the whole state space, perceived by the agent through an observation vector of multiple features, corresponds to a single state. If the statistics, gathered during exploration, indicate that there are significant differences in returns for different regions within a single observation dimension, this observation dimension is *split* into different regions, thus making a finer distinction in the state space. Because splitting is done on the basis of differences in returns, these methods obtain a higher resolution in areas of the state space where that is "important".

A disadvantage of these methods is that they only consider single dimensions at a time (but see McCallum, 1995 for a very limited extension). Therefore, they have trouble making distinctions when different observation dimensions need to be considered in combination, such as parity problems (of which XOR is the simplest case). Note that multilayer feedforward neural networks, in contrast, *can* learn to make such distinctions (see Rumelhart et al., 1986, and chapter 4). In fact, multilayer feedforward neural networks trained with gradient descent can also be viewed as adaptive resolution methods. After all, they form decision boundaries in their input space based on the errors in the outputs. Thus, they will attempt to form "useful" decision boundaries (without a guarantee of optimality), treating areas of the input space as equivalent where possible (low errors), while making finer distinctions in the input space where necessary (high errors). This is reflected in the hidden unit representation, which typically has more units dedicated to areas of the input space that are "important" in this sense (Rumelhart et al., 1986). In the reinforcement learning case, similar states which can safely be generalized over will lead to small errors (e.g. small temporal difference errors) in the neural network, whereas similar states which cannot be generalized over will lead to large errors. Therefore, the neural network that is used as the reinforcement learning algorithm's function approximator will likely realize adaptive resolution implicitly. This is one of the reasons why this thesis uses neural networks.

Rosen, Goodwin, and Vidal (1991) and Kröse and van Dam (1992) present a variation of this idea, based on Kohonen neural networks (Kohonen, 1995) rather than

perceptron-like feedforward neural networks. In these studies, input vectors are classified onto winning neurons in the Kohonen network. The classification in the Kohonen network is used as the input to an actor-critic system. As is normally the case in Kohonen networks, in addition to moving the weight vector of the winning neuron in the direction of the input vector, the weight vectors of neighboring units also move toward the same input vector. However, in these particular studies, this effect is larger with larger temporal difference errors. This results in more neurons in the Kohonen map being assigned to those states that are important in the sense that they are associated with large temporal difference errors, in this way accomplishing adaptive resolution in the state space.

3.6 POMDP solution techniques

3.6.1 Internal state

In MDPs the agent's observation is equivalent with the environment's state. Therefore, all MDP solution techniques discussed above learn a policy which maps observed states to actions. In a POMDP, such a memoryless or perception-based policy will in general not suffice, and the agent must learn an internal state-based policy. Again, approaches doing that can be divided into model-based approaches, where the state transition function, the reward function, and the non-trivial observation function are known, and model-free approaches.

3.6.2 Model-based POMDP solution techniques

3.6.2.1 Exact POMDP solutions based on belief states and PWLC value functions

If an exact model of the environment is available, an exact solution to POMDPs can, in theory at least, be computed. Like dynamic programming techniques for MDPs, the model-based exact POMDP solution techniques consider the problem not so much as a learning problem—there is no trial and error, and no exploration/exploitation dilemma—but rather as a (probabilistic) planning problem. The objective is to compute a plan, formulated as a policy saying what to do in what situation. This computation can be divided into two elements: belief state computation and value function computation based on belief states.

The belief state component outputs a belief state, a (Moore) internal state which says what state the agent believes it is in. It does not just indicate the most likely state, but rather the entire probability distribution over all possible environmental states. Thus, the agent's uncertainty is explicitly represented and taken into account. The belief state is computed based on the previous belief state, the action and the observation, and the model's state transition and observation probabilities, using Bayesian statistics. Given an old belief state b, an action a, and a new observation o, the new

belief of being in state s', $b'(s')$, is computed according to

$$\begin{aligned} b'(s') &= \Pr(s' \mid o, a, b) \\ &= \frac{\mathcal{P}_o^{s'} \sum_{s \in S} \mathcal{P}_{ss'}^a b(s)}{\sum_{s'' \in S} \mathcal{P}_o^{s''} \sum_{s \in S} \mathcal{P}_{ss''}^a b(s)}. \end{aligned} \tag{3.31}$$

The belief of being in a state s' is updated proportionally to the probability of seeing the current observation given the state s' and to the probability of arriving in this state s' given the action and our previous belief, probabilities which are given by the model. For example, consider a robot application where the robot is initially completely uncertain about its location. Seeing a door may, as specified by the model's $\mathcal{P}_o^{s'}$, occur in three different locations. All probability mass then collapses into these three states. Suppose that the agent takes an action, and now observes a T-junction. It may be that given this action, only one of the three possible states can lead to a new state in which a T-junction can be observed. We now know with certainty which state the agent is in. In this example the uncertainty in the belief state completely disappeared after a number of observations and actions. In general however, the uncertainty may increase or decrease, depending in large part on how specific the incoming observations are.

The belief state is a *sufficient statistic*, which means that we cannot do better even if we remembered the entire history of observations and actions. This implies that we have now transformed the POMDP into an MDP. However, it is an MDP with a continuous state space, because the belief state is a vector of continuous probabilities.

The belief state is the input to the second component of the method, the value function computation. The belief state is a point in a continuous space of $N-1$ dimensions (the probability of the last state N is determined by the probabilities of the other states). Thus, the value function must be defined over this $N-1$ dimensional continuous space. This renders direct application of dynamic programming techniques, which assume discrete sets of states, impossible. However, it is known that the value function is subject to various constraints, which can be exploited by solution methods. Specifically, the value function either is or can be approximated arbitrary closely by a piecewise linear and convex (PWLC) function (Kaelbling, Littman, & Cassandra, 1998). Figure 3.6 illustrates this in a 1-dimensional belief state space (2 states).

The convex nature makes some intuitive sense if one considers that being *completely* certain that the agent is in a particular state should always have a higher value than being *fairly* certain that the agent is in that state. After all, you can choose more appropriate actions if you are certain about what situation you are currently in. For this reason, actions in the optimal policy can sometimes be directed solely toward decreasing the uncertainty. This means that the optimal agent may sometimes take an information-gathering action: look around for a certain landmark, for instance. An elegant property of this approach is that such information-gathering actions are not distinguished from conventional actions; both are associated with values and can be incorporated in the optimal policy.

Now the value function defined over the continuous belief state space can have a finite representation, by representing each of the linear segments of the PWLC function. The PWLC value function depends on the number of actions one wants to take, or, in planning terms, the horizon of the planning process. If the agent has

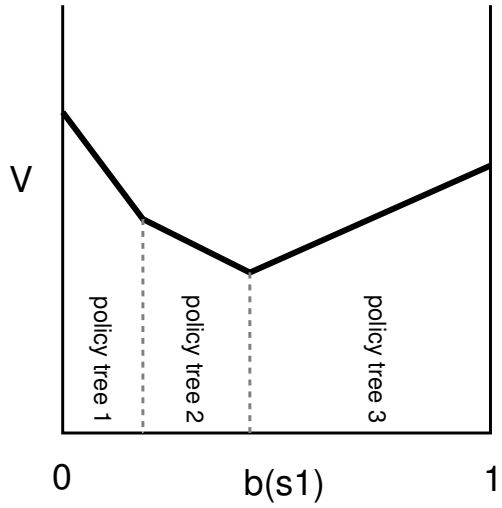

Figure 3.6: Example of a PWLC value function in a 1-dimensional belief state space (2 states). Each linear segment corresponds to a policy tree which is optimal for that part of the belief state space.

to take only one action, the value depends only on the direct reward that can be obtained after this action. If the agent takes two actions, the value depends also on rewards obtainable from the next state, etc. Using a variation of value iteration, the horizon 2 value function can actually be computed on the basis of the horizon 1 value function, etc. In this way, the PWLC value function for a certain desired horizon can be computed iteratively.

Each of the linear segments in the PWLC value function can be understood as corresponding to a "policy tree", which determines the first action to be taken given that the agent is in this segment of the belief state space, as well as the actions to be taken next, *depending on the different possible observations that can occur at the next timesteps*, up until the horizon of the planning process. This gives some intuition for the most important problem with this exact solution method: the number of linear segments in the optimal PWLC value function may grow superexponentially with the horizon of the planning process! However, in many cases a large number of possible policy trees are completely dominated by others. In terms of the PWLC value function: for many possible linear segments there are no points in the belief state space where their values are the highest of all linear segments. Therefore, their corresponding policy trees can be pruned away. Modern exact solution techniques (Cheng, 1988; Cassandra, Littman, & Zhang, 1997) boil down to cleverly generating only policy trees that have some hope of surviving in the final value function and cleverly pruning away hopeless ones. Still, for problems larger than very small ones (consisting of only a handful of states, observations, and actions), these exact solution methods are not practical.

3.6.2.2 Approximations based on belief states

The main bottleneck in computing exact optimal solutions for POMDPs (given a model) lies in the computation of the horizon-n value function, and not so much in the belief state computation. Several authors have therefore suggested to keep the belief state computation element, and to approximate the value function heuristically.

One approach is to truncate the value iteration process long before actual convergence. The idea is that in many cases a near-optimal policy may already be derived from a suboptimal value function (Littman, Cassandra, & Kaelbling, 1995a, 1995b).

Another approach is to first solve the POMDP's underlying MDP, using standard dynamic programming techniques. This is much more feasible than solving the full POMDP, in terms of computation time. The Q-value for each action a in a belief state b can then be estimated as a simple weighted sum of the MDP's Q-value:

$$Q(b,a) = \sum_{s \in S} b(s) Q_{\text{MDP}}(s,a). \tag{3.32}$$

One problem with this approach is that one loses the advantage (of exact solution methods) that optimal actions may sometimes be directed towards gathering information and decreasing uncertainty about the current state. In the underlying MDP, such information gathering is never necessary, so a corresponding action can never get a value because of that. In other words, the optimal value function of the POMDP's underlying MDP can deviate in important ways from the optimal value function of the POMDP. For the same reason, using the value function of the underlying MDP can lead to situations where the agent's actions causes it to get stuck in a loop (Littman et al., 1995a).

Yet another approach is to combine belief states with a variation of Q-learning, which is a model-free technique originally developed for MDPs. This approach basically treats the belief state as a distributed feature representation of the environment's state, very similar to what researchers have done in model-free reinforcement learning aimed at generalization over states (see above). Two types of simple, linear value function approximations have been tried, where each state has an associated parameter, and the parameters are trained using gradient descent-like procedures (Chrisman, 1992; McCallum, 1993; Littman et al., 1995a). However, while this linear approach may work in many cases, it can also lead to arbitrarily poor performance, because its representation of the value function can never be sufficiently complex to accurately approximate the optimal PWLC value function (Littman et al., 1995a; Hauskrecht, 2000).

Even in cases where the environment has many or continuous states, such as robot domains, belief state estimation can still be done relatively reliably and efficiently, such that it can be combined with value function approximation. An interesting recent technique in this context is *particle filtering* (a.k.a. condensation algorithms, Monte Carlo localization). This technique (e.g. Dellaert, Fox, Burgard, & Thrun, 1999; Thrun, 2000; Vlassis, Terwijn, & Kröse, 2002) distributes a limited set or population of particles across the continuous environmental state space. Each particle has an associated measure of likelihood given the action and new observation, called the importance factor, which is updated using a variation of the Bayesian rule described

above. At the next timestep, a new population of particles is computed based on the previous population. However, the probability of particles being incorporated in the new population is proportional to their importance factors (note the similarity to evolutionary algorithms). In this way, the density of particles comes to represent the probability distribution of states: an estimated belief state. This estimated belief state can subsequently be used for value function approximation (Thrun, 2000).

3.6.2.3 Backpropagation through a model

The backpropagation through a model approach, described above for MDPs, can be applied to POMDPs with relatively few conceptual changes (Schmidhuber, 1991c). However, the learning task for the controller is now significantly more difficult. This difficulty stems from the fact that the state is not given now. Therefore, the controller can no longer simply learn a mapping from observations to actions. Instead, it must learn a mapping from *inferred* states to actions.

It is natural to use the state as inferred by the model as input to the controller (see Schmidhuber, 1992a for a similar idea in a different context). The controller can then learn the mapping from the inferred state to actions, again based on errors backpropagated through the model. Or the controller can infer the environmental state itself, if it is also a recurrent neural network. These ideas carry through to backpropagation through a critic (Schmidhuber, 1990).

3.6.2.4 Learning a model

As in MDPs, when the model of the environment is not given, the agent can learn a model and then apply any of the methods described above. In POMDPs, this learning task is significantly harder, however. The model can no longer learn a mapping from current state (observation) and action to next state (observation) and reward. Instead, it must learn, either explicitly or implicitly, to infer the state on the basis of the sequence of observations and actions, and use the inferred state as the basis for its predictions of the next observation and reward.

One example of this strategy is the work of Schmidhuber (1991c), described in the previous section on backpropagation through a model. The model is learned using supervised learning of a fully recurrent neural network. The network is trained to predict the next observation and reward (and pain), and learns to infer the environment's state implicitly, using BPTT or RTRL.

In Lin and Mitchell (1993), a model is also learned using supervised learning of a recurrent neural network, in this case an Elman network. However, in this work the recurrent neural network model is not used for backpropagation through a model. Instead, the model's internal state, which represents the environmental state, is input to a separate feedforward neural network trained to approximate the Q-function using Q-learning.

Another example of the strategy of learning a model has also been mentioned before: Chrisman's (1992) Perceptual Distinctions method learns a model of the environment using a variation of Hidden Markov Modeling (HMM) techniques. The system starts out using a model with two states, and adds states so as to improve its

predictions of next observations and rewards. A probability distribution over these states is maintained: a belief state. A linear Q-learning rule is applied to the belief state (see section 3.6.2.2).

McCallum's (1993) Utile Distinction Memory also attempts to learn a model of the environment using a variation of HMM techniques. However, the important variation here is that states are added only if that helps to predict the *expected return*, rather than next observations and rewards. Thus, no attempt is made to learn a full-blown predictive model independent of expected returns. Instead, the model is "utility-based": states are only distinguished if that is relevant with respect to returns. In a way, this is therefore an intermediate method between model-based approaches and model-free approaches. The algorithm that is used to accomplish this basically looks at statistics on expected returns maintained at the incoming transitions of each model state. If the statistics indicate significantly different returns on the incoming transitions of a model state, this means that this particular model state cannot correspond to just one environmental state, and the model state is split. Similar to Chrisman (1992), something like a belief state, i.e. a probability distribution over the model states, is computed, and linear Q-learning is applied to this belief state.

3.6.3 Model-free POMDP solution techniques

3.6.3.1 Learning without a model

As in MDPs, when the model of the environment is not given, the agent can either learn a model, or attempt to learn a satisficing policy without learning a model.

This section discusses approaches that follow the latter strategy when no model is given: they attempt to learn internal state-based policies without learning a predictive model of the environment. In most of these approaches, the basic idea is that it may be possible to infer (something like) the environmental state directly from the experienced sequence of observations, actions, and rewards, without a model. What the agent must effectively learn (or have built in) is some kind of *algorithm* that stores relevant information from the experienced sequence in a short-term memory. If this algorithm functions properly, this information constitutes a (Mealy) internal state which together with the current observation yields a Markovian environmental state signal. This Markovian environmental state signal can, as we know, directly be mapped to actions to obtain an optimal policy, using model-free learning techniques.

3.6.3.2 Reactive policies

Many model-free approaches to the problem of partial observability use internal state, but another approach to the problem is to simply ignore it. In some cases, a reactive policy mapping observations to actions can have satisficing performance (Littman, 1994; Loch & Singh, 1998). However, this approach can also have arbitrarily bad performance, when two states associated with the same observation require very different actions. It can also easily cause the agent to get stuck in a loop, for instance when one of the perceptually aliased states leads to a dead end when using the action that is optimal in the other perceptually aliased state.

The latter problem can in some cases be alleviated by using explicitly *stochastic* reactive policies rather than deterministic ones (Singh et al., 1994). In perceptually aliased states the agent can then learn to take one action with some probability, and another action with the remaining probability, thus providing a mechanism to get out of loops. However, performance can still be arbitrarily bad, for instance when one of the stochastically chosen actions leads to severe penalties in one of the perceptually aliased states.

3.6.3.3 Finite history windows

Possibly the simplest model-free approach which directly uses the experienced history to create an internal state is the approach that adds to the agent's normal observation a *finite history window* of the n most recent observations and actions. This is a well-known approach in supervised learning and control theory, where it is sometimes referred to as a tapped delay line approach. If the delay lines contain only past observations, Williams (1990) calls this a "conservative" approach. If the delay lines contain past observations and past actions, Williams (1990) calls this a "liberal" approach.

In reinforcement learning, using a finite history window corresponds to assuming that the POMDP can be modeled as an n-order MDP, a POMDP in which the state is always disambiguated by looking at the n previous observations and/or actions. The finite history window approach has been studied by (Lin & Mitchell, 1993, 1992), who used multilayer feedforward neural networks with delay lines to approximate Q-learning's value function. In cases where the POMDP can be viewed as an n-order MDP with limited n, this approach is simple and effective. With larger n, that is, longer-term dependencies between past events and current predictions or actions, this approach quickly becomes infeasible, however. The difficulty lies not only in determining the proper n, but also in the rapidly increasing number of weights (in the case of neural networks) associated with all extra input lines. The problem of long-term dependencies is more extensively discussed in chapter 6.

It is easy to construct POMDPs which can never be solved using a finite history window. Consider a task (Williams, 1990) where the agent first sees a symbol, say either A or B, then has to wait for an arbitrary amount of time, until it sees another symbol, Z. At this point, the agent has to do action 1 if it has previously seen an A to obtain a reward, and action 2 if it has previously seen a B. Effectively, the agent has to implement a flip-flop, setting one bit of short-term memory to either 0 or 1, ignore all irrelevant events, and use its short-term memory when it sees Z. Even though this task requires only one bit of short-term memory, it can never be solved by a finite history window approach, because the time between A/B and Z can be arbitrarily long. In the terminology of Williams (1990), this task has *strongly hidden state*, as opposed to the *weakly visible state* which can be solved using a finite history window. Strongly hidden state requires a "radical approach" (Williams, 1990), where the algorithm that controls the internal state is more complex than any history window approach can provide.

3.6.3.4 Tree-based variable depth history windows

A more sophisticated variation of the finite history window approach is to use a *variable* depth history window (Utile Suffix Memory, McCallum, 1995, U-tree, McCallum, 1996), which allows the history window's depth to be different in different parts of the environmental state space.

These algorithms basically extend Chapman and Kaelbling's (1991) adaptive resolution method, discussed above in the context of generalization (section 3.5.2.9), from the spatial to the temporal domain. The system starts out with zero history. Like Chapman and Kaelbling's (1991) adaptive resolution method, statistics are gathered on expected returns for certain observation dimensions hypothesized to be relevant for the state representation. However, now this concerns an observation dimension (or action) from one timestep ago, rather than the current timestep. If, within that observation dimension from one timestep ago, different regions lead to significantly different returns, that dimension is split, and this distinction becomes part of the state representation. The system can build on the 1-step history, and subsequently look at observation dimensions or actions 2 timesteps ago, to determine if splitting them is useful, etc. In this way, a "history tree" can be built which has variable depth in different dimensions of the observation and action vectors.

Note that this method suffers from a similar problem as Chapman and Kaelbling's (1991) adaptive resolution method: it cannot easily discover *combinations* of feature distinctions that may be useful for predicting returns. Thus, *temporal* parity problems would be difficult for this method. However, (McCallum, 1995, 1996) solve this problem to a limited extent by allowing the depth of the variable history window to increase by more than one step at a time (this does, of course, have repercussions for the amount of computation needed to investigate hypothesized distinctions).

U-tree (McCallum, 1996) also realizes adaptive resolution in the spatial domain, on the current observation. The idea is that the agent can "have both too much sensory data and too little sensory data at the same time" (McCallum, 1996). Too much sensory data is handled by generalization over the current observation vector, too little sensory data is handled by using the history of observations and actions. The elegant thing is that both tasks, often seen as opposite problems, are performed by one and the same adaptive resolution mechanism, and a single, "useful" state representation comes out of this mechanism. This state representation can be associated with Q-values, and the Q-values can be learned using Q-learning or, in McCallum (1995, 1996), value iteration[5].

3.6.3.5 Nearest Sequence Memory

Another variation of the history window approach is the Nearest Sequence Memory (NSM, McCallum, 1997). The basic idea of disambiguating states using the history of observations and actions is the same. However, in NSM all the experiences of the agent

[5]Value iteration is a model-based method, but the variable depth history window approach is discussed in the context of model-free methods because it is so closely related to fixed size history window approaches, and as with (McCallum, 1993), this method does not lead to a complete predictive model of the environment but only to state representations distinguishable in terms of expected returns.

are explicitly stored, and whenever the agent is about to take an action, it looks in the stored experiences for histories of observations and actions that are most similar to the currently experienced history of most recent observations and actions. Those similar histories are considered the k nearest neighbors. The current state's Q-values are simply the averaged Q-values of the k nearest neighbors. As in regular Q-learning, the agent simply chooses the action with the highest Q-value as computed in this way (or it explores with some probability). Regular Q-learning is used to update the Q-values of the nearest neighbors that "voted" for the chosen action. Note how this algorithm can be considered the POMDP equivalent of the k-nearest neighbor algorithm studied in pattern recognition.

Despite its simplicity, this algorithm can work surprisingly well and learn quickly. It is not very robust in the face of noise, however (McCallum, 1997). Of course it also has large memory requirements if the number of experiences keeps growing. Furthermore, note that it will have difficulties if the last n observation-action pairs are irrelevant or even misleading with respect to the currently best action, but the observation-action pair $n+1$ timesteps ago *is* relevant (see chapters 6 and 7 for examples of such tasks). NSM can also be sensitive to sample densities (McCallum, 1997), i.e. the prevalence of certain arbitrary experiences during exploration. For instance, if the agent has experienced early on that a goal can be reached using some detour, it may tend to keep using that detour, because its previous experiences were like that. However, other model-free methods may similarly get stuck in such suboptimal, but satisficing behavior.

3.6.3.6 HQ-learning

In certain restricted cases, a POMDP can be hierarchically decomposed into a set of "pseudo-Markovian" tasks, each of which can be solved by a separate reactive policy mapping observations to actions (Wiering & Schmidhuber, 1997).

For example, suppose the task is to navigate to a certain goal position. It may be that this is impossible for a pure reactive policy, because it requires the agent to turn right at the first T-junction and left at the second T-junction, and the T-junctions cannot be distinguished on the basis of how they look (a POMDP). However, it may be possible to have a combination of *two* reactive policies that does solve the task: one that leads to the first T-junction and then turns right, and one that subsequently takes over and turns left at the second T-junction and proceeds to the goal. Hierarchical Q-learning (HQ-learning) can learn such combinations of reactive subpolicies (Wiering & Schmidhuber, 1997).

The number of reactive subpolicies is fixed a priori, and subpolicies are executed strictly sequentially. The elegant property of HQ-learning is that it can identify the points at which there should be a switch to the next subpolicy, and make that point's associated observation a subgoal for the previous subpolicy. This is done by learning for each subpolicy a table of so-called HQ-values, which represent the "value" of certain observations as a subgoal. An observation's HQ-value is updated based on this subpolicy's own expected return plus the maximum HQ-value from the next subpolicy, if it was invoked after this observation was reached. In this way, an observation functioning as a subgoal that leads to good performance, because of rewards collected

by the current subpolicy itself or because of good performance by successor subpolicies, will get a high HQ-value and will be strengthened as a subgoal. The subpolicy associated with this subgoal can learn its reactive task as normally, using Q-learning. A higher-level controller passes control from one reactive subpolicy to the next when a subgoal is reached. The only internal state of the system resides in the higher-level controller, and basically corresponds to a pointer which says which subpolicy currently has control.

3.6.3.7 Memory bits

Various model-free MDP solution methods can be applied to POMDPs by extending those methods with *memory bits*. Memory bits constitute the (Mealy) internal state of the agent. They can be switched on (1) and off (0) by "internal" actions. These internal actions are part of the total action set, they are not explicitly distinguished from normal, outward actions. Thus, the agent can learn the internal actions in the same way as it learns the outward actions. The memory bits are treated as part of the observation. In this way, the internal state, modified by internal actions, can, in turn, be used to determine new outward and internal actions.

The internal actions can literally be separate actions in the action set, e.g. "set memory bit 1 to 0". Alternatively, they may always be part of a combination with outward actions. The actions then become something like "go left, and set memory bit 1 to 0". An advantage of the latter strategy is that the number of actions needed to reach a goal is smaller, leading to shorter temporal credit assignment paths. For this reason, most authors have used the latter strategy, even though it does increase the action set by a greater amount than the first strategy.

Because increasing the number of memory bits quickly increases the size of both the action set and the observation set, most authors have used only one or a few memory bits. This limits the applicability of this technique to simple problems which require only one or a few bits of short-term memory. However, in contrast to history window approaches, it can be used in tasks with strongly hidden state. Furthermore, it is a natural and easily implementable extension of a number of model-free MDP solution techniques that are relatively well understood.

One such technique is tabular Q-learning. Each additional memory bit doubles the number of observations in the table. (Littman, 1993, 1994; Peshkin, Meuleau, & Kaelbling, 1999; Lanzi, 2000) have shown that tabular Q-learning extended with memory bits in this way can solve various non-Markovian tasks. Lanzi (2000) identifies a problem with this technique, which he calls "aliased pay-offs". It turns out that in certain POMDPs, the expected return in a perceptually aliased situation may be the same when a memory bit is set to 0 as it is when this memory bit is set to 1. This can lead to situations where the agent cannot learn how to set this memory bit before entering this perceptually aliased state. Chapter 6 discusses one task in more detail that has this problem.

Another technique that has been used in combination with memory bits is classifier systems (Cliff & Ross, 1994; Kwee et al., 2001). An advantage over table-based systems is that each additional memory bit only increases the size of the observation feature vector by one, and generalization (using state aggregation, see above) can be

done over the entire observation vector, including the memory bits. In fact, short-term memory has been part of the classifier systems formulation from the start, but early implementations did not work very well (Wilson, 1994, 1995), apparently due to an insufficient credit assignment mechanism and a short-term memory that was too complex. Only with the introduction of better credit assignment mechanisms (Wilson, 1994, 1995; Baum, 1999) and memory bits (Cliff & Ross, 1994) did classifier systems begin to work well in POMDPs.

3.6.3.8 Recurrent neural networks approximating value functions

Neural networks have been used to approximate value functions in MDPs (see section 3.5.2.9). One can do the same in the context of POMDPs. However, instead of *feedforward* neural networks, in POMDPs it makes sense to use *recurrent* neural networks (see Werbos, 1974; Rumelhart & McClelland, 1986b; Williams & Zipser, 1989; Williams, 1990; Elman, 1990; Pollack, 1991; Hertz, Krogh, & Palmer, 1991; Arbib, 1995; Hochreiter & Schmidhuber, 1997 for introductions, overviews, and well-known examples of recurrent neural networks). The difference between feedforward neural networks and recurrent neural networks is that the latter do not only have connections going one-way in the direction of the output units; they also have recurrent connections, allowing past activations to influence the current activations. The recurrent activations can provide the agent with the necessary internal state in a POMDP application. The computed value is now some nonlinear function of the observation vector, possibly the action, and the internal state, implemented as a real-valued recurrent activation vector. Again, the idea is that after some amount of learning the observation plus the (Mealy) internal state will together constitute a Markovian state signal, with which values can reliably be associated.

The internal state is some function of the history of the network's inputs, i.e. the history of observations and possibly actions. What this function is depends on the weights. The weights are usually trained using some variation of backpropagation through time, propagating back temporal difference errors of the value function-based reinforcement learning algorithm. In this case the backpropagated errors not only affect the mapping from the network's input to the network's output, but also the internal state transition function. In principle, recurrent neural networks can store, in their internal state, information from arbitrarily long ago, and thus deal with strongly hidden state POMDPs. In practice, however, it can be difficult to train the network to do that. Chapter 6 discusses this issue in much more detail.

Schmidhuber (1990) investigates an actor-critic architecture, where both actor and critic are fully recurrent neural networks, both trained using RTRL. Meeden, McGraw, and Blank (1993) use an Elman-style recurrent neural network (Elman, 1990), but only for the direct reward case, and trained using standard backpropagation. Lin and Mitchell (1992, 1993) also use Elman networks, which approximate Q-learning's value function and which are trained using BPTT.

All of the technical work presented in this thesis falls in this category of recurrent neural networks approximating value functions. Among the especially attractive properties of this approach are the fact that like several other model-free approaches, the recurrent neural network can learn to remember only information which is relevant

with respect to performance. Furthermore, like McCallum's (1996) U-tree, a single system is used to deal with the problem that the agent can have both too much and too little sensory information at the same time, and this system does generalization with adaptive resolution in different parts of the state space. In this sense, this is an elegant approach: a unified approach to constructing a useful state signal is used, which throws away useless and confusing details in the current observation where possible, and fills in the gaps in the current observation where necessary—and this is all learned on the basis of experienced rewards.

Unlike McCallum's (1996) U-tree and most other methods, the value function can be a complex, nonlinear function over the state space, smooth or rugged depending on the problem and the region in state space. And unlike most other methods, the observations can be either continuous or discrete, and the internal state can adapt to problems that require discrete internal state, continuous internal state, or a combination of discrete and continuous internal state (see Williams, 1990, and chapter 6). The use of a value function allows the system to do effective temporal credit assignment, aided by the backpropagation through time (or related) procedure. Backpropagation through time also does relatively sophisticated and directed *structural* credit assignment, compared to many other structural credit assignment schemes: it adapts an individual parameter based on the computed gradient of the error function with respect to that parameter. Finally, one of the attractive properties of this approach with respect to the goals of this thesis is that the use of artificial neural networks, with its disputed but nevertheless obvious similarities with biological nervous systems, may give some insight into how the brain works and learns.

There is no such thing as a free lunch, so obviously there also disadvantages to this approach. One disadvantage can in fact be the power and generality of recurrent neural networks. This power and generality can also be viewed as a lack of bias toward specific problems (Mitchell, 1980; Geman, Bienenstock, & Doursat, 1992). Therefore, compared to methods that have just the right bias for a specific problem, this approach can be relatively slow on that specific problem. However, there are ways in which one could introduce additional bias into this approach. Moreover, there are interesting tasks for which we *need* the power of recurrent neural networks, and for which very few other types of systems could be used at all. This thesis investigates several such tasks. In general, this thesis is more concerned with the *principles* of learning difficult reinforcement learning tasks than with optimizing learning times. The idea is that if we understand how to get fairly general systems to learn complex tasks (within a reasonable timeframe, of course), we gain some insights into the essential aspects of learning, and we can then apply these insights to more biased systems geared toward specific tasks. It remains, however, an important research problem how to construct systems with the right bias for specific tasks.

Another possible disadvantage of using recurrent neural networks approximating value functions is the lack of convergence guarantees. Actually, this is a problem for virtually all the approaches described in this section, and, for that matter, for most of the methods described earlier—but somehow, this is an argument that is always used especially in the context of neural networks. We have seen above that even simple linear function approximators may diverge when used to approximate value functions. For more complex, nonlinear function approximators such as neural networks the situation

may not be better. There are few, if any, guarantees of convergence to global optima or guarantees of stability. This is the case even for feedforward neural networks (but see Suykens, De Moor, & Vandewalle, 1997), and even more so for recurrent neural networks. In fact, Doya (1992) shows that during learning of many interesting tasks, recurrent neural networks must pass through bifurcation points, at which points there is no gradient to be exploited by gradient descent procedures such as BPTT. The position held in this thesis is that even though there may be few theoretical guarantees, this is offset by the potential power of these systems compared to simpler systems which may be proven to converge for certain tasks, but that cannot be used in more complex tasks. In practice, as is also shown in this thesis, the power of recurrent neural networks combined with value functions can be harnessed and be put to good use in complex reinforcement learning tasks, without necessarily having problems of convergence and stability. Nevertheless, understanding and possibly proving convergence and stability are important research issues for the future.

3.6.3.9 Higher order neural networks

Ring (1993a, 1993b) has proposed the use of a different type of neural network for POMDPs. In fact, this is a system which can be viewed as a (historically preceding) variation of the variable history window approaches of McCallum (1995, 1996). It employs *higher order* neural networks, i.e. neural networks where units may directly feed into a weight so as to dynamically alter it temporarily.

The network is, besides the higher order units, a standard feedforward neural network. It has input units that represent observations, and output units that represent actions. A higher order unit is added whenever the weight to which it might be connected is pulled strongly in opposite directions by the learning algorithm, as measured by statistics maintained for each weight. This indicates a "problem" with this connection in terms of the error function, and it suggests that adding a higher order unit which dynamically adjusts the weight may alleviate the problem. Importantly, a higher order unit affects the weight it connects to only one timestep later than its own activation computation. In this way, information from the past may affect the current network activations and with that the current action.

The process of adding new higher order neurons is applied recursively, leading to the possibility of using information from more than one timestep ago. This incremental adding of ever higher order units which make the system sensitive to ever longer-term dependencies is what makes this system resemble other variable history window approaches. For this reason, like those other variable history window approaches, finding longer-term dependencies if there are no short-term dependencies to build on will be difficult, and the system will not generalize correctly when the relevant past event has even the slightest different temporal delay. However, empirical results indicate that the system can learn surprisingly fast and effectively on certain test problems (Ring, 1993a, 1993b).

3.6.3.10 Eligibility traces

As in MDPs, model-free POMDP solution techniques based on value functions can be combined with eligibility traces. Eligibility traces have been found by several researchers (Loch & Singh, 1998; Wiering & Schmidhuber, 1997) to significantly improve learning in POMDPs. Chapter 5 discusses this issue in more detail.

The special benefit of eligibility traces in POMDPs is probably related to the fact that with $\lambda > 0$, the updates do not bootstrap as much as with $\lambda = 0$. That is, they do not base the update only on the estimated value at the next timestep, but instead on actually obtained rewards at the next timesteps together with estimated values. The point is that in POMDPs the estimated value at the next timestep may be very unreliable (much more so than in MDPs), because the state estimation is unreliable. Using the actual rewards may therefore provide more reliable information. Moreover, multiple estimated values are included in the λ-return. Thus, the effect of one incorrectly estimated value, due to a perceptually aliased state, is compensated for by other estimated values, which may be associated with unambiguous states. For these reasons, in most of the work described in this thesis, eligibility traces are used.

3.6.3.11 Evolutionary algorithms

Evolutionary algorithms can also be used in POMDPs. Now the evolutionary algorithm must search in the space of policies with internal state. Several authors have again used recurrent neural networks as the internal state-based policy representation. An advantage of using evolutionary algorithms for developing recurrent neural networks can be that it can be difficult to use standard learning procedures for recurrent neural networks with arbitrary architectures. Evolutionary algorithms do less directed structural credit assignment than gradient descent-based learning procedures, but they can practically always be used, also in architectures where no gradients can be computed (e.g. because of non-differentiable functions).

Yamauchi and Beer (1994) evolve the weights of a fully recurrent neural network controlling the legs of a simulated cockroach. Slocum et al. (2000) evolve the weights of a fully recurrent neural network controlling an agent that catches objects which may disappear from view and must therefore be remembered.

Gomez and Miikkulainen (1999) extend Moriarty and Miikkulainen's (1996) symbiotic evolution approach, SANE, from feedforward neural networks to recurrent neural networks. In the resulting algorithm, called ESP, each evolved individual corresponds to a unit of a recurrent neural network, together with its connections to neighboring units and the corresponding weights. Another difference with SANE is that in ESP different subpopulations are explicitly maintained. During the test phase of a recurrent neural network, a member of one subpopulation is always combined only with members from other subpopulations. On the other hand, during the reproduction phase only members within one subpopulation can be recombined with each other. In this way, specialization is encouraged more than in SANE. Hopefully, useful specialized building blocks are evolved, corresponding to individual neurons that contribute significantly to the policy's performance by taking on a specific role and cooperating well with the

other units. This approach has been demonstrated on fairly difficult non-Markovian versions of pole balancing. Chapters 5 and 6 consider similar pole balancing tasks.

3.6.3.12 Finite state automata

Meuleau et al. (1999) propose a technique that learns the parameters defining a finite state automaton (FSA) which represents the policy. It is an extension of Baird and Moore's (1998) VAPS algorithm, which is limited to reactive policies. Now internal state-based policies can be learned: the state of the FSA represents the experienced history. One limitation (at least at present) is that the number of states of the FSA, and therefore the possible complexity of the internal state-based policy, must be determined beforehand.

The FSA parameters are updated using a stochastic gradient descent method. The method estimates partial derivatives of a very general error measure, defined over *expected episodes* (it is therefore limited to episodic tasks), with respect to the FSA parameters. Possible error measures include but are not limited to Q(0) and Q(1) temporal difference errors—again, we cannot readily classify this technique as direct policy search or based on value functions. Using an FSA with 10 states, a fairly successful policy was learned for a non-Markovian version of pole balancing (see chapters 5 and 6 for similar tasks).

3.6.3.13 Direct policy search based on the success story algorithm and metalearning

As described in the context of MDP solution techniques, the direct policy search method based on the success story algorithm and metalearning (Schmidhuber & Zhao, 1999; Zhao & Schmidhuber, 1998; Schmidhuber et al., 1997, 1996) makes very few assumptions about the nature of the task, including assumptions concerning complete observability of the state. Therefore, this method can be used straightforwardly for POMDPs—and in fact, it has been from its conception.

Now the agent's learned policy must include internal actions that successfully set internal states which can be subsequently used by the agent. Again, all policy changes are evaluated by the success story algorithm, invoked at certain moments called checkpoints. Schmidhuber et al. (1997) and Zhao and Schmidhuber (1998) present results of successful learning in POMDPs of considerable sizes.

3.6.3.14 Exploration issues

Exploring in a clever way is more important but also more difficult in POMDPs than in MDPs (Chrisman, 1992; McCallum, 1996; Wiering & Schmidhuber, 1997). This is due to the fact that in POMDPs the states are not given to the agent. Part of the exploration must be aimed at discovering the environment's state space structure. Until the state space structure is discovered, the agent may not even be able to tell the difference between regions of the state space it has explored extensively and regions it has not explored much.

In many cases, the POMDP will provide unambiguous observations indicating the state in some parts, while providing ambiguous observations (hidden state) in other

parts. In general, it seems reasonable for the agent to do more exploration in the ambiguous parts, so as to discover the state structure there, and not converge too quickly to particular actions. This suggests that directed exploration is particularly useful in POMDPs. Chapter 6 proposes a new directed exploration mechanism based on these ideas.

3.7 Discussion

3.7.1 Learning a model or learning without a model?

In both MDPs and POMDPs, when we do not have a model of the environment, the agent can either learn a model and apply a model-based technique to the learned model, or use a model-free learning technique. Which is better?

On the one hand, in the interaction with the environment more information is available, specifically concerning next observations, than is typically used by model-free techniques. This information can be used to learn a predictive model of the environment.

In POMDPs, in particular, it may in some cases be easier to first learn a model and then learn the policy using a model-based POMDP method than it is to learn an internal state-based policy directly: in a model-free method, learning to infer the state may be confounded by the additional difficulty of learning a value function or policy actions at the same time. On the other hand, in some cases it may require a lot of effort to learn the model, whereas learning the value function or policy directly may be relatively easy. For instance, in certain POMDPs it may be very difficult to learn to infer the complete state of the environment and predict exactly which observations will follow, but the optimal control policy may not require all that. All that is necessary for optimal control is that state estimation is sufficient to learn the correct actions, not to predict everything that will happen next in the world. For example, consider an office environment where a robot roams around equipped with a camera. It is probably very difficult to predict the next images perceived by the robot, but it may be possible to perform many tasks successfully by exploiting simple features extracted from the visual images.

A virtue of model-based methods is that once we have an accurate model, the information captured by the model can be used over and over again by a model-based technique to learn a policy, without a need for more, and possibly "expensive" experiences in the actual environment. Moreover, when we have a model that is at least partly independent of rewards, we can fairly easily learn different policies, if the agent's objectives change. Model-free methods learn only one policy. If the objective changes, all knowledge accumulated by the model-free method must be thrown away. A model can also be used by the agent to do more or less sophisticated forms of look-ahead planning. Furthermore, in learning the model the system may discover certain regularities about how rewards change over time, which may suggest that certain actions which were optimal until now will no longer be optimal in the future (Schmidhuber, 2002).

On the other hand, an approach based on first learning a model and then deriving a policy from the model can be sensitive to the accuracy of the learned model. A model-free method, in contrast, receives immediate feedback from the environment about the success of its policy, without the complicating and possibly confounding intermediate stage of the model. In this way, a model-free method may be corrected more directly for errors in its expectations about the environment. Furthermore, it is possible for a model-free method to reuse past experience, like model-based methods, simply by storing past experiences and using them later to train or adjust the policy (Lin & Mitchell, 1993, 1992; McCallum, 1997, 1996).

3.7.2 Representations?

In a way, the discussion about models in reinforcement learning is reminiscent of the older AI debate about the use of world models, which was briefly discussed before in sections 2.3.7 and 2.3.8. One of the arguments of the adaptive behavior approach against classical AI's world models was that it is notoriously difficult to learn and update a complete and exact world model. Instead, a "reactive approach" was proposed, in which an agent reacts directly to environmental stimuli and "uses the world as its own best model" (Brooks, 1991a). In POMDPs it is clear that a purely reactive approach cannot possibly work in all cases. There must be some internal state, but this internal state does not have to be in the form of a complete and exact world model. Instead, the internal state may correspond to a purposively incomplete model or to a model which takes into account its own uncertainty and which is therefore relatively robust against noise and inaccuracies. Or the internal state may, in model-free techniques, have a very different form, such as a limited set of past observations and actions, memory bits, FSA states, pointers to subpolicies, or real-valued activation vectors. In none of these approaches there is a need to go back to exact, full-blown world models or symbolic representations to obtain successful policies for moderately complex "representation-hungry" tasks.

Whether or not we should call the internal state a "representation" or not (van Gelder, 1998; Clark, 1997; Keijzer, 2001) then becomes a matter of taste. If one's definition of a representation requires it to be a symbolic representation or to always correspond straightforwardly to easily identifiable properties of the environment, then these different forms of internal state are not representations. If one's definition of a representation only requires it to correspond to some process or structure that is maintained inside the agent and which contains information that is not contained in the immediate sensory input, then these different forms of internal state are representations.

3.7.3 Value functions or direct policy search?

Now that a number of value function approaches and direct policy search approaches have been discussed, we are in a better position to discuss their relative merits.

One advantage of value functions over direct policy search, at least direct policy search in its simplest form, is that the information contained in individual states can be used more efficiently. For example, if a state reliably precedes high reward, a value

function method will give that state a high value. This state will then start to function as a "subgoal", and "subpolicies" that lead to that state will be rewarded. In this way, value functions can automatically break down a problem into subproblems, each of which may be learned more or less independently.

A direct policy search method that varies and evaluates policies as a whole does not automatically do this. Put another way, because the information contained in individual states and state transitions is used more explicitly, value function approaches can do more focused structural and temporal credit assignment than simple direct policy search. However, as we saw above, there are ways of extending direct policy search methods such that they can also do more focused structural and temporal credit assignment and exploit some type of subgoal or building block information (e.g. Schmidhuber et al., 1997; Moriarty & Miikkulainen, 1996; Gomez & Miikkulainen, 1999).

A separate issue is the issue of convergence and stability. As discussed before in the section on generalization (3.5.2.9), a number of studies have shown that certain value function approaches can be unstable and fail to converge when they are combined with even simple function approximators. This is one reason why in recent years several people have started to take a closer look at methods which represent the policy directly rather than indirectly as in straightforward value function methods. Such methods may not suffer in the same way or to the same extent from problems with convergence and stability. However, most of those methods still have also some sort of value function component, which guides the policy component to good policies, e.g. in an actor-critic style architecture.

There are a variety of other factors that determine the relative advantages of direct policy search and value functions. One factor is the number of successful policies in policy space. If there are a lot of policies that all work well, then direct policy search has a greater chance of finding one in a reasonable time, especially if one searches in parallel, e.g. using an evolutionary algorithm. Another factor is the complexity of the policy representation. If a successful policy can be represented by a limited number of parameters, a relatively undirected direct policy search method has a reasonable chance of finding it. In both of these cases, learning a possibly complex value function can be just an inconvenient detour.

In any case, any advantage that value functions may have over direct policy search requires the state to be known, which is not immediately the case in POMDPs. This makes the situation radically different for POMDPs than for MDPs. If the POMDP is such that observations contain very little useful information about the state of the environment, values cannot be reasonably estimated and associated with states, and it may be better to search directly in the space of policies. If, on the other hand, observations contain at least a significant amount of information about the state, then in many cases it would seem to be worth the effort to try to decompose the task into subtasks which can be handled with reactive value function-based subpolicies (Wiering & Schmidhuber, 1997), or to reconstruct the environmental state signal from the experienced history and associate values with these reconstructed states—the approach taken in this thesis.

Chapter 4

The trade-off between perception and internal state

Summary

An MDP can be dealt with by learning a memoryless or perception-based policy, which directly maps perceptual inputs representing environmental states to actions. However, this may involve a difficult pattern recognition problem when the agent must learn to generalize over a large environmental state space. The main point of this chapter is that this pattern recognition problem can become so severe that it makes sense to partially sidestep this problem by using an internal state-based policy which exploits temporal regularities inherent in many reinforcement learning tasks. This point is demonstrated in a simulation experiment where both the difficulty of using perception and the difficulty of using internal state are systematically varied. The agent is controlled by a recurrent neural network capable of learning either type of policy, and the different types of policies that the agent converges upon are analyzed using behavioral tests, FSA extraction, and by looking at learning times.

4.1 Introduction

As explained in the previous chapter, a Markovian reinforcement learning task or MDP can be solved using a reactive or perception-based policy, whereas a non-Markovian task or POMDP requires an internal state-based policy. However, even in the Markovian case, learning just the perception-based policy can be difficult. One common problem is that the environmental state space can be very large. Because the agent will not be able to obtain sufficient evidence for each state-action pair, it must learn to productively generalize over the environmental state space. In most of these cases, the perceptual input is a vector representation of the environmental state, such that the generalization problem effectively becomes a problem of (static) pattern recognition.

The main point of this chapter is that this pattern recognition problem in a Markovian task can become so severe that it makes sense to sidestep it (perhaps partially)

by learning to exploit information from previous timesteps. Rather than learning to recognize complex regularities in individual inputs, it may in some cases be easier to learn temporal regularities in sequences of inputs (and actions). Thus, even though a Markovian task of this kind can in principle be solved using a perception-based policy, it may be a good idea to use an internal state-based policy, which can exploit these temporal regularities. This assumes that there *are* such temporal regularities. However, this seems a reasonable assumption: in most realistic environments transitions to subsequent states are far from being completely random, so there will be temporal regularities in the sequence of observations.

Internal state used in MDPs may be called *supportive state*, because even though this internal state is not strictly *necessary* to solve the (Markovian) task, it provides extra information that *supports* the agent in choosing its actions. Importantly, useful supportive states need not simply be given to the agent. The idea is that even when the agent first has to *learn* useful internal states before it can exploit them, this can still be more efficient than just learning the perception-based policy. This is interesting because it provides an argument against the strong tendency in the reinforcement learning community to practically equate Markovian tasks with the use of perception-based or memoryless policies.

A related but subtly different point was made previously in a different context: research on recurrent neural networks has shown that in some cases, transforming a static pattern recognition problem into a temporal pattern recognition problem can make the problem easier to solve (Schmidhuber, personal communication). An example is the parity problem: an 8-bit parity recognition problem that is very difficult to solve using a feedforward neural network with 8-bit input vectors can become much easier if the 8 bits are fed consecutively into a very small recurrent neural network with one input line. The difference with the argument made in this chapter is that in this chapter the input representation is *not* changed: the point is that in the same task with the same inputs, using an internal state-based policy can make the MDP easier to solve than with a perception-based policy.

More precisely, the hypothesis investigated in this chapter is that there is a *trade-off* between perception and internal state. Depending on the relative difficulty of learning to recognize the patterns in single inputs and the relative difficulty of learning useful internal states, a perception-based policy or internal state-based policy may be easiest to learn. In this chapter, "easiest to learn" means easiest to learn for an agent capable of learning either kind of policy. The interesting thing about this is that the choice of policy is up to the learning system itself: it can autonomously decide which information it finds most useful for its policy, not based on explicit instruction, but based only on experienced rewards. The capability to learn either kind of policy is realized by using an Elman recurrent neural network (Elman, 1990). If the agent always learns an internal-state based policy in a certain condition, this is taken as evidence that this is the policy that is easiest to learn for the agent under that condition. However, we also consider learning times of different policies under different conditions as a measure of difficulty of learning different policies.

The next section describes the setup of the simulation experiments that were carried out to demonstrate the trade-off, and it points out similarities to other work. Section 3

presents the results of the simulations and an analysis of the results. Section 4, finally, contains a general discussion.

4.2 Setup of the simulation experiments

4.2.1 Learning task

The simulated agent's environment is a small grid maze, consisting of open space and impenetrable walls (see figure 4.1). The agent's job is to move to the goal position, starting from the fixed starting position (S). The goal is randomly located in one of two possible positions, which can be referred to as "left" or "right". In figure 4.1 the goal (G) is in the right position, the left goal position is in the upper left corner. If the agent reaches the goal position, it receives a reward of 1, at all other points the reward is 0. The agent can be oriented to the north, east, south, or west. It has a choice of 3 actions: go forward, turn left, or turn right. If the agent attempts to move through a wall, it stays in the same grid position, but turning is possible. The agent's perceptual input is a 13-element vector representing the complete environmental state (see figure 4.1): even though it detects walls, open space, and the presence of the goal only locally around the agent, each environmental state corresponds to a unique perceptual input.[1]

At the end of the corridor coming from the starting position the agent faces a T-junction, at which point it must make a crucial decision whether to go left to reach the left goal position, or right to reach the right goal position. This is the point where the pattern recognition problem associated with a perception-based policy is systematically varied in the experiment. When the goal is randomly placed either left or right at the beginning of a learning episode, a "corresponding" pattern of walls and open space is placed above the T-junction, in the area indicated by bold lines in figure 4.1. Essentially, when the agent is at this T-junction facing north, part of its perceptual input consists of a pattern of walls and open spaces that presents the agent with the *parity problem*. If the number of 1s (open spaces) in that pattern is even, the goal is on the left side, so the agent should turn left. If that number is odd, the goal is on the right side so the agent should turn right.

The parity problem provides us with the means to gradually (though not necessarily uniformly) increase the difficulty of the pattern recognition problem associated with a perception-based policy by increasing N, the length of the pattern of bits. In this experiment, N is varied from 1 (very easy) to 5 (very hard). In the 5-parity condition, all 5 positions indicated by bold lines in figure 4.1 code for the parity. In the 1-parity condition, only the upper left grid position with bold lines is used, and the other four are frozen as walls.

The parity problem is a classic and challenging problem, even using supervised learning. Feedforward neural networks without a hidden layer cannot solve this task for $N > 1$ (Minsky & Papert, 1969), but feedforward networks with a hidden layer can

[1] To prevent a few positions in the maze from looking identical, the wall cells in the maze have small variations in "color". Leaving out this variation would technically make the task non-Markovian. The task would still be Markovian with respect to rewards, and preliminary experiments made clear that it makes little difference for the results.

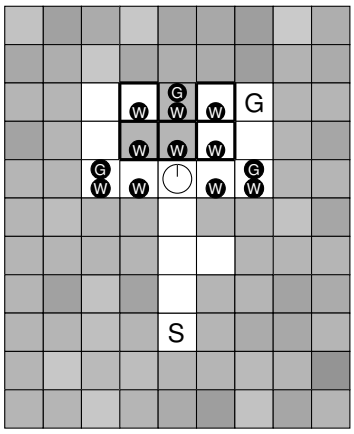

Figure 4.1: Example of a 5-parity, corridor condition 1 maze. The agent is at the central T-junction, oriented to the north. Black circles denote its sensors for walls (W) and for the goal (G). Bold lines indicate the area encoding the odd parity pattern `10011`, which can be perceived using wall sensors. S is the starting position and G is the goal. One step before the T-junction one grid cell on the same side as the goal has been turned into open space.

(Rumelhart et al., 1986). $N = 1$ is easy: if the single input is 0, go left, if it is 1, go right. $N = 2$ already amounts to the XOR problem.

At the same time, the environment presents the opportunity to develop useful internal state—supportive state. This is so because, in addition to the parity pattern being placed at the T-junction at the beginning of an episode, a pattern is placed in the corridor from the starting position leading up to the T-junction. This pattern, again encoded in walls and open space, also indicates where the goal is. If the agent learns to extract this information and encode it in its internal state, at the T-junction it does not have to rely on perception of the parity pattern, but it can instead rely on its internal state.

The difficulty of learning to extract this information, which corresponds to the difficulty of learning an internal state-based policy, is varied systematically as well. It ranges from 1 to 4, where corridor condition 1 means that it is very easy to learn the internal state-based policy, and corridor condition 4 means that it is very hard to learn the internal state-based policy. In corridor condition 1, one wall cell on the side of the goal position is turned into open space one step before the T-junction (see figure 4.1). In corridor condition 2, this becomes two steps, so that the agent must now learn to remember the information one extra timestep. Corridor condition 3 is the same as condition 2, except that now one of the cells used in condition 1 is randomly turned into open space, without this having any relation to the goal position. This amounts to having extra noise which may perturb the agent's internal state from the previous timestep. In corridor condition 4, observation of a single grid cell is no longer sufficient to determine where the goal is (in contrast with the other conditions). Now

the location of the goal depends on the pattern in two grid cells, one of which can only be observed two steps before the T-junction and one of which can also be observed one step before the T-junction. The difficulty of these 4 corridor conditions was determined based in part on known factors (Cleeremans, Servan-Schreiber, & McClelland, 1989; Lin & Mitchell, 1993), and in part on preliminary experiments.

The agent takes steps of 2 grid cells at a time. This prevents the agent from perceiving the parity pattern at the T-junction over multiple time steps, which would be the case if the agent took actions of 1 grid cell at a time and which would seriously confound the experiment. Now the agent's observation only gives information about states that can be reached using one action.

This learning task may look somewhat "fabricated", but this is necessary in order to systematically investigate the trade-off between perception and internal state. Nevertheless, the task shares a number of properties with many reinforcement learning tasks and is realistic in some respects. Both the parity patterns and the corridor patterns are part of the regular input to the agent, in the sense that they are perceived by sensors that are also used to avoid the walls and navigate in general. Furthermore, the parity patterns and corridor patterns are present at the beginning and middle of the sequence of actions leading to the goal, not at the very end. Therefore, the "relevance" of these patterns cannot be determined immediately. This relevance can only be gradually learned, as the reward obtained at the goal position is eventually "fed back" by means of the reinforcement learning algorithm.

In summary, the most important properties of the learning task are that it is Markovian, and that both the complexity of patterns to be exploited by a perception-based policy and the complexity of patterns to be exploited by an internal state-based policy are systematically varied. Effectively, we have a 5×4 experimental design, and each agent is assigned to one particular condition in that experimental design.

4.2.2 Architecture and learning algorithm

The agent is controlled by a Simple Recurrent Network or Elman network (Elman, 1990), depicted in figure 4.2. An Elman network differs from a multilayer feedforward neural network in that it has so-called context units. The context units hold a copy of the hidden unit activations of the previous timestep. Because the hidden unit activations, in turn, are determined in part by the context unit activations, the context units can, in principle, retain information from many timesteps ago.

The network's activations are computed as follows. The net input $net_h(t)$ of hidden unit h at time t is calculated by

$$net_h(t) = \sum_o w_{ho} y_o(t) + \sum_a w_{ha} y_a(t) + \sum_z w_{hz} y_z(t) \tag{4.1}$$

where $y_o(t)$ is the activation of observation unit o, $y_a(t)$ is the activation of action unit a, and $y_z(t)$ is the activation of context unit z. w_{ho}, w_{ha}, and w_{hz} are the corresponding weights. There are as many action units as there are actions. All action units have activation 0 except one action unit, which has activation 1 and which corresponds to the action whose Q-value is computed. Furthermore, there as many context units as there are hidden units. In the computation of the net input of the hidden layer units,

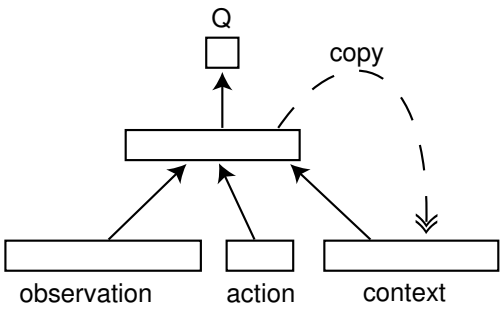

Figure 4.2: Q-Elman neural network. The solid lines indicate unidirectional, fully connected, learning weights. The dashed line represents the copy operation of the hidden units to the context units.

the action vector and the context unit activations are treated as just another input. The activation of hidden unit h at time t is next computed by

$$y_h(t) = f(net_h(t)) \tag{4.2}$$

where f is the standard logistic sigmoid function, squashing the net input to the range $[0, 1]$, using

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{4.3}$$

The net input $net_k(t)$ of the single output unit k is computed by

$$net_k(t) = \sum_h w_{kh} y_h(t) \tag{4.4}$$

where w_{kh} is the weight of the connection from hidden unit h to the output unit k. The output unit's activation is computed by

$$y_k(t) = f(net_k(t)). \tag{4.5}$$

The activations of the context units at the next timestep are simply a copy of the activations of the corresponding hidden units at the current timestep:

$$y_z(t + 1) = y_h(t). \tag{4.6}$$

The activations of the context units at $t = 0$ are 0.

The Elman network learns to approximate the value function of one of the most widely used reinforcement learning algorithms, Q-learning (Watkins, 1989, see section 3.5.2.2), and is therefore called a Q-Elman network. In this experiment, no eligibility traces are used, so we have Q(0)-learning (Watkins, 1989). Compared to standard Q(0)-learning, there are two major modifications. First of all, the Q-values are not stored in a table, but approximated using a neural network. Secondly, the computed

Q-value at time t is not only a function of the observation $o(t)$ and action $a(t)$, but also of the Mealy internal state $z(t)$, encoded by the context units of the Q-Elman network. Thus, the network computes $Q(o(t), z(t), a(t))$. The network's single output $y_k(t)$ codes for the Q-value in the following way:

$$Q(o(t), z(t), a(t)) = 2(y_k(t) - \frac{1}{2}). \tag{4.7}$$

Thus, in the beginning of learning, when the activations of the output unit are around $\frac{1}{2}$ because the randomly distributed weights are around 0, the estimated Q-values are around 0. At each iteration, the Q-value for each action in the current state is computed. For each action, one of the units of the action input vector is set to 1 and the others to 0, and subsequently the activations in the hidden units and the output unit are computed, based on this action input vector, the observation vector, and the context unit vector.

The Q(0)-learning algorithm is implemented as follows. Each iteration, after the execution of an action and the environment's response (an immediate reward and a new observation is obtained), all weights in the network are updated according to

$$w_{im}(t+1) = w_{im}(t) + \alpha E^{TD}(t) \frac{\partial Q(o(t), z(t), a(t))}{\partial w_{im}} \tag{4.8}$$

where

$$E^{TD}(t) = r(t+1) + \gamma \max_a Q(o(t+1), z(t+1), a) - Q(o(t), z(t), a(t)) \tag{4.9}$$

and w_{im} is the weight from unit m to unit i. In this work, $Q(o(t), z(t), a(t))$ is a simple linear function of the activation of the output unit, so

$$\frac{\partial Q(o(t), z(t), a(t))}{\partial w_{im}} = 2 \frac{\partial y_k(t)}{\partial w_{im}}. \tag{4.10}$$

The constant, 2 in this case, can be absorbed into the learning rate. The partial derivatives with respect to the weights from the hidden layer to the output layer in the Elman network are

$$\frac{\partial y_k(t)}{\partial w_{kh}} = f'(net_k(t)) y_h(t) \tag{4.11}$$

and the partial derivatives with respect to the weights to the hidden layer are

$$\frac{\partial y_k(t)}{\partial w_{hm}} = w_{kh} f'(net_k(t)) f'(net_h(t)) y_m(t). \tag{4.12}$$

where m can be an observation unit, an action unit, or a context unit. The nice thing about using the logistic function for our squashing function f is that f' can be computed very easily:

$$f'(net_i(t)) = y_i(t)(1 - y_i(t)). \tag{4.13}$$

In effect, $E^{TD}(t)$, the temporal difference error in equations 4.8 and 4.9, is back-propagated through the Elman network (Werbos, 1974; Rumelhart et al., 1986; Elman,

1990). Standard backpropagation is used, which in this case can be viewed as backpropagation through time truncated at one step back in time, because the context unit activations represent the hidden unit activations from one timestep ago.

The number of observation units in this task is 13. The number of action units is 3 (one for each action: go forward, go left, go right). The number of hidden units and context units used in this experiment is 10. γ is .9. The learning rate α is .1. This relatively high value was chosen because the estimations of the Q-values often differ by only small amounts (especially in the beginning of a run), so that backpropagated errors are small.

The algorithm is applied as follows. First, the Q-values for all 3 actions are computed in the Q-Elman network, using the current observation vector (which exactly represents the environmental state) and the activations of the context units (internal state). Next, the action is selected using the probabilities $p(s, a)$ of each action, computed according to the Boltzmann distribution (see chapter 3):

$$p(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{m=1}^{M} e^{Q(s,a_m)/\tau}} \tag{4.14}$$

where M is the number of possible actions (3) and τ is the temperature of the action selection mechanism, in this experiment fixed at .05 for the entire duration of a run. Thus, in this chapter undirected exploration is used. After selection of the action, the action is executed and the environment feeds back the reward. The maximum of the Q-value in the new situation is computed, again using the Q-Elman network. Together with the reward this value is used to perform a single backpropagation sweep in the network (thus, learning occurs at every timestep, and not in so-called batches). Then a new iteration starts. If the goal is reached ($r = 1$) or the number of actions exceeds the time-out value of 30, a new episode starts.

This reinforcement learning algorithm does not explicitly instruct the network on what actions to take or what information to attend to. The network may "decide" to ignore the context unit activations and only use the immediate perception of the current environmental state. It then basically behaves as a feedforward network and it implements a perception-based policy. Alternatively, the network may decide to pay more attention to the context units, develop differentiated internal states, and thus implement an internal state-based policy. In any case, this choice is based on experienced success in the particular environment, and the hypothesis investigated in this chapter says that there will be a trade-off between the two options.

4.2.3 Related work

Feedforward neural networks are combined with Q-learning by, among others, Lin (1992), Anderson (1993), Crites and Barto (1996), Humphrys (1997), Abul et al. (2000), ten Hagen and Kröse (1998), with SARSA in Rummery and Niranjan (1994), and with TD-learning of state values in Anderson (1987), Tesauro (1992, 1994). In Watkins (1989) and Sutton (1996) CMAC, another type of function approximator, is used in combination with Q-learning and SARSA respectively.

Schmidhuber (1990) investigates fully recurrent neural networks that are trained using TD-like reinforcement learning algorithms. Meeden et al. (1993) use Elman

networks, but with a direct reward reinforcement learning algorithm. Like the current study, Lin and colleagues (Lin & Mitchell, 1993, 1992) combine a version of Q-learning with Elman networks. There are a number of differences in learning algorithm and architecture. First of all, they use a separate network for each action (as in Rummery & Niranjan, 1994; Lin, 1992; Abul et al., 2000), whereas here just one network is used, allowing for more generalization. They use backpropagation through time, whereas here ordinary backpropagation is used. Finally, their specific learning algorithm (which is essentially the same as the one used in Lin, 1992) allows updating only after the completion of an entire episode, and they replay memorized episodes many times. In contrast, the algorithm used here is completely online and local in time, like the one used in Schmidhuber (1990).

4.3 Results

4.3.1 Analysis of the agents' behavior

For each experimental condition, 10 runs were performed. Since we have a 5×4 experimental design, this makes 200 runs in total. The termination criteria were as follows. Using greedy action selection ($\tau = 0$), i.e. choosing the action with the highest Q-value, the agent must reach the goal in at most 5 steps (the optimal number of steps is 4). This must be the case for all parity and corridor patterns that the agent was trained on. This means that the agent must always make the correct decision at the T-junction. Secondly, using stochastic action selection ($\tau = 0.05$), the agent must reach the goal before the normal time-out of 30 steps in more than 95% of the episodes, and the running average of the number of steps needed to reach the goal must be below 7.

The second criterion was added in order to enforce "robust" solutions. Note that the optimal policy using greedy action selection can be found without the Q-values having the exactly correct values. All that is necessary is that the right action has a slightly higher Q-value than the others. This would lead to an optimal greedy policy, but poor stochastic policy. The second criterion thus ensures that the Q-values for a given situation are sufficiently different, so that the solution would not, for instance, be too easily disturbed by noise. If a run did not reach the termination criteria within 100 million iterations, it was considered a failure. This occurred for a few runs in the most difficult experimental conditions. After completion of the runs, they were individually analyzed.

To determine the kind of policy followed by an agent (perception-based or internal state-based), each agent was tested in two ways. The relationship of the parity information with the goal position was reversed, so even parity now corresponds to the goal position being *right* and odd parity corresponds to *left*. If the agent now turns left at the T-junction when it should turn right and vice versa, this means it relies on perception of the parity information. If its performance is not affected, apparently it uses internal state. As the other, complementary test, the relationship of the pattern in the *corridor* with the goal position was reversed, with the normal parity to goal relationship in place. Now, if the agent is misled and goes the wrong way at the T-junction, this means it uses internal state. If it ignores the misleading information and

Table 4.1: Number of policies converged upon in different conditions. Par stands for N-parity condition, Cor stands for Corridor condition.

	Policy	Par 1	Par 2	Par 3	Par 4	Par 5
Cor 1	Perception	10	5	0	0	0
	Mixed	0	4	1	0	1
	Internal State	0	1	9	10	9
Cor 2	Perception	10	10	4	1	1
	Mixed	0	0	3	5	3
	Internal State	0	0	3	4	6
Cor 3	Perception	10	9	8	6	3
	Mixed	0	1	2	1	2
	Internal State	0	0	0	3	3
Cor 4	Perception	10	10	7	6	4
	Mixed	0	0	3	1	0
	Internal State	0	0	0	2	1

maintains good performance, this means it uses perception of the parity information. The results of this analysis are shown in table 4.1.

When the pattern recognition problem associated with the perception-based policy is very easy (parity condition $N = 1$), all agents learn a perception-based policy. This shows, first of all, that agents do not develop internal state simply because the Q-Elman network has recurrent connections. Apparently, there must be a reason to develop internal state, and this reason is absent in the 1-parity cases. However, when the parity problem is hard ($N = 4$ and $N = 5$) but the pattern in the corridor is easy (corridor condition 1), then agents use an internal state-based policy. This internal state can be called supportive state, because an agent could have relied on perception alone, but instead develops internal state to support its action decisions. In the intermediate conditions, a gradual transition from perception-based policies to internal state-based policies can be observed. This result confirms our hypothesis, for this setting at least, that there is a trade-off between perception and internal state. When the pattern recognition problem associated with a perception-based policy becomes difficult, perception may be traded for internal state, and vice versa.

Interestingly, a number of agents adopt *mixed* policies. Such agents make mistakes in *both* behavioral tests. This means that for some parity patterns encountered at the T-junction, the agent uses perception of that pattern, whereas for others it relies on internal state. To explain this, we should note that as far as the hidden layer of the Elman network is concerned, both perceptual input and context unit activations appear as an input pattern. There is no compelling reason why the agent should not exploit distinctive patterns in both types of input.

The results shown in table 4.1 can be visualized in a straightforward way. Using the two tests described above, the *proportion of pattern variations in the maze that*

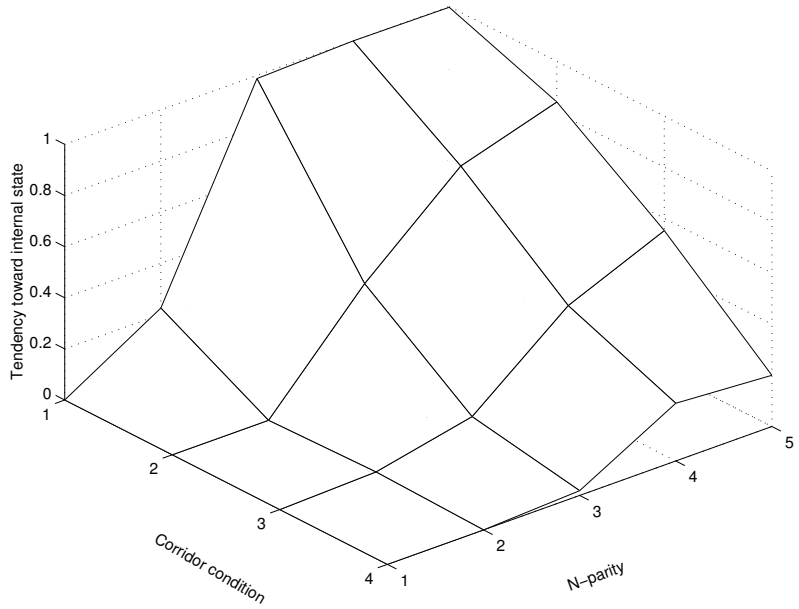

Figure 4.3: The tendency toward internal state as a function of N-parity and corridor condition.

the agent deals with using internal state can be computed. Thus, this value is 1 for a pure internal state-based policy, 0 for a pure perception-based policy, and somewhere in between for a mixed policy (not shown in table 4.1). The average value, called the tendency toward internal state, is plotted for each condition in the graph depicted in figure 4.3. This graph shows the trade-off surface between perception and internal state.

Several internal state-based and mixed agents developed a surprising strategy in the corridor. When the goal is on one side, they proceed to the T-junction directly. When the goal is on the other side, however, they make a turn at the starting position before moving to the T-junction. This action prevents the agent from reaching the optimal policy of 4 steps. Q-learning is only guaranteed to reach the optimal policy under a number of conditions, one of which is that the Q-values must be stored in a table (Watkins, 1989). If function approximation is used, the usual problem of local optima applies. However, in this case, this extra action may actually be useful. It may help in creating two very different hidden unit activation patterns for each of the two goal positions, because it makes the sensory inputs and actions in the corridor very different. This results in two very different internal states which subsequently can be easily discriminated at the T-junction. Thus, this seems to be an instance of spontaneous *epistemic action*, action that is not primarily intended to accomplish a goal, but rather to facilitate the computational task faced by an agent by gathering additional information. Humans and animals often use epistemic action to cope with

difficult tasks (Clark, 1997). It is similarly reminiscent of actions taken to decrease the uncertainty in POMDPs, which in the formal POMDP framework can be shown to be optimal in certain cases (see section 3.6.2.1).

When both the parity pattern and the corridor pattern are very complex, not all runs reach the termination criteria. In those cases, the agent does not always make the correct decision at the T-junction when using greedy action selection; but still it always reaches the goal, because it turns around once it notices that it went the wrong way. It is interesting to note that those runs that *do* reach the termination criteria in these difficult conditions include both perception-based policies and internal-state based policies. For the perception-based policy this means that, as was claimed in the introduction, this kind of policy is indeed *possible* in the 5-parity case, but that the internal state-based policy is in most conditions easier to learn for an agent capable of learning either kind of policy. For the internal state-based policy it means that fairly complex temporal regularities in reinforcement learning tasks can be picked up using even a simple system such as an Elman network.

4.3.2 FSA extraction

It is possible to determine the kind of policy used by an agent by analyzing the internal mechanisms, rather than inferring it from the behavioral tests described above. This can provide additional confirmation for the results obtained with the behavioral tests and it can help explain these results. In general, it is always interesting to attempt to understand policy solutions learned by an agent, in order to better understand the capabilities and weaknesses of the used algorithms. In this case, we may gain insight into the mechanisms employed by the neural network when implementing a perception-based, internal state-based, or mixed policy, instead of treating the neural network as a "black box", as happens in much neural network and machine learning research. For instance, we may learn the number of internal states of an internal state-based or mixed policy induced by the neural network, and we may learn when the network changes its internal state during an episode.

One complication here is that the hidden units of the Elman network are used both to recognize features in the perceptual input and, when they are copied to the context units, as internal state. Because the perceptual inputs change constantly, the hidden units change constantly and with that the context units. This makes it look as if the network uses variable internal state. However, if the weights from the context units to the hidden units are such that the variations in the context unit activations are basically ignored and do not have an effect on the agent's behavior, then the system uses a perception-based policy. Thus, the question is to what extent these changes in the context units *functionally affect* the network's behavior.

The method used here to answer this question is Finite State Automaton (FSA) extraction. One version of this method, based on extracting input FSAs, was used by Crutchfield (1994) to analyze general dynamic systems and by (Giles et al., 1992; Blair & Pollack, 1997; Casey, 1996) to analyze supervised learning recurrent neural networks accepting or rejecting sentences from artificial grammars. Here the method is generalized to input-output FSAs extracted from our reinforcement learning recurrent neural networks.

The basic idea is to "discretize" the continuous internal state space so as to arrive at discrete FSA states. In a second phase, the FSA is minimized by using a variation of the Hopcroft minimization algorithm (Hopcroft & Ullman, 1979). In effect, different FSA states, associated with continuous internal state vectors that can be far apart in context unit activation space but that do not differ in their behavior, are merged, so as to arrive at a minimized FSA whose states are functionally different. In some cases, this reduced the number of states from as many as 228 to only 1 or 2 states. This great reduction is possible because, as explained above, variations in context unit activations often only reflect changing perceptual inputs rather than functionally different internal states. Appendix A describes the algorithms for FSA extraction and subsequent minimization in more detail.

An FSA where both input and output are stored with the edge is a Mealy machine (Hopcroft & Ullman, 1979, and see section 3.3.2). For the purposes of this chapter, this is to be preferred over the alternative input-output FSA convention, Moore machines, where outputs are stored with the states (even though they can always be rewritten into each other). This is so because it allows us simply to equate perception-based policies with single-state input-output FSAs and internal state-based policies with multiple-state input-output FSAs. The Mealy machine convention of edge labeling is used: input/output. Output 0 stands for "go left", output 1 stands for "go forward", and output 2 stands for "go right". The FSA graphs were created automatically, the text between brackets, indicating what certain important input numbers stand for, was manually inserted later.

This procedure was applied to the successful agents. An example from the 1 parity, corridor 1 condition is depicted in figure 4.4. It shows that this agent has learned a perception-based policy. The agent has just one state, and it makes the decision whether to go left (output 0) or right (output 2) at the T-junction based on recognition of the (very simple) parity problem perceived there.

Figure 4.5 shows an example from the 4 parity, corridor 1 condition. Here the parity problem is much harder, and the agent has solved the task by inducing two states: an internal state-based policy. It goes from the initial state, state 0, to state 1 when it sees that in the corridor one step before the T-junction the right wall cell is turned into open space ("Corridor Right"). In state 1, it always turns right (output 2) at the T-junction, no matter what the parity pattern is ("Even" or "Odd"). Conversely, in state 0, the agent always turns left. When this particular agent is in state 1, after making the turn at the T-junction it always changes its state back to state 0. When it is in state 0, it only sometimes changes its state.

Figure 4.6 shows an example of a mixed policy, from the 3 parity, corridor 1 condition. Similar to the example in figure 4.5, the FSA consist of two states. The agent goes from the initial state 0 to state 1 when it perceives "Corridor Right" and stays in state 0 when it perceives "Corridor Left". In state 0 most odd parity patterns are ignored and the agent turns left (output 0), but this is not the case for perceptual input number 10, corresponding to odd parity pattern `100`. Apparently, this pattern is recognized at the T-junction, such that the agent knows it should turn right, even when it is in the "wrong" internal state.

Figure 4.7, finally, shows a more complex internal state-based policy, from the 5 parity, corridor 4 condition. This agent has induced more states than the two states

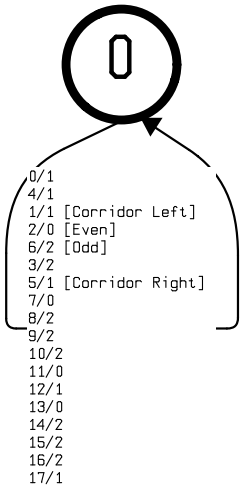

Figure 4.4: Extracted single-state FSA corresponding to a perception-based policy, from the 1 parity, corridor 1 condition. Within this single state, an even parity pattern (number 2) is correctly dealt with by emitting output 0 (go left), an odd parity pattern (number 6) is correctly dealt with by emitting output 2 (go right).

that are strictly necessary for a successful internal state-based policy. Every state is connected to all other states and itself. The FSA uses state 0 (the initial state) and 1 to code for the two different corridor patterns that indicate the left goal position. Conversely, it uses state 2 and 3 to code for the two different corridor patterns that indicate the right goal position. Changes in the state occur at different timesteps for different corridor patterns, and the FSA normally passes through state 1 on its way to state 2 and 3. This agent also uses the kind of epistemic action described above. When the goal is on the right side, it turns left at the starting position before proceeding to the T-junction, presumably to create easily distinguishable internal states.

Inducing more states than strictly necessary for a successful internal state-based policy actually occurred for many internal state-based and mixed agents, especially the ones in the more difficult conditions. The number of induced states could be as high as 6. In most cases, these extra states only affect the behavior in the testing phase, when the agent is misled by the corridor pattern, such that it goes the wrong way at the T-junction; but this does prevent these extra states from being "minimized away". The agent's subsequent behavior then depends in a subtle way on the particular history of inputs, even though the agent did not rely on those inputs at the T-junction. Note that it does not require any extra "effort" on the part of the recurrent neural network to induce a few more states than is strictly necessary. Such redundant internal states are therefore not very surprising. More surprisingly, even a few perception-based agents contained some redundant internal states that could not be minimized away. However,

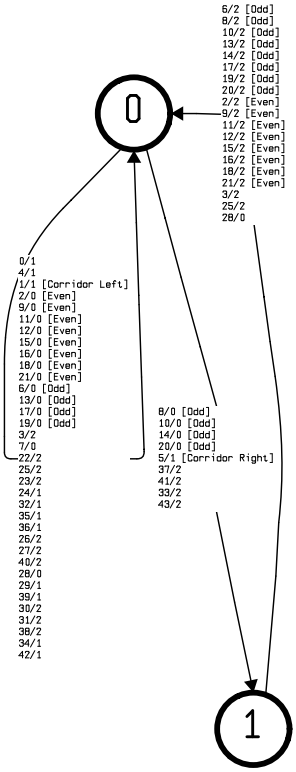

Figure 4.5: Extracted two-state FSA corresponding to an internal state-based policy, from the 4 parity, corridor 1 condition. When the agent detects that the right wall cell in the corridor is turned into open space (observation 5, "Corridor Right"), action 1 (go forward) is emitted and the state changes to state 1. All odd and even parity patterns are subsequently dealt with by emitting output 2 (go right).

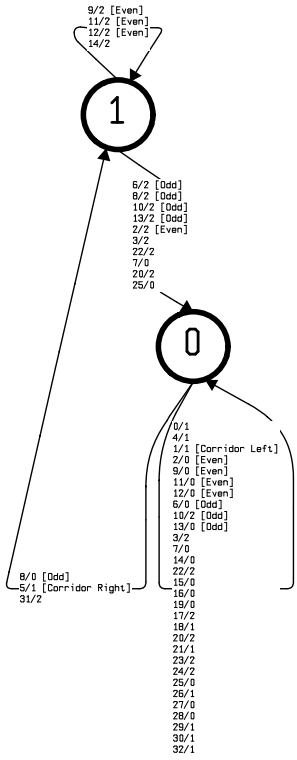

Figure 4.6: Extracted two-state FSA corresponding to a mixed policy, from the 3 parity, corridor 1 condition. Input number 10, corresponding to an odd parity pattern, is correctly dealt with in both states by emitting action 2 (go right).

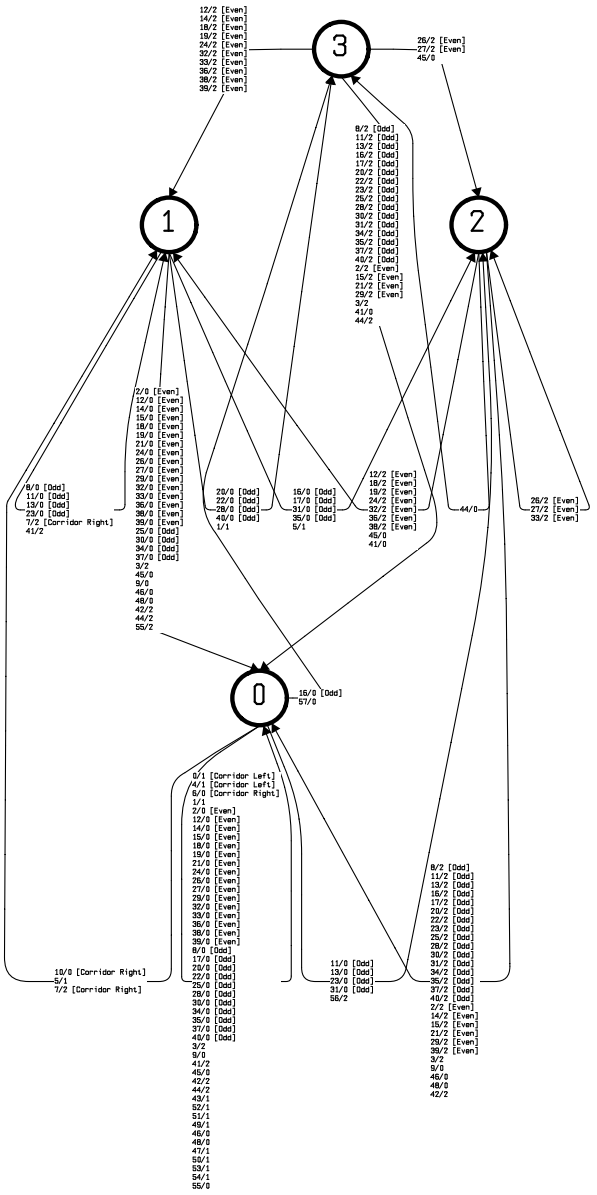

Figure 4.7: Extracted four-state FSA corresponding to an internal state-based policy, from the 5 parity, corridor 4 condition. State 0 and 1 code for corridor patterns that indicate the left goal position, state 2 and 3 do the same for the right goal position. All states are connected to all states.

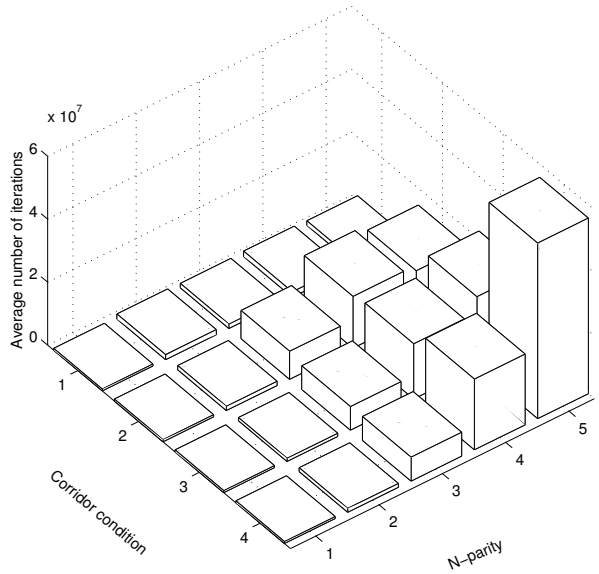

Figure 4.8: Average number of iterations needed to reach the termination criteria as a function of N-parity and corridor condition.

since such agents still rely on perception at the T-junction and normal behavior is not affected by the redundant internal states, we can still classify them as perception-based.

4.3.3 Time needed to reach the termination criteria

Figure 4.8 shows the average number of iterations until the termination criteria were reached for each experimental condition, plotted in the same spatial layout as figure 3. As expected, the more difficult the condition is both in terms of perception and in terms of internal state (high N-parity and high corridor condition), the higher the average number of iterations. If the parity pattern is easy, learning takes a relatively short time because the perception-based policy can easily be learned. Conversely, if the corridor pattern is easy, learning takes a relatively short time because the internal state-based policy can easily be learned. A two-way analysis of variance of these data supports these conclusions. The results indicate that there is a significant main effect (F=5.804, p=.000), significant effects of both N-parity (F=4.092, p=.004) and corridor condition (F=8.874, p=.000), as well as a significant interaction effect (F=3.552, p=.010).

Figure 4.9 shows the average number of iterations per N-parity condition for corridor condition 3, and sorted by the kind of policy converged upon (similar findings are obtained for other corridor conditions). It shows that the average number of iterations needed to learn a perception-based policy increases rapidly with higher N-parity. For $N = 4$ and $N = 5$, the internal state-based policy, which the agent can also converge upon, takes much less time to learn than the perception-based policy. These findings

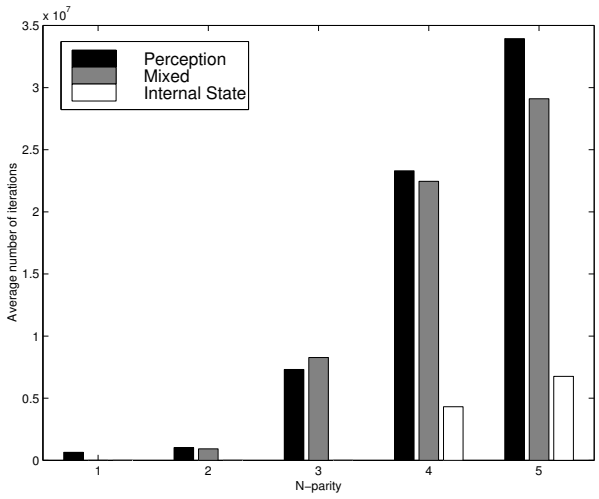

Figure 4.9: Average number of iterations per N-parity condition for corridor condition 3, sorted by the kind of policy converged upon.

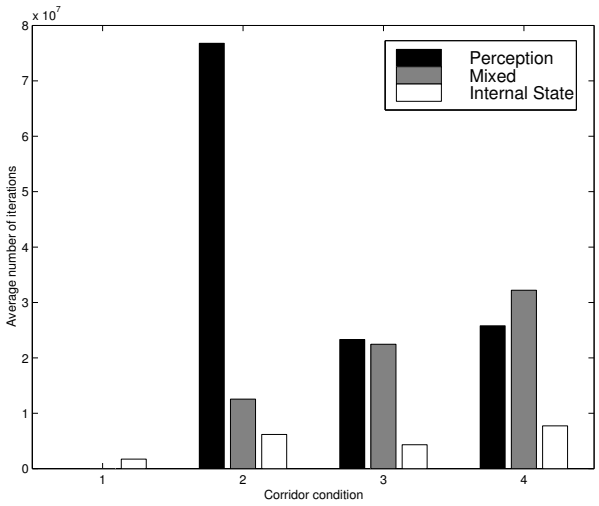

Figure 4.10: Average number of iterations per corridor condition for N-parity condition 4, sorted by the kind of policy converged upon

suggest that also in terms of learning times, an internal state-based policy can be easier to learn than a perception-based one as the pattern recognition problem associated with perception becomes more difficult.

Figure 4.10 shows the average number of iterations per corridor condition for N-parity condition 4, and again sorted by the kind of policy converged upon. Here the results are less clear. The graph does suggest that there is a positive correlation between difficulty of the pattern in the corridor and average learning time of an internal state-based policy. However, the average learning time for a perception-based policy is never shorter than those for the internal state-based policies.

Even though there are not enough data points to statistically confirm these results (the "average" learning time of the perception-based policy in corridor condition 2 is even made up of just one run), similar results are obtained in other N-parity conditions, and they are sufficiently clear to call for an explanation. An explanation may be found by considering the learning algorithm. The learning algorithm is based on gradient descent. This means that the learning process can be described in terms of the solution in weight space being attracted to one of the local or global optima. Being in the basin of attraction of one optimum does not necessarily say very much about the time until that optimum is finally reached. Here it may be the case that even if the conditions are such that the network will usually end up in the basin of attraction of a perception-based solution, the time until the optimum is reached is longer than if the networks ends up in the basin of attraction of an internal state-based policy. Thus, a preference for a perception-based policy in a certain condition does not mean that this policy will necessarily be learned much faster than the alternative, internal state-based policy. This suggests that neither learning times nor the network's own decisions which policy to adopt should *by themselves* be taken as the only or unequivocal answer to the question which kind of policy is "easiest to learn".

4.4 Discussion

In the simulation experiments of agents in a Markovian reinforcement learning task presented in this chapter, there is a trade-off between perception and internal state. If the pattern recognition problem associated with perception becomes too hard, agents develop internal states, called supportive states, that reduce the reliance on perception. Conversely, if the development of internal state is made more difficult, agents rely on perception more often. Of course, the environments used in this chapter were constructed in a way that made developing supportive states possible and, in many cases, even relatively easy. However, it can be argued that in many reinforcement learning tasks there are temporal regularities that can be exploited by internal state-based policies. In any case, this work presents a proof of principle that argues against the tendency in the reinforcement learning community to consider the use of perception-based or memoryless policies as a given when the task is Markovian.

Obviously, the general conclusions still hold in non-Markovian tasks. In non-Markovian tasks too there is a trade-off between perception and internal state: when the observation vector in principle provides an unambiguous state representation, but

the pattern recognition problem associated with the observation is severe, it may be easier to use internal state.

The argument may even be generalized beyond reinforcement learning, for example to supervised learning tasks. Even when solving a task can in principle be based on direct input-output mapping, if there are temporal regularities in a task, exploiting these temporal regularities may make learning easier. Indeed, it might in some cases yield more robust systems, for instance when the system's inputs are very noisy so the pattern recognition process is unreliable. In the context of neural networks, it means that even if a feedforward network could be used to solve a task, it might sometimes be more efficient or effective to use recurrent neural networks.

Looking at a trade-off like the one investigated in this chapter has interesting methodological advantages. First of all, a trade-off proves that the fact that under certain conditions one kind of policy is better than the other is not an "artefact" caused by a strong bias in the learning system to learn that kind of policy. After all, in other conditions the other kind of policy is better. In general, a single finding that one specific kind of policy or learning algorithm is better than another one for one specific test problem can be sensitive to details of the test problem and the learning algorithm. This makes it hard to draw general conclusions that are valid in other settings. When one demonstrates a trade-off this is less of a problem, because an array of findings is considered rather than a single finding, and the focus is on the general shape of the trade-off surface rather than on a single difference.

In a way, the trade-off shows that the tendency to learn one or the other kind of policy is "under the control" of the parameters that are varied, in this case the relative difficulties of perception and internal state. If the details of the learning system, learning algorithm, and test problem were varied, this would probably not change that qualitative result. For instance, it is likely that if Q-learning was implemented using complete backpropagation through time (Rumelhart et al., 1986) rather than ordinary backpropagation, or if a Long Short-Term Memory network (Hochreiter & Schmidhuber, 1997) was used rather than an Elman network (see chapter 6), the agent would have converged to an internal state-based policy in more conditions. The position of the trade-off surface would have shifted, but we would still have the qualitative result of the trade-off surface.

Therefore, the finding of a trade-off suggests that the trade-off is also important in other settings that share these parameters. Thus, even though the usefulness of demonstrating the trade-off is limited in the sense that we may not be able to say *where* exactly the transition from one kind of policy to the other will occur in other settings (because of differences in architecture, learning algorithm, and learning task), we can safely predict *that* there will be such a point. This is useful because it shows that one should take this trade-off into account whenever either one of the parameters of the trade-off is varied.

In any case, finding a trade-off like the one investigated in this chapter amounts to identifying factors that play an important role in the learning task. In this way, it contributes to our understanding of reinforcement learning tasks and the corresponding suitable learning systems.

Chapter 5

Reinforcement learning in POMDPs with Advantage(λ) learning and Elman networks

Summary

This chapter investigates the same basic architecture as used in the previous chapter, but now for the problem class of Partially Observable Markov Decision Processes. The main technical contribution, compared to the previous chapter and to work by other authors, is a new algorithm called Advantage(λ) learning. Another main focus of this chapter lies in the behavioral and mechanical analysis of the agents.

5.1 Introduction

In the previous chapter, the application of reinforcement learning agents controlled by recurrent neural networks to Markov Decision Processes (MDPs) was investigated. It was shown that even in that case, where internal states are not strictly necessary, internal states can still be useful, because they may make learning easier. In the current chapter, the case is considered where internal states *are* strictly necessary: Partially Observable Markov Decision Processes (POMDPs). Thus, we now start to consider the particular class of problems which is the main focus of this thesis, and which is why we turned to recurrent neural networks in the first place. The networks must learn to induce internal states that compensate for the insufficient information in the observations.

Two basic tasks (and variations on them) are investigated. The first is a continuous control task, a variation of the pole balancing or inverted pendulum control problem. The second is a discrete maze navigation task, of a similar type as the one investigated in the previous chapter. These tasks allow us to investigate the validity of

the approach of using recurrent neural networks that approximate value functions for solving different types of POMDPs, and the strengths and weaknesses of this approach.

Another focus of this chapter is an investigation of *how* the recurrent neural networks solve particular POMDPs, and what the resulting agents' behavior is in the face of unexpected changes in the environment. Among other things, this investigation allows us to make a connection to classical learning studies in psychology, and make some observations regarding the classical cognitive science concept of the *cognitive map* (Tolman, 1948), which has recently been re-examined in the adaptive behavior literature (Mataric, 1991; Clark, 1997).

The main technical contribution of this chapter, compared to the previous chapter and to work by other authors, is the particular reinforcement learning algorithm that is used. Instead of Q-learning, now Advantage learning (Harmon & Baird, 1996; Baird, 1999) is used. To the best of my knowledge, this is the first time Advantage learning is applied to partially observable reinforcement learning problems. Furthermore, Advantage learning is extended with eligibility traces, yielding the new algorithm called Advantage(λ) learning. Experimental results are presented to show the benefit of using Advantage(λ) learning.

The next section describes the modifications in the learning algorithm. Section 5.3 contains the description and results of the pole balancing experiments. Section 5.4 contains the description and results of the maze navigation experiments. The last section presents conclusions.

5.2 Advantage learning with Elman networks

5.2.1 Architecture of the recurrent neural network

The network architecture used in this chapter is the same as in the previous chapter, see section 4.2.2 for details. The only difference is that the output unit is now linear. The activation of the output unit is simply its net input:

$$y_k(t) = g(net_k(t)) = net_k(t) = \sum_h w_{kh} y_h(t). \tag{5.1}$$

The more important difference with the previous chapter is the particular reinforcement learning algorithm that is used to train this Elman recurrent neural network. In this chapter Advantage learning (Harmon & Baird, 1996; Baird, 1999) is used rather than Q-learning. Furthermore, Advantage learning is extended with eligibility traces, yielding the new algorithm called Advantage(λ) learning.

5.2.2 Advantage learning

Advantage learning (Harmon & Baird, 1996; Baird, 1999) is a reinforcement learning (RL) algorithm derived from Advantage updating (Baird, 1994), which in turn was designed as an improvement on Q-learning for continuous-time RL. In continuous-time RL, values of adjacent states typically differ by only small amounts, relative to the possible overall variance of values. This means that the optimal Q-values of different

actions in a given state, from which the policy is directly derived, differ by only small amounts. These differences, then, can easily get lost in the noise, especially when we use function approximators, such as neural networks.

Advantage learning remedies this problem by artificially decreasing the values of suboptimal actions in each state. The differences between the values of different actions in each state are thus greater than in Q-learning, and less likely to get lost in the noise. Doya (2000) shows that Advantage updating (so presumably Advantage learning as well) can be understood as one approach to deriving an efficient control policy for continuous-time systems using the estimated *gradient* of the Q-value function.

Even though Advantage learning was originally designed for continuous-time RL, it can be fruitfully used for discrete-time RL as well, as we will see in this chapter and the next chapters. Note that the same problem of small differences between the values of adjacent states applies to any RL problem with long paths to rewards, and therefore the same solution might be beneficial.

Furthermore, Advantage learning may have a special edge over Q-learning when applied to POMDPs. In POMDPs, in contrast to MDPs, the state of the environment is not directly given to the agent in the form of the current observation. Therefore, the agent cannot be as certain about both the current state and the next states, and for this reason, very precise value estimation may be difficult. However, it may be easier to estimate something like "relative Q-values" in each state, which is what Advantage learning effectively does. This does not require that values are estimated exactly, but only that better actions have higher values. As long as the best action in each state has the highest relative value, this will still lead to an optimal policy.

5.2.3 Bellman equation and learning algorithm

In Advantage learning, the value of an environmental state is defined as

$$V^*(s) = \max_a A^*(s,a). \tag{5.2}$$

The optimal Advantage $A^*(s,a)$ for each action a in state s is defined as

$$\begin{aligned} A^*(s,a) &= V^*(s) + \frac{E\{r(t+1) + \gamma V^*(s(t+1)) \mid s(t)=s, a(t)=a\} - V^*(s)}{\kappa} \\ &= V^*(s) + \frac{\sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')] - V^*(s)}{\kappa} \end{aligned} \tag{5.3}$$

where, as usual, γ is a discount factor in the range $[0,1]$. For an optimal action, the second term is zero, meaning that the value of this action is the value of the state. For suboptimal actions, the second term is negative. The size of the second term depends on κ, a constant scaling the difference between values of optimal and suboptimal actions (κ replaces $K\Delta t$ of the original formulation as found in Harmon & Baird, 1996; Baird, 1999). Equation 5.3 is Advantage learning's equivalent of the Bellman equation, compare equation 3.12. Note that when $\kappa = 1$, this Bellman equation reduces to Q-learning's Bellman equation. Thus, we can view Q-learning as a special case of Advantage learning.

To implement Advantage learning, the righthand side of equation 5.3 can be viewed as the target output for the Advantage of the executed action at the current timestep. As in Q-learning, actual, sampled experiences in the environment replace the true expected values, and the system's own current approximations to $V^*(s)$ and $V^*(s')$ are used. Thus, the update is based on the immediate reward received after executing action a, and the maximally attainable Advantage value in the next state, estimated by the system itself. This yields the temporal difference error $E^{TD}(t)$ at time t, which basically computes the difference between the estimated righthand side and lefthand side of equation 5.3:

$$E^{TD}(t) = V(s(t)) + \frac{r(t+1) + \gamma V(s(t+1)) - V(s(t))}{\kappa} - A(s(t), a(t)). \tag{5.4}$$

Consider the case where Advantage learning is implemented using a function approximator, as in this thesis and in most other work with Advantage learning. The tabular representation can be viewed as a special case of function approximation. Let us say the function approximator has parameters or weights w_{im}. At each timestep t, the weight update prescribed by direct Advantage learning[1] is

$$\Delta w_{im}(t) = \alpha E^{TD}(t) \frac{\partial A(s(t), a(t))}{\partial w_{im}} \tag{5.5}$$

for all parameters w_{im}. When we use a tabular representation, each parameter w_{im} simply corresponds to one table entry $A(s, a)$ for each state-action pair, and $\frac{\partial A(s(t),a(t))}{\partial w_{im}}$ is 1 for the state and action of time t, and 0 otherwise.

5.3 Advantage(λ) learning

5.3.1 The λ-return

In its simplest form, Advantage learning, like other temporal difference learning algorithms, performs back-ups based on the *immediate reward plus the discounted value of the next state*. This is called the 1-step return, or $R^{(1)}(t)$. Advantage learning's temporal difference error can be written as

$$\begin{aligned} E^{TD}(t) &= V(s(t)) + \frac{r(t+1) + \gamma V(s(t+1)) - V(s(t))}{\kappa} - A(s(t), a(t)) \\ &= V(s(t)) + \frac{R^{(1)}(t) - V(s(t))}{\kappa} - A(s(t), a(t)). \end{aligned} \tag{5.6}$$

It is natural, as in other TD-based learning algorithms, to attempt to speed up Advantage learning by updating state-action values not just on value information from one

[1] Certain RL algorithms based on temporal difference learning, such as Q-learning and Advantage learning, can in principle lead to divergence when used in conjunction with function approximators such as neural networks (e.g. see Harmon & Baird, 1996; Baird, 1999; Sutton & Barto, 1998). It is unclear at this point how serious this problem is, and it did not appear to be a problem in the work reported in this thesis. If it turns out to be a problem in practical cases, one could use the safer "residual" or "residual gradient" versions of Advantage learning (Harmon & Baird, 1996; Baird, 1999), rather than the direct version this chapter is concerned with.

timestep later, but also from multiple timesteps later. An obvious candidate is doing updates based on the λ-return rather than the 1-step return (see section 3.5.2.4). As explained in section 3.6.3.10, this could be especially beneficial for the class of problems that this chapter (and the remainder of this thesis) is concerned with, partially observable problems. In that case the system cannot be sure about current and next states, and therefore value estimations, which are associated with states, are less reliable than in completely observable problems. Using the λ-return rather than the 1-step return makes the system less vulnerable to errors in individual value estimations, because a weighted sum of value estimations associated with multiple states and of actually received rewards is used.

Thus, we want to include Advantage learning into the TD(λ) family of algorithms (Sutton, 1988). In the case of off-policy methods (Sutton & Barto, 1998), such as Advantage learning, updates are done based on the λ-return R^λ(t) defined in the following way (Watkins, 1989):

$$\begin{aligned} R^\lambda(t) =& (1-\Lambda(t+1))R^{(1)}(t) + \Lambda(t+1)(1-\Lambda(t+2))R^{(2)}(t) \\ &+ \Lambda(t+1)\Lambda(t+2)(1-\Lambda(t+3))R^{(3)}(t) + \dots \end{aligned} \tag{5.7}$$

where

$$\Lambda(t) = \begin{cases} \lambda & a(t) = \arg\max_a A(s(t),a) \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

where $0 \leq \lambda \leq 1$ is a constant that weighs the importance of long-term rewards as opposed to short-term rewards. If $\lambda = 0$ (Advantage(0)-learning or plain Advantage learning), this reduces to standard Advantage learning's 1-step return. If $\lambda = 1$ (Advantage(1)-learning), on the other hand, the λ-return is based only on actual rewards obtained during the episode. Thus, in that case the λ-return is not based on estimated values; in other words, it does not bootstrap. In the intermediate cases, $0 < \lambda < 1$, an average of multiple-step returns is taken, weighted in the particular way of TD(λ) algorithms.

As with Q(λ)-learning, the λ-return is truncated at the point where an exploring action is chosen, i.e. when $a(t) \neq \arg\max_a A(s(t),a)$. This corresponds to the notion that rewards obtained after that point no longer reflect the value of the currently estimated best policy. After all, in off-policy methods like Q-learning and Advantage learning we wish to learn about the best policy, and not about a policy with exploring, possibly suboptimal actions.

5.3.2 Eligibility traces

As with Q-learning and other TD-learning algorithms, the update of a value based on the λ-return can be accomplished by maintaining a simple scalar measure, called the *eligibility trace*, for each parameter of the reinforcement learning agent (Sutton, 1988, 1989; Watkins, 1989). Thus, there is one eligibility trace e_{im} per parameter w_{im}. The eligibility trace $e_{im}(t)$ at time t for Advantage learning is the same as that for Q-learning, and it is defined as

$$e_{im}(t) = \gamma\Lambda(t)e_{im}(t-1) + \frac{\partial A(s(t),a(t))}{\partial w_{im}}. \tag{5.9}$$

As in other algorithms that use eligibility traces, the eligibility trace can be interpreted as information that says to what extent the corresponding parameter w_{im} can be held "responsible" for rewards obtained later on. If an exploring action is taken then $\Lambda(t) = 0$, in other words, the eligibility traces are reset to 0. $e_{im}(0)$ is 0 for all parameters.

The update rule for Advantage(λ) learning, implemented using eligibility traces, then becomes

$$\Delta w_{im}(t) = \alpha E^{TD}(t) e_{im}(t) \tag{5.10}$$

for all parameters w_{im}. In the case of Elman neural networks as the function approximator and using standard 1-step backpropagation, the partial derivatives of the Advantage value with respect to the weights in equation 5.9, which are used to update the eligibility traces, are computed as follows. As in the previous chapter, the current state is approximated by the current observation plus the Mealy internal state. That is, $A(s(t), a(t))$ is approximated by $A(o(t), z(t), a(t))$. $A(o(t), z(t), a(t))$ is equal to the activation of the output unit, so

$$\frac{\partial A(o(t), z(t), a(t))}{\partial w_{im}} = \frac{\partial y_k(t)}{\partial w_{im}}. \tag{5.11}$$

For the weights w_{kh} from the hidden layer to the output layer in the Elman network, the partial derivatives are

$$\frac{\partial y_k(t)}{\partial w_{kh}} = g'(net_k(t)) y_h(t) = y_h(t) \tag{5.12}$$

and the partial derivatives with respect to the weights w_{hm} to the hidden layer are

$$\frac{\partial y_k(t)}{\partial w_{hm}} = w_{kh} g'(net_k(t)) f'(net_h(t)) y_m(t) = w_{kh} y_h(t)(1 - y_h(t)) y_m(t). \tag{5.13}$$

where, as before, m can be an observation unit, an action unit, or a context unit. Note that internal state is treated in the same way as observation and action, as far as eligibility is concerned. That is, the weights from the context layer to the hidden layer have the same type of eligibility trace updates as, for instance, the weights from the observation layer to the hidden layer. This makes sense because the internal states from the past are "responsible" for current rewards in the same way as observations and actions. This weight update scheme effectively does standard 1-step backpropagation in the Elman network, attempting to minimize errors based on the λ-return.

5.3.3 The forward and backward view of Advantage(λ) learning

Advantage(λ) learning implemented using eligibility traces is the backward view (Sutton & Barto, 1998, and see section 3.5.2.4) of Advantage(λ) learning. At each point in time it looks back to the agent's past, using the eligibility traces, to estimate which parameters used in the past were responsible for the current situation. Advantage(λ) learning can also be implemented by straightforwardly using the λ-return in the target output. This is the forward view, because at each point in time it looks forward in time for rewards and values in the future.

Appendix B contains the proof of equivalence of the forward and backward views of Advantage(λ) learning with function approximators. More precisely, it shows that offline Advantage learning with the same eligibility traces as Q-learning, and on the basis of the one-step return (the backward view), leads to the same weight updates as offline Advantage learning without eligibility traces on the basis of the λ-return (the forward view). This is important because it is not obvious that Advantage learning can be combined with eligibility traces in the same way as Q-learning and lead to the same desired result, namely back-ups based on the λ-return. It is not obvious because the corresponding Bellman equations differ for the two algorithms, and the known results for Q(λ) and TD(λ) depend on the corresponding Bellman equations.

5.3.4 Related work

In chapters 2 and 3, a number of studies combining recurrent neural networks and reinforcement learning algorithms have been discussed. As explained in chapter 3, the work that is most closely related to the work of that and this chapter is that of Lin & Mitchell (1993, 1992). They combine Q(λ)-learning with Elman networks. Several differences with this thesis' work were already discussed in the previous chapter. An additional difference is of course that in this chapter Advantage(λ) learning is used rather than Q(λ)-learning. Another difference worth mentioning in the context of this chapter is that they use a forward view implementation of Q(λ)-learning, rather than a backward view implementation using eligibility traces, as is done in this chapter. For this reason, they have to postpone updates until the end of episodes, yielding a learning algorithm that is not local in time.

5.4 Partially observable pole balancing

5.4.0.1 The task

The pole balancing or inverted pendulum task (see figure 5.1) is a classical continuous control problem. A large number of researchers have studied variations of this task using machine learning techniques (Michie & Chambers, 1968; Barto et al., 1983; Anderson, 1987; Lin & Mitchell, 1993; Schmidhuber, 1991c; Moriarty & Miikkulainen, 1996; Meuleau et al., 1999; Doya, 2000), giving it the status of a benchmark problem. For this reason alone, it is interesting to see how our reinforcement learning recurrent neural network approach performs on it. Furthermore, it is the first continuous state and continuous time problem investigated in this thesis, and it is interesting to see if and how our approach copes with the corresponding problems, such as the problem of effective generalization over the state space.

In the pole balancing task, an agent must balance an inherently unstable pole, which is hinged to the top of a wheeled cart that travels along a track, by applying left and right forces to the cart (see figure 5.1). The state of the environment is defined by the cart position x, the pole angle θ, the cart velocity $\dot{x}$, and the pole's angular velocity $\dot{\theta}$. See e.g. Anderson (1987), Lin and Mitchell (1992) for the equations describing the physics of the pole balancing task. The task requires fairly precise control to solve it, i.e., to balance the pole indefinitely. Another difficulty in the

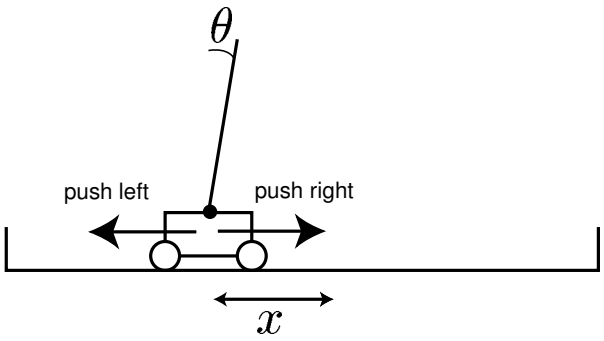

Figure 5.1: The pole balancing task.

reinforcement learning version investigated in this chapter is the sparseness and strong delay of the reward signal. The only reward signal is -1 if the pole falls past $\pm 12°$ or if the cart hits either end of the track. These are also the only points where the episode ends and a new one starts. At the start of each episode, the cart is randomly positioned on the track: $-1 < x < +1$ (the track runs from -2.4 to 2.4), and the pole is also positioned randomly: $-3° < \theta < +3°$. Time is "discretized" by equating one neural network/learning algorithm iteration with .02 seconds of simulated physical time, and by using difference equations updated using the Euler method for the simulated physics.

Most studies have considered only the standard, completely observable version of pole balancing. In the completely observable version, the agent's observation consists of all state variables, i.e. x, θ, $\dot{x}$, and $\dot{\theta}$. Here a partially observable version is considered, as in Lin and Mitchell (1993), Schmidhuber (1991c), Gomez and Miikkulainen (1999), Wieland (1991), Meuleau et al. (1999), in which the agent cannot observe the state information corresponding to the cart velocity $\dot{x}$ and pole angular velocity $\dot{\theta}$. The agent has to learn to approximate this information using its internal state, in this case its recurrent activations, in order to solve the task. This is continuous information, making a recurrent neural network approach seem a more appropriate solution technique than, for instance, a memory bits approach, a U-tree approach, or an FSA approach (but see Meuleau et al., 1999).

5.4.1 The experiment

An Elman network with 10 hidden units and 10 context units was used. In this particular experiment, half of the hidden units were linear units instead of sigmoid units. This was done to give the output units linear access to the observation variables. It improved performance considerably in terms of learning times, compared to using only sigmoid units, but it is not essential for getting the same qualitative results. The number of observation units was 2: one for the cart position x and one for the pole angle θ. Both inputs were normalized to the range $[-1, 1]$. Furthermore, as in the previous chapter, there are additional input units coding for the action, one

for action "push left" and one for action "push right". For each of the two actions, the Advantage value is computed by setting the corresponding action input unit to 1 and the other to 0, and computing the activation of the single output unit, which codes directly for the Advantage value. The discount rate γ of reinforcement learning was .98, and λ was .9. In this experiment the focus was mainly on the difference in performance between Q-learning and Advantage learning in the case of non-Markovian tasks. Therefore, two experimental conditions were compared: $\kappa = 1$ (Q(λ)-learning) versus $\kappa = .2$ (Advantage(λ) learning). In the test problem discussed later, the discrete maze navigation problem, we will have a closer look at the possible benefit of eligibility traces, i.e. λ is varied as well.

As in the previous chapter, the agent explores using a stochastic action selection mechanism, namely Boltzmann exploration, which computes the probability $p(s, a)$ of each action a in state s according to:

$$p(s,a) = \frac{e^{A(s,a)/\tau}}{\sum_{m=1}^{M} e^{A(s,a_m)/\tau}} \tag{5.14}$$

where M is the number of possible actions and τ is the temperature of the action selection mechanism.

Because Advantage values and temporal difference errors are different for different values of κ, it is quite possible that the optimal temperature of action selection τ as well as the optimal learning rate α is different for the two conditions. For this reason, a range of parameter values for both α and τ was tried. The range for both $\kappa = 1$ and $\kappa = .2$ was determined using preliminary experiments that suggested reasonable parameter ranges. For both conditions, going much outside these ranges decreases performance; it is therefore not very meaningful to compare the two experimental conditions on exactly the same parameter ranges. In general, Q(λ)-learning ($\kappa = 1$) requires larger learning rates α and smaller temperature parameters τ than Advantage(λ) learning. This is probably because differences between estimated values, both between consecutive states and between action actions within one state, are often very small in Q(λ)-learning, so that they disappear in the noise for smaller learning rates and larger temperatures.

10 test runs were performed for each combination of parameter values. The termination criterion for each run had two requirements. First, the running average of the balancing time during learning, using stochastic action selection, had to exceed 80 iterations. Second, using greedy action selection, the agent had to be able to balance the pole indefinitely. The latter criterion was tested "offline", every 50,000 iterations. The first criterion ensures that solutions are at least somewhat robust, i.e., minor weight changes probably do not affect performance that much. The maximum number of iterations allowed for each run was 25,000,000, which is more than enough to learn the task with reasonable parameter settings. If a run did not reach the termination criteria before this maximum number of iterations, it was considered a failure.

Figure 5.2a shows the number of runs (out of 10) that reach the termination criteria for Q(λ)-learning ($\kappa = 1$), for different parameter values of α and τ. Figure 5.3a shows the same information for Advantage(λ) learning ($\kappa = .2$). Figures 5.2b and 5.3b show the average number of iterations needed to reach the termination criteria, for successful

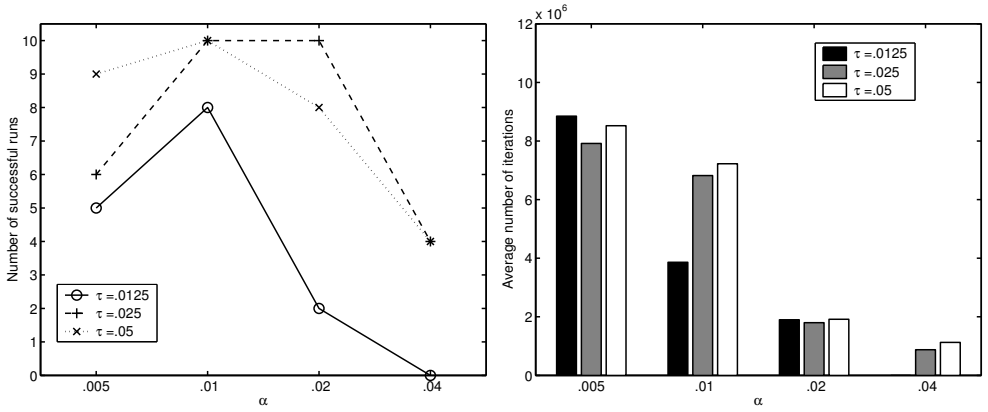

Figure 5.2: Fig. a (left). Number of successful runs (out of 10) in the pole balancing task as a function of learning rate α and temperature of action selection τ for Q(λ)-learning ($\kappa = 1$). Fig. b (right). Average number of iterations until success, given that the success criteria are reached, for the same conditions.

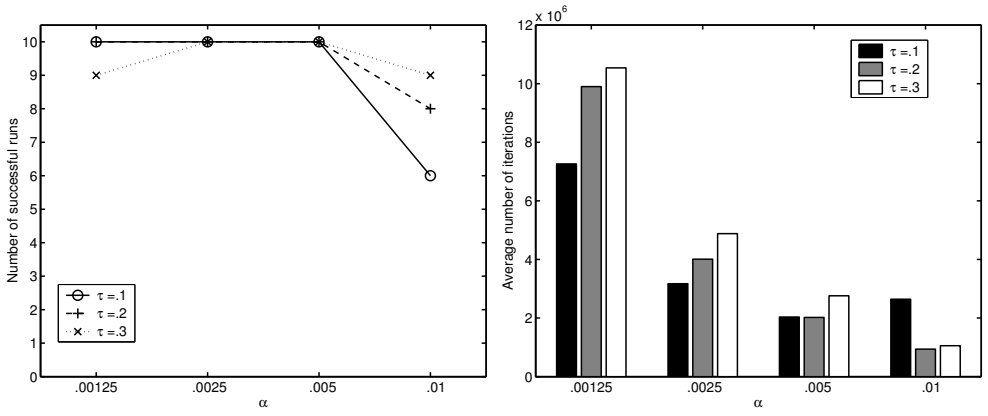

Figure 5.3: Fig. a (left). Number of successful runs (out of 10) in the pole balancing task as a function of learning rate α and temperature of action selection τ for Advantage(λ)-learning ($\kappa = .1$). Fig. b (right). Average number of iterations until success, given that the success criteria are reached, for the same conditions.

runs only. The clearest result that can be seen from these graphs is that Advantage(λ) learning reaches the success criteria in many more cases than Q(λ)-learning. It is therefore more robust, at least in this particular context.

These results are consistent with the original ideas behind Advantage learning and with previous results, which essentially say that in continuous time problems Advantage learning should do better than Q-learning, because optimal Advantage values within one state differ much more than optimal Q-values, making it easier to learn them. What is new about this result is that Advantage learning works well in partially observable tasks as well, and in that context evidently has a similar advantage over Q-learning. Another new result is that apparently eligibility traces can be successfully combined with Advantage learning. That is, Advantage(λ) learning is a feasible reinforcement learning algorithm, also in difficult reinforcement learning problems such as partially observable problems with strongly delayed reward. The learning algorithm allowed the recurrent neural network to successfully organize its internal state space and internal state transitions such that it approximated sufficiently well the continuous information, cart velocity and pole angular velocity, that was missing from the observations but that was needed for solving the task.

5.5 Maze navigation task

We now turn to a discrete partially observable task, a maze navigation task. The maze is depicted in figure 5.4, together with the agent and its sensors. The agent has an orientation, and its sensors give very limited information about its immediate surroundings. Each sensor detects walls (0) versus open space (1) at a specific location relative to the agent. There are only 28 observations for 173 states, and many states with the same observation require different actions (e.g. T-junctions). The actions are the same as in the previous chapter, i.e. they consist of actions "move forward", "turn left and move in the new direction", and "turn right and move in the new direction". As in the previous chapter, the agent takes steps of 2 grid cells at a time, such that its sensors do not provide information about states one action beyond the current position. If the agent attempts to move through a wall, it stays in the same location, but turning is possible. Even though the observations are highly ambiguous, still some generalization over the observations is possible: e.g. the agent should learn that an action toward a certain direction only makes sense if the sensors indicate that there is open space in that direction.

The objective is to get to the goal position in the upper right corner. At this point, the only reward, $r = 1$, is obtained and the episode ends. If the agent fails to reach the goal in 300 actions, the episode ends as well. A new episode starts with the agent starting at a random location anywhere in the maze. Since the agent starts each episode at a random location, it cannot simply learn a fixed route "by heart", for example a feedforward control policy which simply executes a series of actions in a fixed order. Nor can the agent tell immediately where in the environment it is based on the first observation, which would be the case in MDPs. Instead, the agent must figure out in the beginning of each episode where it is, based on a sequence of observations

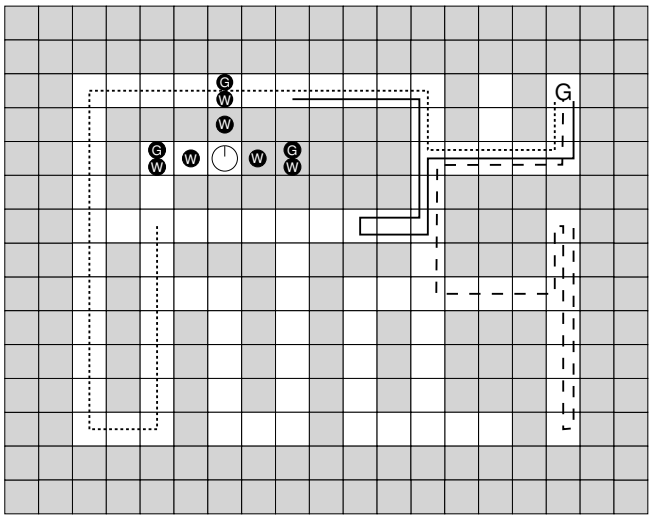

Figure 5.4: The partially observable maze. The agent is depicted, oriented to the north, together with its sensors for walls (W) and the goal (G). The goal location G is in the upper right corner. Many states are ambiguous with respect to observations, i.e. many position/orientation combinations give rise to the same observation. Three typical paths are depicted when the agent starts from different random starting positions.

it sees when it executes actions. Most of these observations are highly ambiguous—corresponding to "corridor", "corner", or "T-junction", for instance, which could be anywhere—but the particular sequence of observations given particular actions should be sufficient to eventually determine where it is. From then on the agent should proceed to the goal more or less directly, while still being able to disambiguate ambiguous observations along the way. Note that all this has to be learned based on a very sparse training signal, the reward obtained after finally reaching the goal.

5.5.1 The experiment

A similar neural network architecture was used as in the pole balancing task, but now with 3 action input units, 9 observation units, and 10 sigmoid hidden and context units. The same Boltzmann exploration mechanism was used as before. Again an experimental comparison was made between Q(λ)-learning ($\kappa = 1$) and Advantage(λ) learning ($\kappa = .1$), for different values of α and τ. For this comparison, λ was set to .9. For each set of parameters, 10 runs were done. The success criteria for each run were as follows. The running average of the proportion of episodes in which the agent reaches the goal before the time-out value of 300 actions, during learning and using its exploration mechanism, must be higher than .95. Second, for those episodes where the agent reaches the goal, the average number of actions in the episode must be below 40 actions. The latter criterion means that the agent cannot take many excess actions.

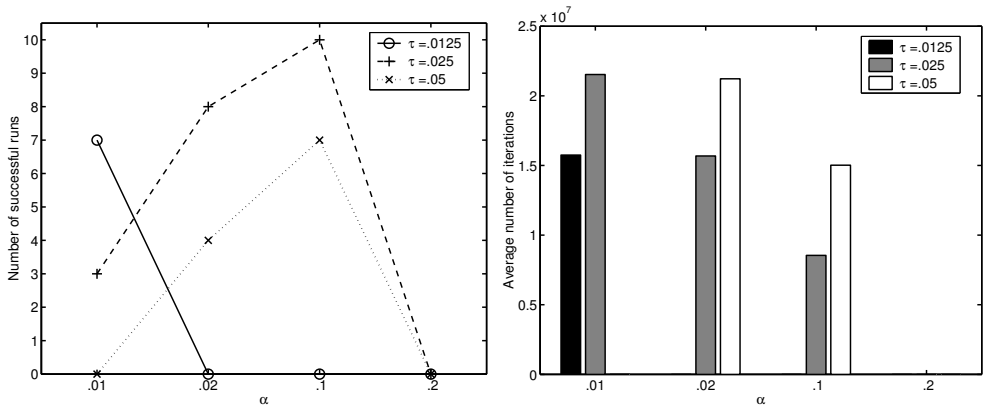

Figure 5.5: Fig. a (left). Number of successful runs (out of 10) in the maze navigation task as a function of learning rate α and temperature of action selection τ for Q(λ)-learning ($\kappa = 1$). Fig. b (right). Average number of iterations until success, given that the success criteria are reached, for the same conditions.

To give an idea about what these numbers mean, at the beginning of a run, before any learning is done, the proportion of episodes where the agent reaches the goal thanks to random action selection is around .4, and the average number of actions if it reaches the goal is 130.

Figures 5.5 and 5.6 show the results, both in terms of the number of successful runs out of 10 test runs, and in terms of the time needed to reach success given that it was reached. These results are displayed for different combinations of parameters. It is apparent that, as in the pole balancing task, Advantage(λ) learning clearly outperforms Q(λ)-learning in both respects: it learns faster and more reliably. Advantage(λ) learning reaches the success criteria in many more cases than Q(λ)-learning. Advantage(λ) learning is, in this sense, more robust than Q(λ)-learning. In the one combination of parameter values in which Q(λ)-learning always reached the success criteria, $\alpha = .1$ and $\tau = .025$, it needed 2.34 times more learning iterations, on average, to reach the success criteria than the best parameter combination of Advantage(λ) learning, $\alpha = .01$ and $\tau = .05$.

Having obtained some evidence that Advantage(λ) learning is better than Q(λ)-learning for the tasks considered in this chapter, we now turn to the question how beneficial eligibility traces are in combination with Advantage learning. The goal is to investigate the hypothesis, expressed in the beginning of this chapter, which stated that Advantage learning should benefit from eligibility traces in the same way as other temporal difference based learning algorithms, and especially in partially observable tasks. In other words, we want to see if the new algorithm called Advantage(λ) learning can be justified by empirical results. This question is investigated by systematically varying the value of λ from 0 to 1, taking steps of .1. Recall that $\lambda = 0$ corresponds to plain Advantage learning. κ was held constant at .1. τ was fixed at .1. There is

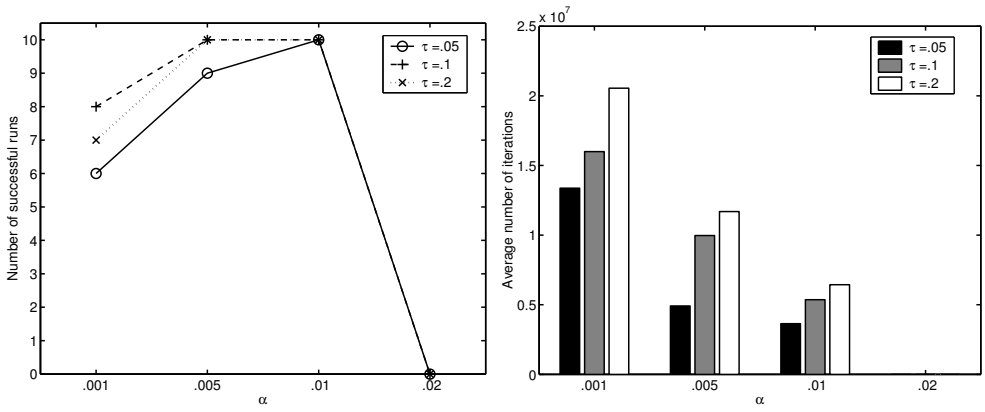

Figure 5.6: Fig. a (left). Number of successful runs (out of 10) in the maze navigation task as a function of learning rate α and temperature of action selection τ for Advantage(λ)-learning ($\kappa = .1$). Fig. b (right). Average number of iterations until success, given that the success criteria are reached, for the same conditions.

no need to vary τ in this case, because the optimal Advantage values are not different for different values of λ: varying λ just provides different ways to learn the same Advantage values. However, we do need to vary α again, because weight changes are different for different values of λ (specifically, weight changes will in general be larger for larger λ in tasks with sparse and strongly delayed reward such as this one), so different values of α might be optimal (see Sutton & Barto, 1998 who also vary α when investigating the effect of λ).

10 runs in the same maze navigation task were performed for each combination of α and λ. Figure 5.7 shows the number of successful runs, out of 10, for the different combinations of parameter values. It is apparent that it is very beneficial to use $\lambda > 0$. The optimal value of λ for this task seems to be 1 or close to 1. If we look at the runs in terms of the number of iterations needed to learn the task, for those runs in which the task was learned, we see that in that respect $\lambda > 0$ also does much better than $\lambda = 0$. Thus, for this task at least, Advantage(λ) learning seems to be a useful extension of Advantage learning.

Figure 5.4 shows characteristic behavior of a successful agent *after learning* (this is an agent from the $\kappa = .1$, $\lambda = .9$, $\alpha = .01$ condition). Sample trajectories are shown of episodes when the agent starts in different locations. In most cases, at the beginning of an episode the agent first seems to be wandering around without making clear progress to the goal. After a number of actions, however, the agent seems to "realize" where it is, judging from the fact that from then on it proceeds more less or directly to the goal, without going back and forth anymore. We could say the agent becomes "entrained" with the environment (Tani & Nolfi, 1998; Clark, 1997; Beer, 1995; Kelso, 1995) after the first number of actions that are needed to obtain a sequence of observations that disambiguate the initial states.

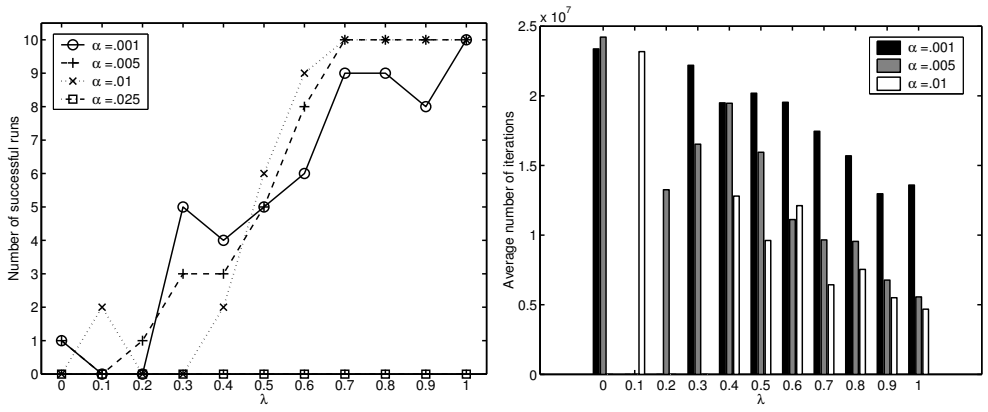

Figure 5.7: Fig. a (left). Number of successful runs (out of 10) as a function of λ for Advantage(λ)-learning ($\kappa = .1$). Fig. b (right). Average number of iterations until success, given that the success criteria are reached, for the same conditions.

It is important to note that even after the first initial entrainment phase where the agent figures out where it is, the agent does not always take the shortest route to the goal. In a small number of cases, it takes a suboptimal path, but never a really bad path. Such an agent has converged to a suboptimal but satisficing policy. This is to be expected when we use reinforcement learning methods with function approximators.

5.5.2 Cognitive maps in rats and robots

The maze experiment was repeated with the same architecture, but now starting from a fixed starting position, indicated in figure 5.8. The termination criteria were now 95% correct, and at most a running average of 25 actions to the goal, using the agent's online stochastic action selection. Advantage(λ) learning was used, with $\kappa = .1$, $\lambda = .9$, $\tau = .1$ and $\alpha = .01$. In 10 runs, the agent always reached these criteria within, on average, 4,458,522 iterations.

The agents always learned the shortest path from the starting position to the goal (see figure 5.8). Note that here there is no need for an initial entrainment phase at the beginning of an episode, because the agent can rely on starting always at the same position. However, it is interesting to now see how the trained agent will do when it is put in different starting positions, as in the previous experiment. No additional training was done; the aim here is to investigate the ability of the agent to cope with this unexpected change. Figure 5.8 shows a typical route taken by the agent. From other starting positions the agent also takes routes to the goal that are not always optimal, but that are always satisficing in that they lead via a reasonable trajectory to the goal. Similarly, if the agent is picked up in the middle of an episode and placed somewhere else, after an initial "entrainment" phase it once again takes a fairly direct path to the goal.

Thus, even though in this particular experiment the agent was never explicitly trained to display these capabilities, it still has them. This shows that the agent has not learned only a fixed route from the original starting position to the goal, but much more than that, and it generalizes fairly well over the entire environment. The reason it can do this is that during learning the agent has explored many different states and actions, and has apparently found reasonable approximations to the optimal Advantage values not just for the limited number of states and actions along its optimal path, but for most of the environment.

This type of problem, where an agent is put in a different place from where it "thinks" it is, and is then supposed to figure out where it is, is also known as the "lost" or "kidnapped" robot problem (e.g. Thrun, 2000; Vlassis et al., 2002). It is argued by Thrun (2000) that this problem calls for explicit representation of uncertainty by the agent, as is done in belief state estimation approaches (see section 3.6.2). However, we see in the current experiment that it is possible to have similar capabilities without explicit representation of uncertainty.

Next, with the same agent once again starting only from the original, fixed starting position, an obstacle was introduced in the optimal path. In terms of the agent's sensors, this obstacle looked like a wall. The obstacle and the path taken by the agent are depicted in figure 5.8. The agent first moves toward the obstacle. When it perceives the obstacle, however, it turns around and takes another path to the goal, thus showing a surprising ability to deal with unexpected changes in the environment and still reach the goal. The reason for this ability lies partly in the fact that early on in its training, the agent has learned that, in general, it is better not walk into walls but rather to turn around and go back. In this sense, the agent generalizes successfully over different states. It has learned this not because it was punished for bumping into walls (there is only a reward at the goal), but because you get to the goal faster and more often when you do not bump into walls. This preference causes the agent to simply turn around when it encounters the "unexpected" wall and quickly settle into a new route which takes the agent to the goal.

It should be noted that, depending on the particular agent and the location of the obstacle, the agent does not always pick the best alternative route to the goal; sometimes it makes a detour. In some cases the second route would also come across the obstacle, but it is rare for the agent not to reach the goal eventually, within the time limit. Thus, even though the agent does not have a perfect conception of what its world looks like, and it does not know or remember perfectly where the obstacle is, it nevertheless has an impressive ability to adaptively handle such unexpected changes in the environment. In this sense, the technique of reinforcement learning combined with recurrent neural networks leads to robust solutions to fairly difficult partially observable problems.

At this point it is interesting to consider again Tolman's (1948) work on navigating rats, discussed in chapter 2. He and his colleagues similarly investigated mazes with few perceptual cues, combined with "reinforcement learning"; i.e. behaviorist operant conditioning experiments where rats could find food at certain positions in the maze. Reviewing a large number of experiments, Tolman came to the conclusion that simple stimulus-response descriptions of the type favored by contemporary behaviorist theory did not suffice to explain the rats' abilities. Instead, the rats must have learned

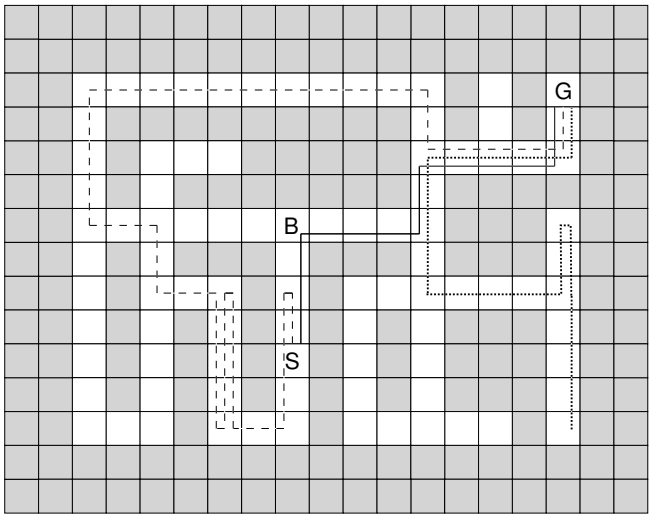

Figure 5.8: The partially observable maze in which the agent was trained to go from the starting position S to the goal G. The agent always learned the shortest path, indicated by the solid line. The dotted line indicates a typical route when the agent is put in a different starting position than S. The dashed line indicates the route taken when an obstacle is introduced in the position indicated by B.

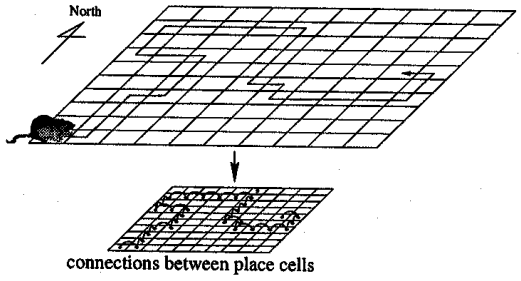

Figure 5.9: Typical model of the cognitive map as it is assumed to be implemented in neural structures. Note the isomorphic mapping from the real world to the neural structure. Adapted from Trullier & Meyer (1998).

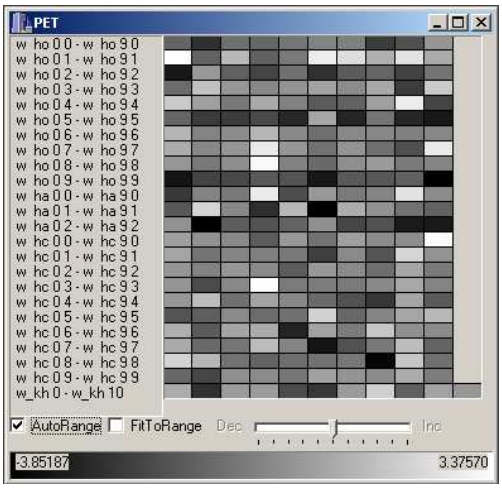

Figure 5.10: The weights of the Elman network after learning. A brightness coding is used, in which lighter means a higher value.

"cognitive maps", allowing them to cope with unexpected changes in the environment almost immediately without any additional training and to generalize over mazes in a smarter way than can be explained by pure fixed route learning. Tolman (1948) informally defines a cognitive map as follows (p. 193):

> In a comprehensive map a wider arc of the environment is represented, so that, if the starting position of the animal be changed or variations in the specific routes be introduced, this wider map will allow the animal still to behave relatively correctly and to choose the appropriate new route.

According to this definition of a cognitive map, and considering the simulation experiments presented, the artificial agent must contain a cognitive map. Tolman's arguments and similar arguments by other authors were instrumental in bringing about the new cognitive science, which unlike behaviorism allowed for internal constructs such as cognitive maps. As described in chapter 2, the cognitive map has virtually always been interpreted as a distinctly separate part or "module", independent of the sensory part on the one hand and the planning/control part on the other. Furthermore, the map is assumed to consist of a more or less isomorphic mapping from the real world to the internal structure, or to "mimic" the world in its internal layout. That is, different places in the world correspond to distinctly different places or elements in the map, and spatial relationships between different places in the world are more or less directly replicated in the map. See figure 5.9 for a typical example of this type of cognitive map model, from Trullier and Meyer (1998).

We can see that none of these properties are obviously apparent for the system investigated here. The sensory part, the controller, and the "cognitive map" are all encoded in the same set of weights. Figure 5.10 shows the set of weights of the network

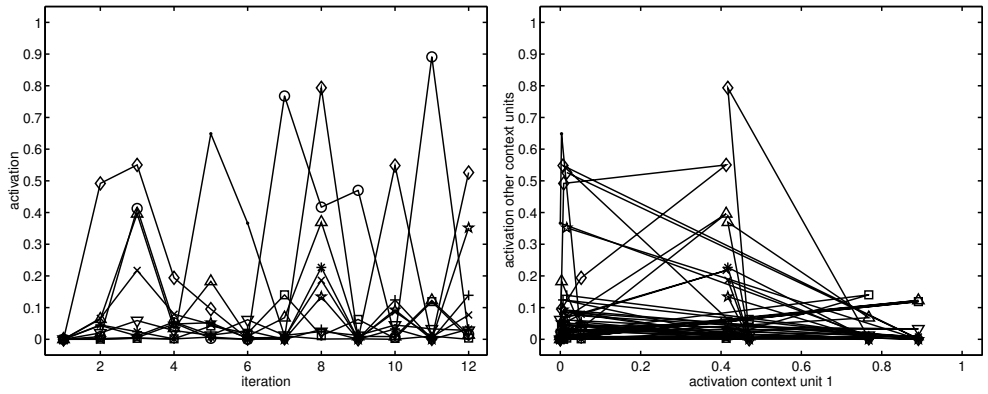

Figure 5.11: Fig. a (left). Context unit activations over time within an episode in the maze task. Fig. b (right). The same context unit activations over time, but now plotted against the activation of context unit 1, yielding a phase space trajectory plot.

whose outward behavior was discussed above and depicted in figure 5.4. No obvious cognitive map can be discerned.

Figure 5.11a shows the context unit activations, which correspond to the network's internal state, over time during an episode. Figure 5.11b shows the same information, but now as a phase space trajectory plot, i.e. as a projection of the state (or phase) space onto 2 dimensions. This can in some cases give additional insight into a recurrent network's functioning (Casey, 1996; Rodriguez, Wiles, & Elman, 1999; Wiles & Elman, 1995; Blair & Pollack, 1997), especially with regard to the internal state space's attractor structure.

In neither of these graphs, an obvious cognitive map can be discerned. For example, states in the environment that are close to each other do not necessarily lead to internal states that are close to each other in state space. In general, the network makes large jumps through internal state space with each new observation and action. Thus, it does not move closer to one particular attractor in state space with each time step, which would correspond to the network doing hysteretic computation, in the terminology of Casey (1996). This is apparent when we artifially clamp the input layer to a single input vector. In that case, the context unit activations do converge to single point attractors. During normal operation, with each new observation-action pair the network can arrive in the basin of attraction of another attractor, and the internal state takes a jump in the new direction: transient computation, in the terminology of Casey (1996). In any case, neither the attractors in the state space nor the points in the state space visited at different times have an immediate correspondence to states in the world, which would be the expectation given a traditional cognitive map interpretation of the network's functioning.

However, when we look at many different episodes started from different positions, there are statistically significant differences ($p < .01$) between the average vectors of

context unit activations in different environmental states, as measured by a Multivariate Analysis Of Variance (MANOVA) (van Dartel, 2001). This basically means there is a "correlation", which is significant but certainly not 1, between specific world states and context unit activations. This is to be expected, because one world state is typically preceded by similar observations and actions and therefore the internal states arrived at in such a world state are likely to have similarities. It is conceivable that a similar effect explains the existence of so-called "place cells", neurons in the rat's hippocampus which fire correlated with positions in the world (O'Keefe & Nadel, 1978). This finding has usually been interpreted as an argument for the existence of a straightforward cognitive map in the hippocampus, but we see here that it could indicate something weaker.

Finally, a (Mealy) Finite State Automaton (FSA) was extracted from the network, in the same way as in the previous chapter. The idea behind this is that even if network weights or activation vectors cannot immediately be interpreted in terms of a cognitive map, perhaps the abstraction afforded by an FSA description of the network's functioning might allow us to identify the cognitive map. As in the previous chapter, the resolution of the discretization process was increased until the size of the minimized FSA did not increase anymore and the network's behavior was replicated accurately.

Figure 5.12 shows the extracted FSA after Hopcroft minimization, which models the network's behavior accurately. It has 26 states. The network can apparently be modeled well using 26 states; in other words, we can say that the network *effectively* uses 26 states for its overall behavior. First of all, the network's internal state space is evidently more complex than in the previous chapter, where the maze navigation task was simpler and the networks induced in the order of 1 to 6 functional states. Furthermore, the extracted FSA obviously has a very complex pattern of edges connecting the states. That is, given that the system is in a certain internal state, the transition to a next internal state depends heavily and in an intricate manner on the particular next observation and action. Most importantly for the purposes of our current analysis, there is no straightforward correspondence between FSA states and world states. The most we can say is that at the end of each episode, when the agent is approaching the goal, the FSA tends to go through the same small number of FSA states in the same order. This makes sense because once the agent is close to the goal, the agent "knows" where it is and always takes the same actions for particular observations.

In summary, the recurrent neural network trained using reinforcement learning apparently behaves as if it has something like a cognitive map, without having an obviously apparent "explicit" or straightforward cognitive map implementation as would be expected from the traditional neuroscience and cognitive science perspective. It can still be the case that further analyses, e.g. principal component analysis on the hidden unit acivations or other, more sophisticated analyses, *would* reveal where the cognitive map resides and that there are straightforward correspondences between locations in the world and internal states. However, this would still be a very different type of cognitive map from the one hypothesized by traditional neuroscience and cognitive science, which clearly assumes a distinct module, separate from other processing, with straightforward isomorphic correspondences between locations in the world and locations or "units" in the cognitive map module.

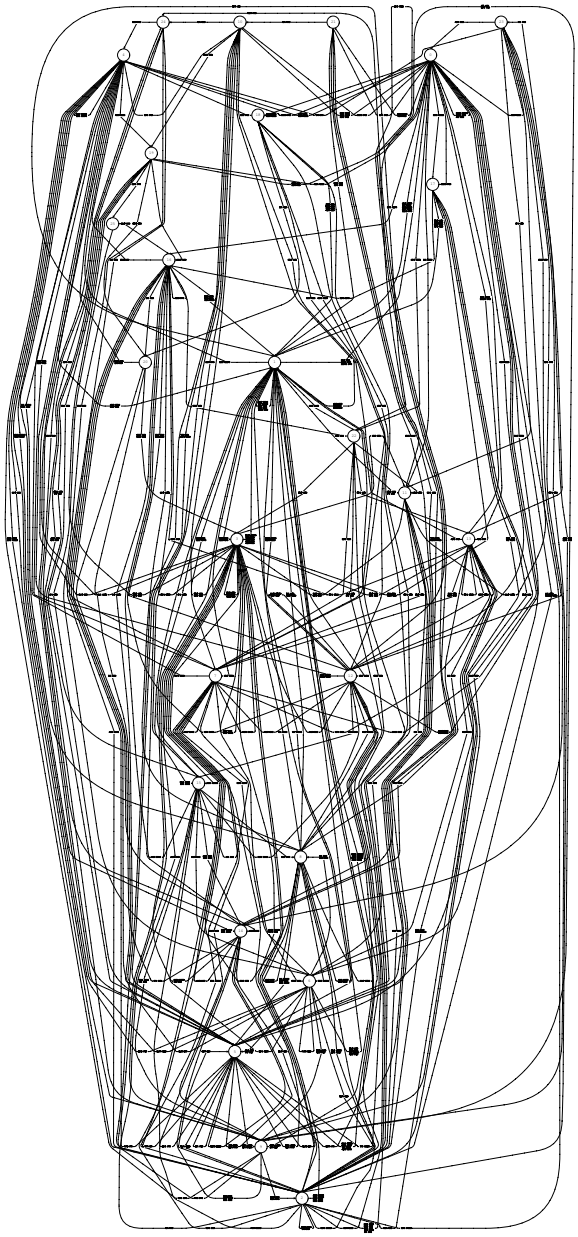

Figure 5.12: The Mealy FSA extracted from the trained Elman network, after Hopcroft minimization. The FSA has 26 states.

Admittedly, the "cognitive map" of the recurrent neural network investigated here is limited in its capabilities. For example, it cannot be adapted in a simple way to navigate to another goal position (Kröse, personal communication). However, the main point is that it is possible to have certain behavioral capabilities that at first sight suggest that the system must contain a certain subsystem, without necessarily implementing that subsystem straightforwardly. This provides another example of the principle discussed in chapter 2 which says that bottom-up engineering can lead to a different type of system than top-down engineering, and post hoc analysis can reveal that a bottom-up engineered system works in another way than an a priori analysis of the behavior would suggest. This is an important realization, because cognitive scientists are often eager to draw conclusions from behavioral data about what the brain's internal architecture must look like. This study shows, once again, that we have to be careful in doing that. In this particular case, cognitive scientists and neuroscientists have tended to look for straightforward cognitive maps in rats' and humans' brains (O'Keefe & Nadel, 1978; Wagatsuma & Yamaguchi, 1999; Trullier & Meyer, 1998). The simulation study reported here shows that learning, brain-like systems *may* accomplish navigation tasks in a very different way. It certainly does not *prove* that rats' and humans' brains *do not* contain straightforward cognitive maps. It only says that it is possible to achieve cognitive map-like behavior without having an explicit cognitive map in the sense expected by traditional neuroscience and cognitive science, it says that it may be impossible to find a straightforward cognitive map in the brain, and it says that we have to keep our eyes open for other types of solutions in the brain to the navigation problem than a straightforward cognitive map.

5.6 Discussion

This chapter investigated reinforcement learning in POMDPs using Elman recurrent neural networks. The main technical contribution was the application of Advantage learning to partially observable tasks, and the extension of Advantage learning to Advantage(λ) learning. Experimental data showed that, at least in the settings investigated in this chapter, Advantage(λ) learning clearly outperforms Q(λ)-learning, and Advantage(λ) learning clearly outperforms plain Advantage learning. This was shown both in a continuous task, partially observable pole balancing, and in a discrete task, partially observable maze navigation.

For the maze navigation task, a more extensive analysis of the agents' behavior and mechanisms was presented. It turns out that the behavior is such that according to classical criteria, the agent can be said to behave as if it has a cognitive map. However, the interesting thing is that this behavior is accomplished without a straightforward explicit cognitive map (in the traditional neuroscience and cognitive science sense) being obviously discernible in the internal mechanism. This suggests that we may have to rethink the classical conception of what a cognitive map must look like, and whether we can and should expect to find straightforward cognitive maps in animals' and humans' brains.

Chapter 6

Reinforcement learning with Long Short-Term Memory

Summary

This chapter presents reinforcement learning with a Long Short-Term Memory recurrent neural network: RL-LSTM. Model-free RL-LSTM using Advantage(λ) learning and directed exploration can solve non-Markovian tasks with complex and long-term dependencies between relevant events. This is demonstrated in a T-maze task, as well as in a difficult variation of the pole balancing task. Furthermore, two well-known POMDPs from the literature are investigated, in order to make a comparison to other algorithms and to show that LSTM has good performance on them as well.

6.1 Introduction

Most of the approaches to solving partially observable reinforcement learning tasks, surveyed in chapter 3 and including the approaches of chapters 4 and 5, have difficulties if there are complex and long-term dependencies between relevant past events and the currently best action. This is an important problem because there are no strong reasons to suppose that realistic partially observable reinforcement learning tasks will have only short-term dependencies, that is, can always be solved by simply remembering the last few observations and/or actions. An example of a long-term dependency task is a maze navigation task where the only way to distinguish between two T-junctions that look identical is to remember a distinguishing observation a long time before either T-junction.

In such a case there is, for example, no straightforward way to decompose the non-Markovian task into Markovian subtasks using Wiering and Schmidhuber's (1997) Hierarchical Q-learning approach (section 3.6.3.6): the agent simply must remember the relevant piece of information up until the T-junction. There are also obvious problems with fixed size history window approaches (section 3.6.3.3): if the relevant

piece of information to be remembered falls outside the history window, the agent cannot use it.

Ring's (1993a, 1993b) and McCallum's (1996) variable history window approaches (sections 3.6.3.9, 3.6.3.4, and 3.6.3.5) have, in principle, the capacity to represent long-term dependencies. However, these algorithms start out with zero history and increase the depth of the history window step by step, based on gathering statistics. This process makes learning long time lag dependencies virtually impossible when there are no short-term dependencies to build on, and even if it worked, it would be a very time consuming process. In addition, a history window approach must represent the entire history from the relevant piece of information onwards, even if intermediate events are irrelevant, leading to a very large number of trainable parameters (the curse of dimensionality) and yielding an unnecessarily large policy representation.

Model-free approaches based on memory bits (section 3.6.3.7, Peshkin et al., 1999; Lanzi, 2000; Cliff & Ross, 1994), FSAs (section 3.6.3.12, Chrisman, 1992; Meuleau et al., 1999), or recurrent neural networks (section 3.6.3.8), which were investigated in the previous two chapters, do not have to represent (possibly long) entire histories, but can in principle extract and represent just the relevant information for an arbitrary amount of time. The same is true for a number of approaches that learn a predictive model of the environment (section 3.6.2.4).

However, *learning* to extract and represent information from a long time ago has proven difficult, both for model-based and for model-free approaches. The difficulty lies in discovering the correlation between a piece of information and the moment at which this information becomes relevant at a later time, given the distracting observations and actions between them. This is also true for the recurrent neural networks studied in the previous chapters, and one of the reasons for their relatively long training times and occasional failure.

This difficulty can be viewed as an instance of the general difficulty of learning long-term dependencies in timeseries data (Hochreiter, 1991; Bengio, Simard, & Frasconi, 1994; Hochreiter, Bengio, Frasconi, & Schmidhuber, 2001). This chapter uses one particular solution to this problem that has worked well in a variety of *supervised* timeseries learning tasks: the Long Short-Term Memory (LSTM) recurrent neural network (Hochreiter & Schmidhuber, 1997; Gers, Schmidhuber, & Cummins, 2000; Eck & Schmidhuber, 2002).

In this chapter an LSTM recurrent neural network is used in conjunction with model-free RL, in the same spirit as the model-free recurrent neural network approaches studied in the previous chapters. Again the network learns to approximate the value function of the reinforcement learning algorithm called Advantage(λ) learning. Another technical contribution of this chapter is a novel directed exploration method.

The next section describes LSTM. Section 6.3 presents LSTM's combination with reinforcement learning in a system called RL-LSTM. Section 6.4 contains simulation results on partially observable RL tasks with long-term dependencies. Section 6.5, finally, presents the general conclusions.

6.2 LSTM

6.2.1 Memory cells

LSTM is a recently proposed recurrent neural network architecture, originally designed for supervised timeseries learning (Hochreiter & Schmidhuber, 1997; Gers et al., 2000; Eck & Schmidhuber, 2002). It is based on an analysis of the problems that conventional recurrent neural networks and their corresponding learning algorithms, e.g. Elman networks with standard one step backpropagation as used in the previous chapters, or Elman networks with backpropagation through time (BPTT) or real-time recurrent learning (RTRL), have when learning timeseries with long-term dependencies. These problems boil down to the problem that errors propagated back in time tend to either vanish or blow up (see Hochreiter & Schmidhuber, 1997).

LSTM's solution to this problem is to enforce *constant* error flow in a number of specialized units, called Constant Error Carrousels (CECs). This turns out to correspond to the CECs having linear activation functions which do not decay over time. In order to prevent the CECs from filling up with useless information from the timeseries, access to them is regulated using other specialized, multiplicative units, called input gates. Like the CECs, the input gates receive input from the timeseries and the other units in the network, and they *learn* to open and close access to the CECs at appropriate moments.

Access *from* the activations of the CECs to the output units (and possibly other units) of the network is regulated using multiplicative output gates. Similar to the input gates, the output gates learn when the time is right to send the information stored in the CECs to the output side of the network. A recent addition is forget gates (Gers et al., 2000), which learn to reset the activation of the CECs (in a possibly gradual way) when the information stored in the CECs is no longer useful. The combination of a CEC with its associated input, output, and forget gate is called a memory cell. See figure 6.1b for a schematic of a memory cell. It is also possible for multiple CECs to be combined with only one input, output, and forget gate, in a so-called memory block.

6.2.2 Activation updates

Figure 6.1a shows the general LSTM network architecture used in this (and the next) chapter. The network's activations are computed as follows. The net input $net_i(t)$ of any unit i at time t is calculated by

$$net_i(t) = \sum_m w_{im} y_m(t-1) \tag{6.1}$$

where w_{im} is the weight of the connection from unit m to unit i.[1] A standard hidden unit's activation y_h, output unit activation y_k, input gate activation y_{in}, output gate activation y_{out}, and forget gate activation y_φ is computed as

$$y_i(t) = f_i(net_i(t)) \tag{6.2}$$

[1] For reasons of ease of notation, the activations feeding into a unit are always viewed as activations from one timestep ago.

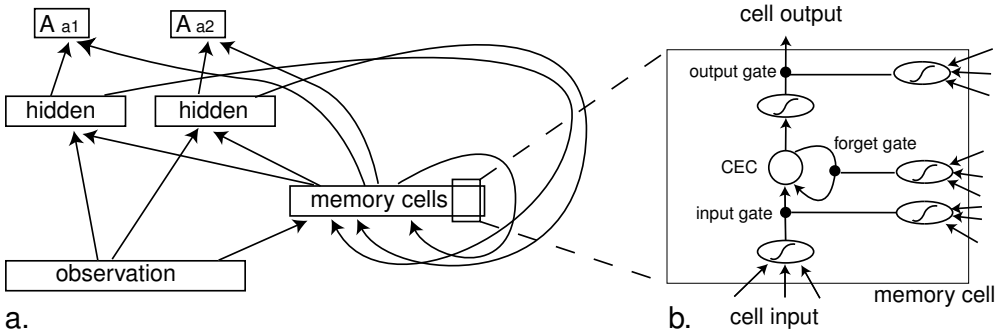

Figure 6.1: Fig. a (left). The general LSTM architecture used in this chapter. Arrows indicate unidirectional, fully connected weights. The network's output units (2, in this illustration) directly code for the Advantage values of individual actions. Each output unit has its own associated hidden layer feeding into it. Fig. b (right). One memory cell.

where f_i is the standard logistic sigmoid function, squashing the net input to the range $[0, 1]$. The standard hidden units receive input from the input layer and the memory cell outputs. The gates receive the same input, plus input from the hidden layer and all the gates. The CEC activation $z_{c_j^v}$, or the state of memory cell v in memory block[2] j, is computed as follows:

$$z_{c_j^v}(t) = y_{\varphi_j}(t) z_{c_j^v}(t-1) + y_{in_j}(t) g(net_{c_j^v m}(t)) \tag{6.3}$$

where g is a logistic sigmoid function scaled to the range $[-2, 2]$, and $z_{c_j^v}(0) = 0$. Note how the memory cell's input gate activation y_{in_j} determines, in a multiplicative way, to what extent the net input "enters" the memory cell. This net input comes from the input layer, standard hidden layer, the gates, and the memory cell outputs. The memory cell's output $y_{c_j^v}$ is calculated by

$$y_{c_j^v}(t) = y_{out_j}(t) h(z_{c_j^v}(t)) \tag{6.4}$$

where h is a logistic sigmoid function scaled to the range $[-1, 1]$. Similar to the input gate, the output gate activation y_{out_j} determines, in a multiplicative way, to what extent the memory cell's contents are made available to other units, among which are the output units. An output unit's activation y_k, finally, is computed using

$$y_k(t) = f_k(net_k(t)) = net_k(t). \tag{6.5}$$

In this chapter, as in the previous chapter, f_k is the identity function. An output unit receives input from a dedicated layer of standard hidden units (for reasons explained in section 6.3.1), and from all the memory cell outputs.

[2]In this thesis, there is always only one memory cell per memory block. Therefore, one of the indices v or j could be omitted. They are both left in in order to maintain generality, and to maintain compatibility with the standard LSTM formulation (see Hochreiter & Schmidhuber, 1997; Gers et al., 2000).

6.2.3 Learning

At some or all timesteps of the timeseries, the output units of an LSTM network may make prediction errors. Errors are propagated just one step back in time through all units other than the CECs, including the gates. However, errors are backpropagated through the CECs for an indefinite amount of time, using a variation of RTRL. In contrast to traditional recurrent neural network architectures, such as the Elman networks of the previous two chapters, the linear nature of CECs prevents computed gradients with respect to past events from decaying, while the gates prevent disturbing influences to and from the CECs. Thus, each weight is updated based on the estimated gradient of the error of the network's output units with respect to this weight. For example, the weight of a connection to an input gate may be increased because the estimated gradient of the error with respect to this weight says "if this weight is increased, the error will go down", and because of this, the input gate may learn to open for a particular input.

Unlike standard BPTT and RTRL, the resulting learning algorithm for LSTM (Hochreiter & Schmidhuber, 1997; Gers et al., 2000) is local in both space and time, and its update complexity per weight and timestep is $O(1)$. Weight updates can be done at every timestep, which fits in nicely with the philosophy of online reinforcement learning. The learning algorithm is adapted slightly for reinforcement learning, as explained in the next section.

6.3 RL-LSTM

6.3.1 Model-free RL-LSTM

As in the previous two chapters, the recurrent neural network is used as the function approximator for a model-free, value function-based reinforcement learning algorithm (see figure 6.1a). Again, the state of the environment is approximated by the current observation, which is the input to the network, together with the recurrent activations in the network, which represent the agent's history. In this case, the recurrent activations in the specialized memory cells (figure 6.1b) are supposed to learn to represent relevant information from long ago.

In contrast with the previous chapters, the network has multiple output units. Each output unit codes directly for the Advantage value of one of the possible actions (see figure 6.1a). At each timestep, the current temporal difference error (see equation 5.4) is the error for the output unit coding for the Advantage of the executed action at the current timestep. The other output units do not receive error signals.

Each output unit has its own dedicated standard hidden units layer. This is done because preliminary experiments as well as previous experience with combining Q-learning and neural networks (Lin, 1992; Abul et al., 2000) suggest that if there is only a single hidden layer, temporal difference error signals, which concern only one action and therefore only one output unit, lead, over time, to conflicting error signals to the single hidden layer, which makes learning difficult. This problem is overcome by giving each output unit its own associated hidden layer feeding into it and receiving its temporal difference error signal. There is only one layer of memory cells, however,

because the same short-term memory information should be available to all output units. Another approach to this problem of conflicting error signals to the hidden layer is, of course, to give an action representation as input to the network, as in the previous two chapters. Still another approach is to use separate networks for each action (Lin, 1992; Abul et al., 2000). The advantage of the current approach is that the Advantage values for different actions are computed using one sweep through one network, and only one internal state is used for all state-action pairs, hopefully facilitating generalization.

In sum, this architecture allows the output units to exploit information available in the current observation directly, using the connections via the hidden layer. At the same time, the output units have access to the short-term memory information encoded in the memory cells, using both a direct connection from the memory cells and a connection via the hidden units. The latter connection allows for information in the memory cells to be used by the output units in a more complex, nonlinear way.

6.3.2 Advantage(λ) learning using LSTM

As in the previous chapter, the model-free RL algorithm that is used to train the recurrent neural network is Advantage(λ) learning. It is again implemented using eligibility traces. Each weight update corresponds to

$$\Delta w_{im}(t) = \alpha E^{TD}(t) e_{im}(t) \tag{6.6}$$

for all parameters w_{im}, where eligibility trace $e_{im}(t)$ is computed as

$$\begin{aligned} e_{im}(t) &= \gamma\Lambda(t) e_{im}(t-1) + \frac{\partial A(s(t),a(t))}{\partial w_{im}} \\ &= \gamma\Lambda(t) e_{im}(t-1) + \frac{\partial y_K(t)}{\partial w_{im}} \end{aligned} \tag{6.7}$$

where K is the index of the output unit representing the Advantage value of the current action, and

$$\Lambda(t) = \begin{cases} \lambda & a(t) = \arg\max_a A(s(t),a) \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

LSTM's variation of real-time recurrent learning truncates $\frac{\partial y_k(t)}{\partial w_{im}}$ at all points in the network other than the CECs (see section 6.2.3). Since the same approximations to $\frac{\partial y_k(t)}{\partial w_{im}}$ are used in equation 6.7, this means that eligibility traces are truncated at the same points.

In addition to the inclusion of eligibility traces, a minor rearrangement of the variables and the gradient computations is required, compared to the original LSTM formulation. This is so because in contrast with the original LSTM formulation, the partial derivatives $\frac{\partial y_K(t)}{\partial w_{im}}$ must now be calculated separately and explicitly. None of these changes affect LSTM's update complexity.

For the network architecture used in this chapter, which is depicted in figure 6.1a and described in section 6.2.2, the partial derivatives $\frac{\partial y_K(t)}{\partial w_{im}}$ can be derived from the activation update equations by repeated application of the chain rule. This yields the

following equations. For the weights from memory cell outputs and from the dedicated layer of standard hidden units to the output unit corresponding to the current action ($i = K$),

$$\frac{\partial y_K(t)}{\partial w_{Km}} = f'_k(net_K(t))y_m(t-1). \tag{6.9}$$

For the weights from input units and memory cell outputs to standard hidden units h,

$$\frac{\partial y_K(t)}{\partial w_{hm}} = w_{Kh}f'_k(net_K(t))f'_h(net_h(t))y_m(t-1). \tag{6.10}$$

For the weights from input units, memory cell outputs, gates, and standard hidden units to output gate units out_j,

$$\begin{aligned}\frac{\partial y_K(t)}{\partial w_{out_j m}} =& \sum_{v=1}^{Z_j} h(z_{c_j^v}(t))\Big(w_{Kc_j^v}f'_k(net_K(t)) + \sum_h w_{Kh}f'_k(net_K(t))w_{hc_j^v}f'_h(net_h)(t)\Big) \\ & \cdot f'_{out_j}(net_{out_j}(t))y_m(t-1)\end{aligned} \tag{6.11}$$

where Z_j is the number of CECs (1, in this chapter) in memory block j. For the weights from input units, memory cell outputs, gates, and standard hidden units to CEC units c_j^v,

$$\begin{aligned}\frac{\partial y_K(t)}{\partial w_{c_j^v m}} =& \Big(w_{Kc_j^v}f'_k(net_K(t)) + \sum_h w_{Kh}f'_k(net_K(t))w_{hc_j^v}f'_h(net_h)(t)\Big) \\ & \cdot y_{out_j}(t)h'(z_{c_j^v}(t))\frac{\partial z_{c_j^v}(t)}{\partial w_{c_j^v m}}\end{aligned} \tag{6.12}$$

where $\frac{\partial z_{c_j^v}(t)}{\partial w_{c_j^v m}}$ is the information that needs to be stored in LSTM's version of real time recurrent learning used within the memory cells. It is updated as follows:

$$\frac{\partial z_{c_j^v}(t)}{\partial w_{c_j^v m}} = \frac{\partial z_{c_j^v}(t-1)}{\partial w_{c_j^v m}}y_{\varphi_j}(t) + g'(net_{c_j^v}(t))y_{in_j}(t)y_m(t-1). \tag{6.13}$$

For the weights from input units, memory cell outputs, gates, and standard hidden units to input gates in_j,

$$\begin{aligned}\frac{\partial y_K(t)}{\partial w_{in_j m}} =& \sum_{v=1}^{Z_j}\Big(w_{Kc_j^v}f'_k(net_K(t)) + \sum_h w_{Kh}f'_k(net_K(t))w_{hc_j^v}f'_h(net_h)(t)\Big) \\ & \cdot y_{out_j}(t)h'(z_{c_j^v}(t))\frac{\partial z_{c_j^v}(t)}{\partial w_{in_j m}}\end{aligned} \tag{6.14}$$

where $\frac{\partial z_{c_j^v}(t)}{\partial w_{in_j m}}$ is calculated by

$$\frac{\partial z_{c_j^v}(t)}{\partial w_{in_j m}} = \frac{\partial z_{c_j^v}(t-1)}{\partial w_{in_j m}}y_{\varphi_j}(t) + g(net_{c_j^v}(t))f'_{in_j}(net_{in_j}(t))y_m(t-1). \tag{6.15}$$

Finally, for the weights from input units, memory cell outputs, gates, and standard hidden units to forget gates φ_j,

$$\begin{aligned}\frac{\partial y_K(t)}{\partial w_{\varphi_j m}} = &\sum_{v=1}^{Z_j}\Big(w_{Kc_j^v} f'_k(net_K(t)) + \sum_h w_{Kh} f'_k(net_K(t)) w_{hc_j^v} f'_h(net_h)(t)\Big) \\ &\cdot y_{out_j}(t) h'(z_{c_j^v}(t)) \frac{\partial z_{c_j^v}(t)}{\partial w_{\varphi_j m}}\end{aligned} \tag{6.16}$$

where $\frac{\partial z_{c_j^v}(t)}{\partial w_{\varphi_j m}}$ is calculated by

$$\frac{\partial z_{c_j^v}(t)}{\partial w_{\varphi_j m}} = \frac{\partial z_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} y_{\varphi_j}(t) + z_{c_j^v}(t-1) f'_{\varphi_j}(net_{\varphi_j}(t)) y_m(t-1). \tag{6.17}$$

Because the initial state of the network does not depend on the weights, $\frac{\partial z_{c_j^v}(0)}{\partial w_{im}} = 0$ for all units i that need to store this information, i.e. the CECs, the input gates, and the forget gates.

6.3.3 Exploration

Non-Markovian RL requires extra attention to the issue of exploration (Chrisman, 1992; McCallum, 1996; Wiering & Schmidhuber, 1997). This issue was ignored in the previous chapter, but we have to address it now, because the investigated tasks become more complex. Undirected exploration attempts to try out actions in the same way in each environmental state. However, in non-Markovian tasks, the agent initially does not know which environmental state it is in. Part of the exploration should be aimed at discovering the environmental state structure. Furthermore, in many cases, the non-Markovian environment will provide unambiguous sensory information indicating the state in some parts, while providing ambiguous sensory information (hidden state) in other parts. In general, we want more exploration in the ambiguous parts. Finally, reconstructing the environmental state signal based on the experienced history of observations and actions, which is the general approach to dealing with partial observability in this thesis, depends on there being certain temporal regularities in that history. Finding those temporal regularities is facilitated a great deal by reducing the amount of exploration where it is possible, i.e. taking more or less the same set of actions when possible. This will lead to less diverse sequences of observation-action pairs, in which that information in the sequence which can disambiguate the state is more easily visible to the system.

This chapter employs a new directed exploration technique based on these ideas. A separate multilayer feedforward neural network, with the same input as the LSTM network (representing the current observation) and one output unit y_v, is trained concurrently with the LSTM network. It is trained, using standard backpropagation, to predict the absolute value of the current temporal difference error $E^{TD}(t)$, plus its own discounted prediction at the next timestep:

$$y_v^D(t) = |E^{TD}(t)| + \beta y_v(t+1) \tag{6.18}$$

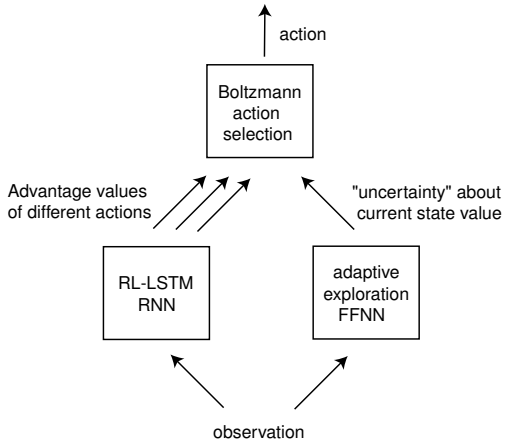

Figure 6.2: Schematic representation of the overall action selection mechanism. The current observation goes into both the RL-LSTM network and the adaptive exploration feedforward neural network. The latter network outputs a measure of the system's "uncertainty" about the value of the current state. This measure is linearly scaled and used as the temperature of a Boltzmann action selection rule, which operates on the Advantage values estimated by the RL-LSTM network.

where $y_v^D(t)$ is the desired value for output $y_v(t)$, and β is a discount parameter in the range $[0, 1]$. This amounts to attempting to identify which observations are "problematic", in the sense that they are associated with large errors in the current value estimation (the first term), or precede situations with large such errors (the second term). Thus, at each timestep the feedforward neural network produces an output $y_v(t)$ for the current observation, which represents something like the "uncertainty" about the current state, in terms of the expected return.

Next, $y_v(t)$ is linearly scaled and used as the temperature of the Boltzmann action selection rule:

$$p(s,a) = \frac{e^{A(s,a)/Cy_v(t)}}{\sum_{m=1}^{M} e^{A(s,a_m)/Cy_v(t)}} \tag{6.19}$$

where C is a constant that scales y_v. Recall that the Boltzmann action selection rule was used in the previous two chapters as well, but there it was used with fixed temperature. Finally, the resulting probabilities are rescaled such that there is a maximum action selection probability of .95, in order to always have at least 5% exploratory actions. Figure 6.2 shows, schematically, how the overall action selection mechanism works.

The end result of this exploration scheme is much exploration when, in the current state, differences between estimated Advantage values are small (the standard, desired effect of Boltzmann exploration), or when there is much uncertainty about current Advantage values or Advantage values in the near future (the effect of a large value of

$y_v(t)$ computed by the feedforward neural network, leading to a high temperature in the Boltzmann rule).

This exploration scheme has obvious similarities with the statistically more rigorous technique of Interval Estimation (Kaelbling, 1990), as well as with certain model-based approaches where exploration is greater when there is more uncertainty in the predictions of a model (Schmidhuber, 1991a; Thrun & Möller, 1992). See section 3.5.2.8 for a more extensive description of those techniques. An important difference with Kaelbling's Interval Estimation technique is that that technique requires the storage of experienced returns obtained from each state, so as to be able to compute confidence intervals of expected returns. The method described here uses only absolute temporal difference errors available at the current moment, and uses a function approximator to learn to estimate the uncertainty about values on that basis. An important difference with the methods of (Schmidhuber, 1991a; Thrun & Möller, 1992) is that those are based on increasing exploration when the agent is uncertain about future observations, independent of whether or not that is important for the policy. The method described here, in contrast, is utility-based: it explores adaptively based on its uncertainty about returns obtainable from the current state.

In summary, the good thing about this directed exploration technique is that it will reduce the amount of exploration when the uncertainty about the current state is small, leading to relatively deterministic action selection and therefore fairly stable observation-action sequences in which temporal regularities can more easily be recognized, while still maintaining sufficient exploration where necessary. In the experiments described next, this exploration scheme (using 6 hidden units in the network) led to satisfactory results, whereas preliminary experiments showed that straightforward Boltzmann exploration, of the type used in the previous two chapters and in many studies on reinforcement learning, led to very poor performance.

6.4 Experiments

6.4.1 Long-term dependency T-maze.

6.4.1.1 The task

The first test problem is a non-Markovian grid-based T-maze (see figure 6.3). It was designed to test RL-LSTM's capability to bridge long time lags, without confounding the results by making the control task difficult in other ways. Variations of this task were studied before in a supervised learning context (Ulbricht, 1996; Rylatt & Czarnecki, 2000; Linåker & Jacobsson, 2001).

The agent has four possible actions: move North, East, South, or West. It does not have an orientation. The agent must learn to move from the starting position at the beginning of the corridor to the T-junction. There it must move either North or South to a changing goal position, which it cannot see. However, the location of the goal depends on a "road sign" (X), which the agent has seen at the starting position. If the agent reaches the goal, it receives a reward of 4. If it goes the wrong way at the T-junction, it receives a reward of $-.1$. In both cases, the episode ends and a new episode starts, with the new goal position and corresponding road sign set randomly

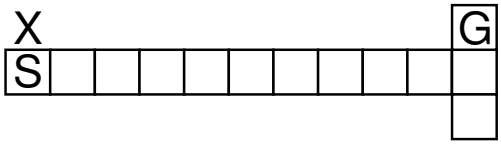

Figure 6.3: Long-term dependency T-maze with length of corridor $N = 10$. At the starting position S the agent's observation (X) indicates where the goal position G is in this episode.

either North or South. During the episode, the agent receives a reward of $-.1$ when it stands still.

At the starting position, the observation is either `011` or `110`, in the corridor the observation is `101`, at the T-junction the observation is `010`. The difficulty of this task depends strongly on the length of the corridor, N. In the simulation experiment, N was systematically varied from 5 to 70. In each condition, 10 runs were performed.

If the agent takes only optimal actions to the T-junction, it must remember the observation from the starting position for N timesteps to determine the optimal action at the T-junction. Note that the agent is not aided by experiences in which there are shorter time lag dependencies. In fact, the opposite is true. Initially, it takes many more actions until even the T-junction is reached, and the experienced history is very variable from episode to episode. The agent must first learn to reliably move to the T-junction. Because of the long delay and uncertainty of reward, this is already non-trivial, and it is one of the important bottlenecks in achieving good performance. Once this is accomplished, the agent will begin to experience more or less consistent and shortest possible histories of observations and actions, from which it can learn to extract the relevant piece of information. The directed exploration mechanism is crucial in this regard. It learns to set exploration low in the corridor and high at the T-junction. This results in the desired constancy in behavior where it is possible (the corridor), while still maintaining sufficient exploration where it is necessary (the T-junction).

6.4.1.2 Experimental comparisons

The LSTM network had 3 input units, 12 standard hidden units, 3 memory cells, and 4 output units. Learning rate $\alpha = .0002$. The following parameter values were used in all conditions: $\gamma = .98$, $\lambda = .8$, $\kappa = .1$. An empirical comparison was made with several other model-free POMDP solution methods. As noted before, the long-term dependency nature of a task like this virtually rules out history window approaches. Instead, three alternative systems were used that, like LSTM, are capable in principle of representing information for arbitrary long time lags.

In order to determine the specific contribution of LSTM to performance, in the alternative systems all elements of the overall system except LSTM remained the same, such as the Advantage(λ) learning algorithm and the directed exploration mechanism.

Preliminary experiments determined reasonable parameter settings for each of the alternatives, such as the magnitude of learning rate α and the number of hidden units.

As a first alternative to LSTM, the LSTM network was replaced by a reinforcement learning Elman network, trained using standard backpropagation, as in chapters 4 and 5. In contrast with those chapters, but like the LSTM network of this chapter, the Elman network did not have an action vector as input, but rather 4 output units coding for each of the Advantage values, and 4 corresponding dedicated hidden layers. The total number of hidden units was 16, there were 16 context units, and $\alpha = .01$.

The second alternative uses the same Elman network, but now trained using backpropagation through time (BPTT, Lin & Mitchell, 1993). Note that the unfolding of the RNN necessary for BPTT means that this is no longer truly online RL. In this system, $\alpha = .001$.

The third alternative was a table-based system extended with memory bits, which are part of the observation and that the controller can switch on and off (Littman, 1994; Peshkin et al., 1999, see section 3.6.3.7 for a more elaborate description). Because the task requires the agent to remember just one bit of information, this system had one memory bit, and $\alpha = .01$.

A run was considered a *success* if the agent learned to take the correct action at the T-junction in over 80% of cases, using its stochastic action selection mechanism. In practice, this corresponds to 100% correct action choices at the T-junction using greedy action selection, as well as optimal or near-optimal action selection leading to the T-junction.

6.4.1.3 Results

Figure 6.4 shows the number of successful runs (out of 10) as a function of the length of the corridor N, for each of the four methods. It also shows the average number of timesteps needed to reach success. It is apparent that RL-LSTM is able to deal with significantly longer timelags than the three alternatives. RL-LSTM has perfect performance up to $N = 50$, after which performance gradually declines. In those cases where the alternatives also reach success, RL-LSTM also learns faster.

The Elman network trained using standard backpropagation has the worst performance. The reason is probably that in Elman networks, the context unit activations, which are simply a copy of the hidden unit activations of the previous timestep, are easily overwritten by new information coming in through observations. In other words, even though the context units can *in principle* represent information from long ago, information from long ago quickly decays, such that standard backpropagation cannot assign credit to differential context unit activations that may help to solve the task. The Elman network trained using BPTT does much better, because credit is also assigned to context unit activations from previous timesteps, when the relevant information has not decayed as much as at the current timestep.

In contrast with the Elman networks, information stored in the internal state of the memory bits system does not decay automatically as new observations come in. Explicit internal actions are needed to change the memory bits. Still, the memory bits system does not perform very well. The reason why the memory bits system performs so much worse than both Elman-BPTT and LSTM may be that, in contrast with

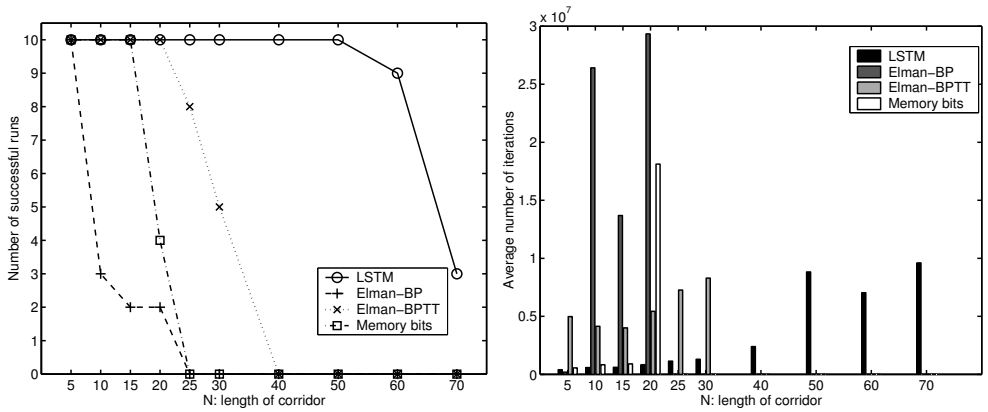

Figure 6.4: Fig. a (left). Number of successful runs (out of 10) as a function of N, length of the corridor, for each of the tested reinforcement learning systems in the noise-free T-maze task. Fig. b (right). Average number of iterations until success as a function of N.

these two, it does not explicitly compute the gradient of performance with respect to past events. Gradient computation is a powerful credit assignment mechanism. The Elman-BPTT system does compute such a gradient, but in contrast to LSTM, the gradient information tends to vanish quickly with longer timelags (as explained in section 6.2).

In this task, RL-LSTM has excellent performance up to a timelag of around 50 timesteps. Supervised LSTM is known to be capable of bridging minimal time lags of over 1000 timesteps in some cases. There are several possible reasons why RL-LSTM does not reach that level of performance, all related to the inherent differences between supervised learning and reinforcement learning.

One reason is the fact that, unlike supervised learning, the agent largely determines its history of inputs and outputs itself, as described above. This results in a less constant timeseries than in supervised learning (certainly in the beginning of learning), in which it is harder to find the regularities. A second reason is the fact that in RL based on temporal difference learning, target outputs change over time, because the system itself estimates values which are used in the target output, and these estimates change during learning. This may lead to radically changing backpropagated errors, for example when a value estimate that first was high later becomes low, because long-term negative effects become apparent. A third reason is that value functions are in general fairly complex, compared with many tasks tried in supervised LSTM studies (but see Gers, Eck, & Schmidhuber, 2001; Eck & Schmidhuber, 2002). A fourth reason is that in most supervised learning tasks where LSTM learned to bridge very long time lags, only the outputs at very few timesteps mattered, and error was only injected at those timesteps. In RL-LSTM, the system must estimate the value of each action at each timestep, and it receives a temporal difference error at each timestep.

To illustrate some of these differences, when the task is modified to a version where the policy leading up to the T-junction is fixed and only the last action needs to be learned using RL, RL-LSTM can easily learn time lag dependencies over 100 timesteps.

6.4.2 T-maze with noise.

6.4.2.1 Changes to the T-maze task and the architectures

It is one thing to learn long-term dependencies in a noise-free task, it is quite another thing to do so in the presence of severe noise. To investigate this, a very noisy variation of the T-maze task described above was designed. Now the observation in the corridor is `a0b`, where `a` and `b` are independent, uniformly distributed random values in the range $[0, 1]$, generated online. All other aspects of the task remain the same as above.

The LSTM, Elman-BP, and Elman-BPTT system were left unchanged. To allow for a fair comparison, the table-based memory bit system's observation was computed using Michie and Chambers's (1968) BOXES state aggregation mechanism (see Sutton & Barto, 1998 and section 3.5.2.9), partitioning each input dimension into three equal regions.

6.4.2.2 Results

Figure 6.5 shows the results. The memory bit system suffers most from the noise, relatively speaking. This is not very surprising because a table-based system, even if it is augmented with BOXES state aggregation, does not allow for very sophisticated generalization. The RNN approaches are less affected by the severe noise in the observations. Most importantly, RL-LSTM again significantly outperforms the others, both in terms of the maximum timelag it can deal with, and in terms of the number of timesteps needed to learn the task.

6.4.2.3 An extensive analysis

Let us take a closer look at how RL-LSTM solves this T-maze task. One type of analysis could be based on FSA extraction, as was done in the previous two chapters. However, this is both less straightforward and less necessary with LSTM networks than with the Elman networks of the previous chapters. It was more straightforward with the Elman networks because there the internal state was realized using a number of equivalent units with activations in the range $[0, 1]$. The LSTM network's internal state, in contrast, is realized using different types of units and activations that can be strongly interdependent, and the CECs are not constrained, a priori, to a fixed range. Moreover, as we shall see in a following section, FSA extraction may not even be possible at all (or at least be very unnatural), because the LSTM network may induce a system which is best understood as an automaton of a higher computational class than FSAs.

However, FSA extraction is also less necessary to make sense of the workings of the network in the case of LSTM networks than in the case of Elman networks, exactly *because of* the distinct functional roles played by different types of units, which facilitate a different type of analysis. For example, in Elman networks, without FSA

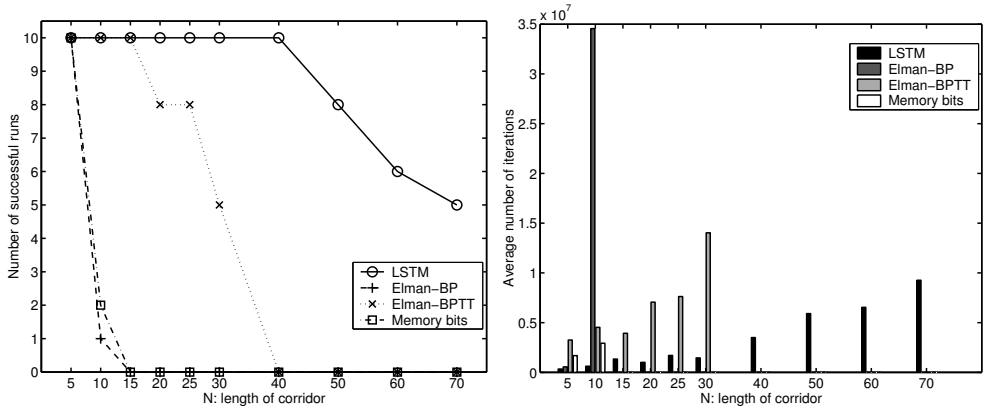

Figure 6.5: Fig. a (left). Number of successful runs (out of 10) as a function of N, length of the corridor, for each of the tested reinforcement learning systems in the noisy T-maze task. Fig. b (right). Average number of iterations until success as a function of N.

extraction it is difficult to determine whether changing context unit activations reflect functionally different internal states or just changing observations which do not change the Mealy internal state. In LSTM networks, this is less of a problem because input gates typically learn to close access to observations that are irrelevant with respect to internal states.

Figures 6.6 through 6.10 show the behavior over time, within an episode, of an RL-LSTM network that was trained on the noisy T-maze task with corridor length $N = 50$. Figure 6.6a shows the Advantage values of the different actions (N, E, S, W) estimated by the agent over time, together with the direct reward signal the agent obtains at each iteration. This concerns an episode where X is to the north of the starting position, such that action Go North (N) is the correct action at the T-junction. The estimated Advantage values are not exactly equal to the optimal Advantage values. However, they have the correct *relative* values, such that optimal behavior can be derived from them—and that is what is important. Within the corridor, action Go East (E) has the highest Advantage value. At the T-junction, action Go North (N) has the highest Advantage value. This is the correct action, so the agent obtains a reward of 4. Figure 6.6b shows the same information, but now for an episode where X is to the south of the starting position. Again, within the corridor action E has the highest Advantage value. However, at the T-junction actions S now has the highest Advantage value, which is the correct action in this case.

How does the LSTM network store the information necessary at the T-junction? Figure 6.7 shows the activations of the CECs over time for the two different conditions. CEC 2 and CEC 3 follow different trajectories over time and end up with different activation values at the T-junction. This provides the memory of the location of the X observed at the T-junction. CEC 1 does not behave very interestingly or differently for the two conditions.

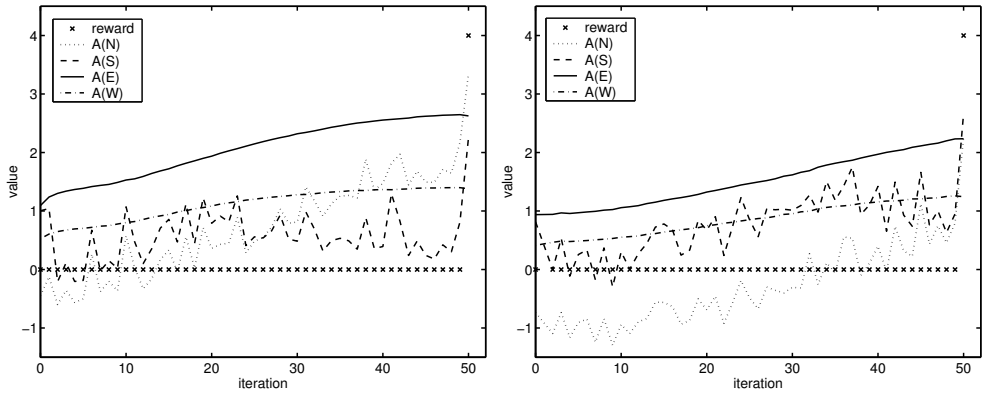

Figure 6.6: Advantage values over time within one episode of the noisy long-term dependency T-maze task. Fig. a (left). An episode where X is to the north of the starting position. Fig. b (right). An episode where X is to the south of the starting position.

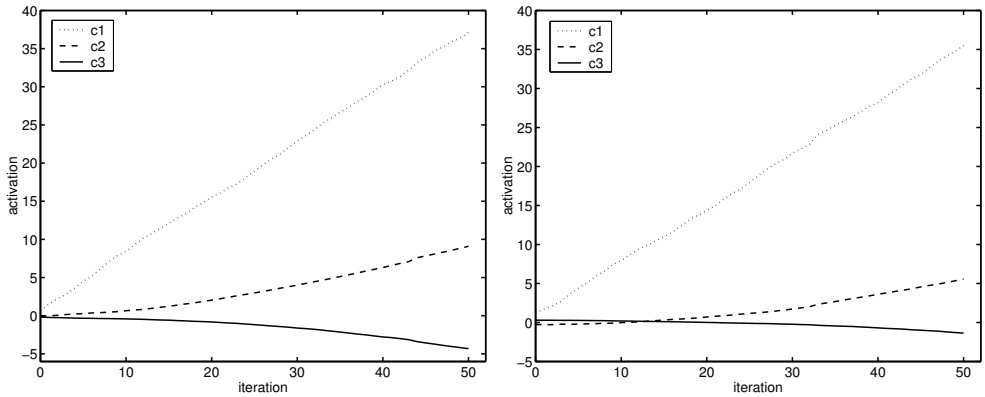

Figure 6.7: CEC activations over time within one episode of the noisy long-term dependency T-maze task. Fig. a (left). An episode where X is to the north of the starting position. Fig. b (right). An episode where X is to the south of the starting position.

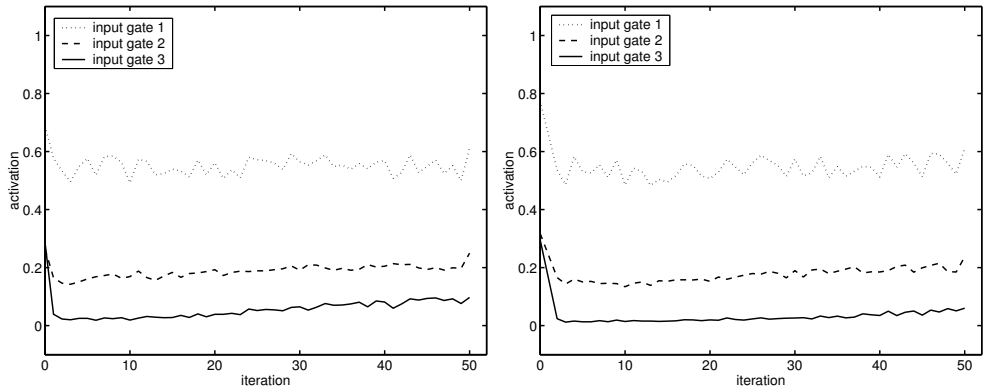

Figure 6.8: Input gate activations over time within one episode of the noisy long-term dependency T-maze task. Fig. a (left). An episode where X is to the north of the starting position. Fig. b (right). An episode where X is to the south of the starting position.

Figure 6.8 shows the input gate activations for the two conditions. It is apparent that in both conditions, after the starting position where the information from the observation is stored in CEC 2 and 3, the associated input gates 2 and 3 get lower activations, such that CEC 2 and 3 are not disturbed too much by subsequent observations. CEC 1's corresponding input gate does not close at all, such that CEC 3's activation keeps growing with every new, irrelevant observation. Note that the input gate activations do not behave differently in the two conditions.

Figure 6.9 shows the output gate activations for the two conditions. Again, the activations do not behave differently in the two conditions. In both cases, output gate 1 associated with CEC 1 is closed throughout the episode. Apparently LSTM has learned that CEC 1 does not contain useful information. Output gates 2 and 3 are open throughout the episode.

Figure 6.10, finally[3], shows the outputs of the memory cells for the two conditions. Because output gate 1 is closed the whole time, the output of memory cell 1 is 0 throughout the episode. The outputs of memory cells 2 and 3 basically reflect CEC 2 and 3's activations. The output of memory cell 3 is apparently used to encode the position of the X seen at the starting location. Furthermore, the outputs of both memory cells 2 and 3 seem to be used as a kind of timers. In this way, they represent how close the agent thinks it is to the goal. After all, state-action pairs close to the goal must have higher values.

[3]Forget gate activations are not plotted because they were always 1 ("open") throughout the episode, in both conditions.

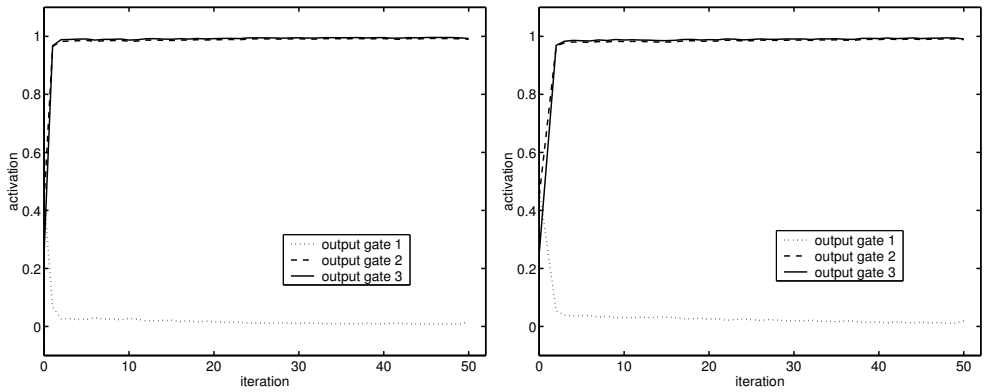

Figure 6.9: Output gate activations over time within one episode of the noisy long-term dependency T-maze task. Fig. a (left). An episode where X is to the north of the starting position. Fig. b (right). An episode where X is to the south of the starting position.

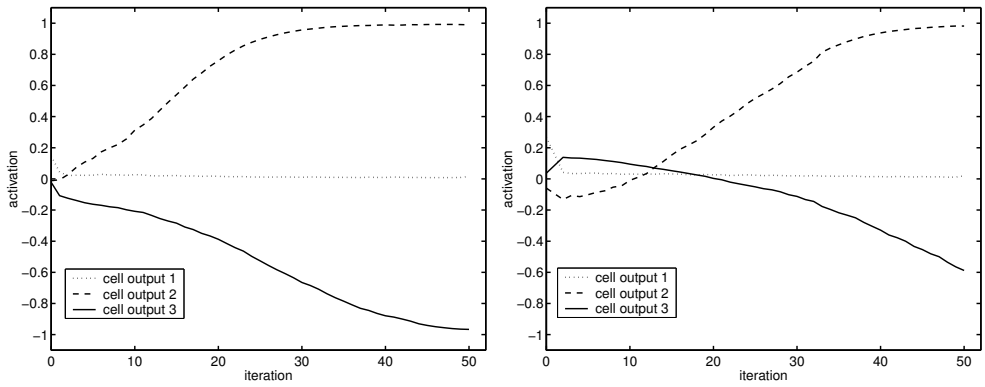

Figure 6.10: Memory cell outputs over time within one episode of the noisy long-term dependency T-maze task. Fig. a (left). An episode where X is to the north of the starting position. Fig. b (right). An episode where X is to the south of the starting position.

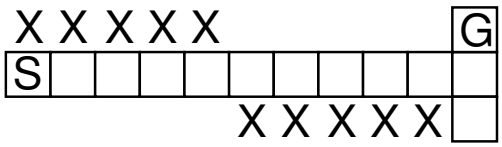

Figure 6.11: Non-regular T-maze. In this particular example $n = 5$, the length of the corridor is 10, and the sequence is grammatical.

6.4.3 Non-regular reinforcement learning

6.4.3.1 Task and architecture

In the previous test problems, the agent was faced with long-term dependency problems, but they were problems with a dependency on only one observation in the past, and they required only one bit of short-term memory. Let us consider now a task in which actions depend on complex combinations of past observations, a task also which requires a larger short-term memory.

The task, schematically shown in figure 6.11, is again a variation of the T-maze tasks described above. However, now the correct action at the T-junction depends not on a single observation made at the starting position, but on the entire sequence of observations made in the corridor. Specifically, in the first part of the corridor the agent sees Xs to the North (observation `100`), in the second part it sees Xs to the South (observation `001`). The correct action is to go North when the number of Xs in the first part is equal to the number of Xs in the second part, and go south if this is not the case. The length of the corridor varies between episodes, from 4 to 40. Thus, the agent must learn a general solution, basically counting Xs observed to the North and to the South.

In effect, for correct performance the agent must induce the non-regular language $a^n b^n$. "Grammatical" sequences of observations in the corridor must be followed by going North at the T-junction, "ungrammatical" sequences of observations must be followed by going South. In this task, ungrammatical sequences still have first Xs to the North and then Xs to the South, but the number of Xs to the North and the South is different.

An interesting property of non-regular languages is that they cannot be generated or recognized by finite state automata. That is, they require an automaton of a higher computational class in the Chomsky hierarchy, specifically an automaton with a push-down stack or a counter (Hopcroft & Ullman, 1979). This makes it a challenging learning task. Note also that now it is particularly important to take only optimal actions in the corridor leading up to the T-junction: wrong actions completely ruin the counting of Xs.

Starting with this test problem, the LSTM network was extended with so-called *peephole connections* (Gers & Schmidhuber, 2000). Peephole connections are connections from the activations of the CECs to the input of the memory cells and to the gates. Before, the memory cell and its gates only had access to the actual outputs of

the memory cells $y_{c_j^v}(t)$, i.e. after the CEC activations had passed through the output gates. However, the output gates are often closed so as not to influence the LSTM network's output at inappropriate times. It is possible that the information stored in the CECs could at those moments be useful for internal processing purposes. The contents of the CECs might reach certain values which should in fact trigger the output gates to open. This would, for instance, correspond to the CECs saying: "we have observed a particular history of observations; this means that now the outward behavior of the LSTM network should change; therefore, open the output gates!". Peephole connections allow processing of this kind, because the memory cell and its gates now receive direct information about the CEC activations. They increase the computation only by a small amount, due to the relatively few additional connections.

In contrast with Gers and Schmidhuber (2000), the peephole inputs are not $z_{c_j^v}(t)$, but $h(z_{c_j^v}(t))$ (see equation 6.4). The reason is that preliminary experiments showed that since $z_{c_j^v}(t)$ can grow very large, they can start to "dominate" the inputs to memory cells and gates and lead to explosions of weights. $h(z_{c_j^v}(t))$, on the other hand, is squashed to the range $[-1, 1]$; this loses some information, but is much safer.

Except for the addition of peephole connections, the same system was used as before, with the same number of hidden units, memory cells, learning parameters, and directed exploration mechanism. A comparison was made with the two best alternatives from the long-term dependency task, i.e. Elman-BPTT and the memory bits system. The Elman-BPTT system was left unchanged. The task requires many more bits of short-term memory than before (an infinite number if the n in $a^n b^n$ could be arbitrarily large); the memory bits system had 5 bits, more than enough to solve the task for the sequence lengths trained on.

6.4.3.2 Results

10 runs were done for each of the methods. A run was considered a *success* if the agent learned to take the correct action at the T-junction in over 90% of cases, using its stochastic action selection mechanism. Memory bits and Elman-BPTT never got significantly better than chance, i.e. around 50%. LSTM always reached the success criterion, in 3,498,230 iterations on average. Figure 6.12 shows the running average of the probability of correct action selection at the T-junction during a run, for typical runs of each of the methods. The graph shows typical runs, because averaging over runs is not very insightful: different successful runs discover and quickly converge to the correct policy at different moments.

After reaching the success criterion, training was stopped and the RL-LSTM agents were tested on their generalization ability. The agents had been trained on sequences with maximum $n = 20$ (corridor length 40), but now they were tested on sequences with maximum $n = 50$ (corridor length 100). Figure 6.13 shows the results for one typical agent. The proportion correct action selection at the T-junction, using greedy action selection, is plotted as a function of n. Results for grammatical and ungrammatical sequences are plotted separately. The results show that the agent has perfect performance on grammatical sequences of much greater length than it was trained on. The agent makes many more mistakes for ungrammatical sequences, treating some of

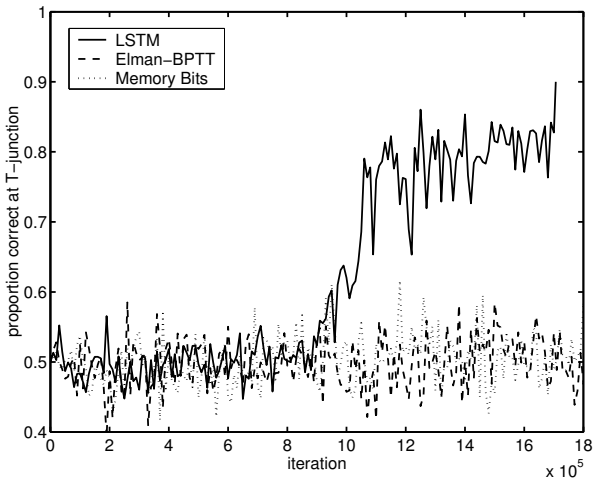

Figure 6.12: The probability of correct action selection at the T-junction as a function of the number of learning iterations, for typical runs of each of the three methods. Only LSTM gets significantly higher than chance and reaches the success criterion.

the ungrammatical sequences as grammatical. Closer inspection of these mistakes reveals that mistakes occur mostly in sequences that are "almost grammatical", i.e. have nearly the same number of Xs to the North as to the South. Nevertheless, for ungrammatical sequences the proportion correct action selection is still far better than chance, even with sequences much longer than the ones the agent was trained on.

Figure 6.14 depicts the CEC activations over time during an episode, for both a grammatical episode (left) and an ungrammatical episode (right), illustrating how LSTM solves the task. The CECs apparently implement counters which count up during the first part of the corridor when the agent sees Xs to the North, and count down when the agent sees Xs to the South. At the T-junction, some of the CEC activations must have particular values, typically close to 0, for the agent to choose action North (grammatical) at that point. This region of values associated with action North is typically a bit too large, such that the agent sometimes takes action North for ungrammatical sequences that end up in that region (almost grammatical sequences). The counters work the same for sequences of greater lengths than trained on, which is why the system generalizes as well as it does. In fact, the sequences can be much larger still than what is depicted in figure 6.13.

Figures 6.15, 6.16, and 6.17 show activations over time during an episode of the input gates, the output gates, and the resulting memory cell outputs. A striking difference between the grammatical and the ungrammatical sequence can be observed in the output gate activations. Output gate 1 seems to open at the moment when the sequence of observations in the corridor *so far* is grammatical. It is presumably triggered by watching the CEC activations using the peephole connections. If the sequence goes on and becomes ungrammatical, output gate 1 opens even further. In

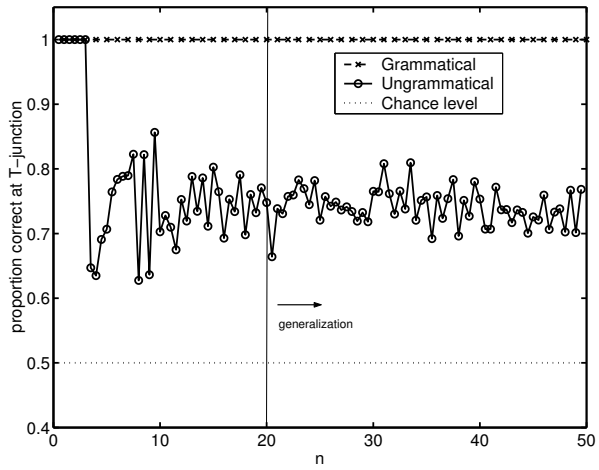

Figure 6.13: The proportion of correct action selection at the T-junction as a function of n, plotted separately for grammatical and ungrammatical sequences. RL-LSTM generalizes well to sequences of much greater lengths than the maximum length it was trained on (indicated by the vertical line).

either case, the opening of output gate 1 makes CEC activation 1, with very different values for the grammatical and ungrammatical sequence, available as information to be used by the output side of the network. This information is used at the T-junction. In the grammatical sequence, at the T-junction the resulting memory cell output 1 activation (figure 6.17) is around .35. In the ungrammatical sequence, the final memory cell output 1 activation has a value of around -1. These differing values allow the network to make the correct crucial decision at the T-junction in each of the two cases.

There are good reasons for saying that the LSTM network has really induced an (admittedly imperfect) automaton of a higher computational class than FSAs, rather than just an FSA that works well for the training exemplars. First, the network generalizes well to sequences of greater length, and it generalizes in the way expected of a counter- or stack-based automaton. There is no reason why an arbitrary implementation of an FSA would generalize in this way. Second, the network has a very apparent implementation of counters in the memory cells.

In grammar induction studies using recurrent neural networks, other authors have also induced such automata of a higher computational classes than FSAs (Blair & Pollack, 1997; Rodriguez et al., 1999; Gers et al., 2001). That work was done using supervised learning, and usually the network was trained to predict the next input, the next symbol in the grammar. Gers et al. (2001) use an LSTM network for that task, and their system induces similar counters as in this work. They also find that LSTM networks clearly outperform other recurrent neural network architectures.

The interesting result here is that RL-LSTM is apparently able to induce this relatively complex automaton based on the very sparse training information provided

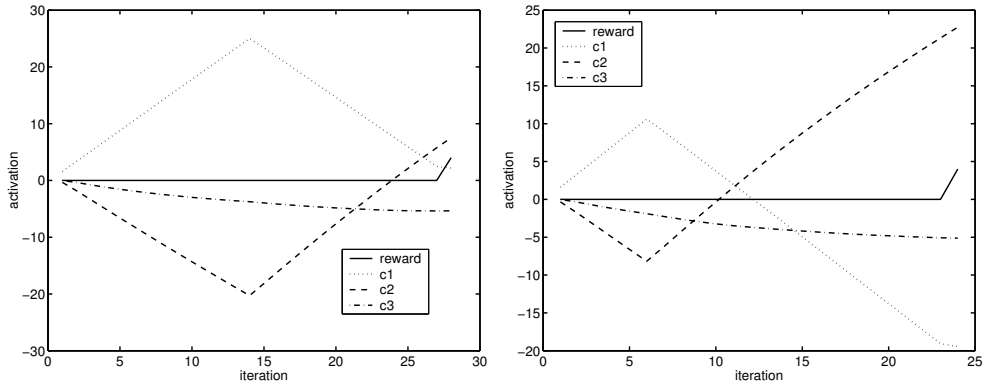

Figure 6.14: CEC activations over time within one episode of the non-regular T-maze task. Fig. a (left). A grammatical episode. Fig. b (right). An ungrammatical episode.

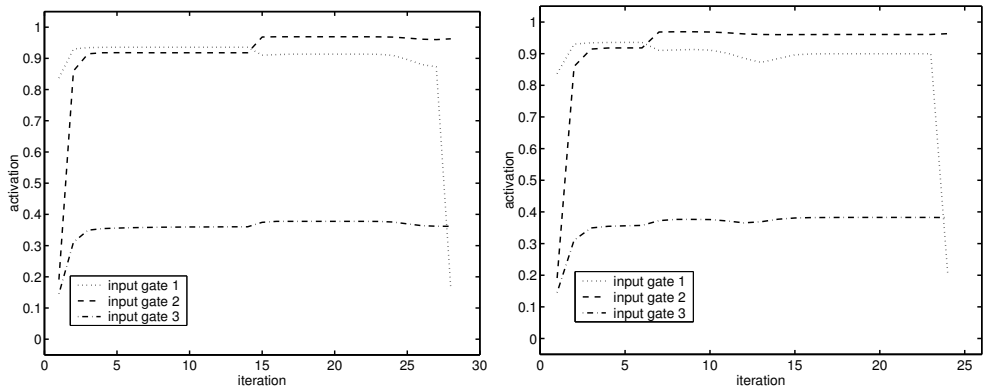

Figure 6.15: Input gate activations over time within one episode of the non-regular T-maze task. Fig. a (left). A grammatical episode. Fig. b (right). An ungrammatical episode.

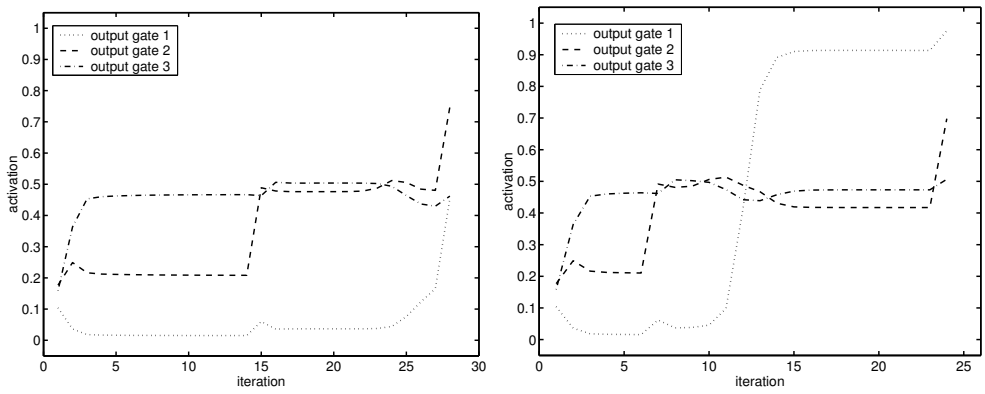

Figure 6.16: Output gate activations over time within one episode of the non-regular T-maze task. Fig. a (left). A grammatical episode. Fig. b (right). An ungrammatical episode.

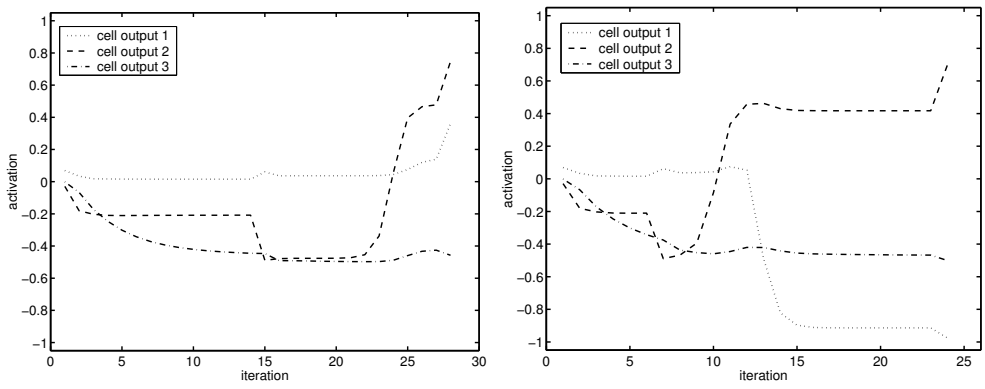

Figure 6.17: Memory cell outputs over time within one episode of the non-regular T-maze task. Fig. a (left). A grammatical episode. Fig. b (right). An ungrammatical episode.

by the reinforcement learning task, namely rewards and penalties obtained when the agent takes the final action at the T-junction. To the best of my knowledge, this is the first time an automaton of a higher computational class than FSAs is induced in a reinforcement learning context. More importantly, the results show that RL-LSTM can learn complex temporal dependencies between combinations of many past events and current actions in partially observable reinforcement learning tasks, and it can learn to induce a large short-term memory.

6.4.4 Multi-mode pole balancing

6.4.4.1 Task and architecture

The next test problem is more realistic than the T-mazes and has more complicated dynamics. It is another variation of the pole balancing task, studied also in the previous chapter. The version used in this experiment is made more difficult by two sources of hidden state. First, as in the previous chapter and in Lin and Mitchell (1993), Schmidhuber (1991c), Moriarty and Miikkulainen (1996), Meuleau et al. (1999), the agent cannot observe the state information corresponding to the cart velocity $\dot{x}$ and pole angular velocity $\dot{\theta}$. It has to learn to approximate this information using its recurrent connections in order to solve the task. Second, the agent must learn to operate in two different modes. In mode A, action 1 is left push and action 2 is right push. In mode B, this is reversed: action 1 is *right* push and action 2 is *left* push. Modes are randomly set at the beginning of each episode. The information which mode the agent is operating in is provided to the agent only for the first second of the episode. After that, the corresponding input unit is set to zero and the agent must *remember* which mode it is in. Obviously, failing to remember the mode leads to very poor performance.

As in the previous chapter, the only reward signal is -1 if the pole falls past $\pm 12^\circ$ or if the cart hits either end of the track. These are also the only points where the episode ends and a new one starts. Note that the agent must learn to remember the mode information for an infinite amount of time if it is to learn to balance the pole indefinitely. Thus, this is a task with strongly hidden state (chapter 3, Williams, 1990). This rules out history window approaches altogether. However, in contrast with the T-maze tasks, the system now has the benefit of starting with relatively short time lags.

Furthermore, the task requires the network to organize its internal state space such that it represents both continuous information (cart and pole velocities) and discrete information (mode of operation). Both this requirement and the requirement to remember information indefinitely have been described by Williams (1990) as examples that illustrate the promise of recurrent neural networks for control. As Williams (1990) notes, it is hard to see how either a system based on traditional control theory (which is mainly concerned with continuous systems, and which sometimes uses fixed size history windows), or a system based on traditional artificial intelligence (mainly concerned with discrete systems) could solve such a task.

The LSTM network has 2 output units, 14 standard hidden units, and 6 memory cells. It has 3 input units: one each for x and θ; and one for the mode of operation, set

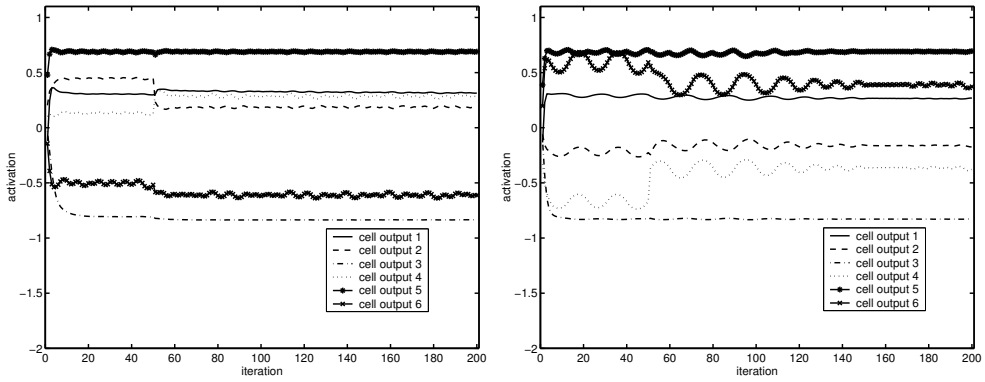

Figure 6.18: Memory cell outputs over time within one episode of the multi-mode pole balancing task. After 50 iterations, the input unit coding for the mode of operation information is switched off. Fig. a (left). Mode A episode. Fig. b (right). Mode B episode. In this particular episode, it takes somewhat longer to dampen the oscillations of the pole, as shown by the oscillations of the memory cell outputs.

to zero after one second of simulated time (50 timesteps). $\gamma = .95$, $\lambda = .6$, $\kappa = .2$, $\alpha = .002$. In this problem, directed exploration was not necessary, because in contrast to the T-mazes, imperfect policies lead to many different experiences with reward signals, and there is hidden state everywhere in the environment. For a continuous problem like this, a table-based memory bit system is obviously unsuitable, so a comparison was only made with the Elman-BPTT system, which had 16 hidden and context units and $\alpha = .002$.

6.4.4.2 Results

The Elman-BPTT system never reached satisfactory solutions in 10 runs. It only learned to balance the pole for the first 50 timesteps, when the mode information is available, thus failing to learn the long-term dependency. However, RL-LSTM learned optimal performance in 2 out of 10 runs (after an average of 6,250,000 timesteps of learning). After learning, these two agents were able to balance the pole indefinitely in both modes of operation. In the other 8 runs, the agents still learned to balance the pole in both modes for hundreds or even thousands of timesteps (after an average of 8,095,000 timesteps of learning), thus showing that the mode information was remembered for long time lags, and the velocity information approximated reasonably well. In most cases, such an agent learns optimal performance for one mode, while achieving good but suboptimal performance in the other.

Figure 6.18 shows the memory cell outputs over time during an episode, for one successful RL-LSTM solution. The left graph corresponds to a mode A episode, the right graph corresponds to a mode B episode. The graph only shows the first 200 iterations of each episode; the agent can balance the pole indefinitely in both modes. After 50 iterations, the input unit coding for the mode of operation information is

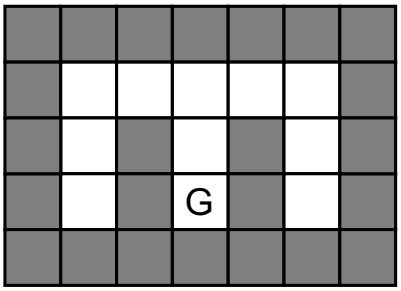

Figure 6.19: The cheese maze. G indicates the goal location, where the only reward of the task is given.

switched off. By that time, the activations of memory cells 2, 4, and 6 are very different in the different modes of operation, and they stay different. This provides the memory for which mode the system is currently operating in. Simultaneously, these same memory cells code, apparently in a distributed way, for (something like) the cart and pole position and velocity information, which is necessary for the actual balancing of the pole.

6.4.5 McCallum's cheese maze

6.4.5.1 Task and architecture

The final two test problems for RL-LSTM are well-known test problems from the POMDP literature. Thus, they allow us to make additional comparisons with other solution techniques.

McCallum (1993) was the first to describe the maze depicted in figure 6.19. The objective is to move to a single goal location, where the only reward of the task is obtained, the "cheese". The agent's actions consist, as above, of single steps to the four cardinal directions North, East, South, and West. The agent only perceives the wall or open space on the four cardinal positions next to it. There are several perceptually aliased states (see figure 6.19). For example, the corridor to the left and the right of the middle T-junction look the same to the agent. It can disambiguate these states by remembering the observation from one timestep ago. If the agent reaches the cheese, the episode ends and the agent goes to a random location in the maze.

McCallum (1993) successfully demonstrated his Utile Distinction Memory on this task. Later, Littman (1994) used this maze to demonstrate tabular Q-learning with memory bits. Wilson (1994) and Cliff and Ross (1994) used it to demonstrate classifier systems with memory bits.

Lanzi (2000) shows that tabular Q-learning with memory bits can suffer from "aliased pay-offs" in this task (see section 3.6.3.7); especially, and perhaps counterintuitively, when the agent explores extensively. The reason is that the expected return in the two aliased states to the left and to the right of the central T-junction is the same, for actions "if a corridor is observed and memory bit 1 is 0, go left" and "if a

corridor is observed and memory bit 1 is 1, go right". This causes the system to have difficulty setting the memory bit correctly before it enters this corridor.

6.4.5.2 Results

The same RL-LSTM system as in the T-maze tasks was used for this task, but now with 4 input units, one for each cardinal direction. $\gamma = .98$, $\alpha = .01$, $\kappa = .2$, $\lambda = .8$. The success criteria were to have the optimal policy using greedy action selection, and to reach the goal in less than 6 steps on average using stochastic action selection, to enforce more or less robust solutions.

The system has little difficulty in finding the solution. In all 10 out of 10 test runs, the agent converged to optimal solutions, in an average of 643,175 iterations. Apparently, RL-LSTM works well on this benchmark POMDP test problem. First of all, this shows that RL-LSTM's success in the tasks described before is not due to particularities in those tasks. Furthermore, the system never has any difficulty with aliased pay-offs. The reason for this is that, unlike in a memory bits system, LSTM's internal state will virtually always be somewhat different for different histories. In the two aliased states to the left and the right of the central T-junction, the internal state will never be the same. Therefore, the expected return for actions "if a corridor is observed and the internal state has value A, go left" and "if a corridor is observed and the internal state has value B, go right", will not be the same.

6.4.6 89-state stochastic office navigation problem

6.4.6.1 Task and architecture

The final test problem in this chapter is a difficult partially observable "office navigation" task, depicted in figure 6.20, and first described in Littman et al. (1995a, 1995b). They used it to test state of the art model-based POMDP algorithms that approximate value functions defined over the belief state space.

It is one of the larger POMDP test problems in the literature, with 89 states. In contrast with the other test problems in this chapter, but like the test problems in the previous two chapters and in the next chapter, the agent has a position and an orientation, to the North, East, South, or West. The agent starts the episode in a random location and with a random orientation. The goal is to move to the goal location in the lower right corner, which contains the only reward of the task. After reaching the goal or taking more than 251 actions, the episode ends and a new episode starts.

The observations come from 4 independent sensors that detect walls vs. open space to the agent's front, back, and sides. However, sensors are very unreliable: a wall is detected only with probability .9, open space is detected with probability .95. Thus, the sensor mistakenly detects open space when there is a wall with probability .10, and a wall when there is open space with probability .05. All sensors make mistakes independently from each other. This means that the probability of getting the correct observation lies between .65 and .81, depending on the agent's location. Even if the correct observation is obtained, the high degree of symmetry in the environment implies that the observation can still correspond to many different states. The only exception is

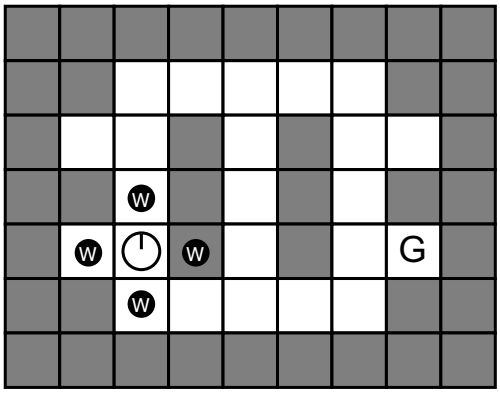

Figure 6.20: The 89-state maze. G indicates the goal location, where the only reward of the task is given. The agent is also depicted, oriented to the north, together with its sensors which detect walls versus open space.

the observation obtained in the goal state, which is unique and deterministic. Together there are only 17 observations for the 89 states.

State transitions are also very stochastic. The actions go forward, turn left, turn right, and turn around are only successful with an average probability of .70, the remaining probability being distributed over the other actions (see Littman et al., 1995b for a complete description of the state transition probabilities). There is an additional action "stay in place" which is always successful, and which can help in reducing uncertainty about the current state, because it leads to another observation in the same state.

The difficulty of this task comes from the relatively large size of the state space, the large amount of stochasticity both in observations and in actions, and the many states that look the same even with correct observations. The large amount of stochasticity requires a system that can handle stochasticity well. This is the first time RL-LSTM is tested on a problem with this nature.

6.4.6.2 Results

All the algorithms investigated by Littman et al. (1995a, 1995b) include a Bayesian belief state estimation component. Based on the belief state, a number of heuristic value function computation approaches were attempted, because the problem is much too large for exact PWLC value computation. They are listed, together with their results, in table 6.1. See chapter 3, specifically section 3.6.2.2, for more details on these algorithms. One algorithm is truncated value iteration. Another, called Q_{MDP} here, is value iteration on the underlying MDP. In the POMDP, the MDP's Q-values are used, weighted by the belief state components. Replicated Q-learning and linear Q-learning are both versions of linear Q-learning on the belief state.

Table 6.1: Results on the 89-state maze task for different methods. As comparisons, both a random walk policy and the results of a human trained on this task are included. The "seeded" algorithms are seeded with the values computed by Q_{MDP}. Results, other than those of RL-LSTM, are taken from Littman et al. (1995a) and Loch and Singh (1998).

	goal%	median
Random walk	25.9	>251
Truncated Value Iteration	44.6	>251
Q_{MDP}	25.9	>251
Replicated Q on the belief state	2.8	>251
Linear Q on the belief state	5.2	>251
Q_{MDP} (no stay)	57.8	40
Replicated Q on the belief state (seeded)	10.8	>251
Linear Q on the belief state (seeded)	58.6	51
Sarsa(λ)	77.0	73
RL-LSTM (best 7/10 runs)	93.9	61
Human (Michael Littman)	100.0	29

The results of these algorithms, displayed in table 6.1, show the proportion of episodes in which the goal is reached by the trained agent and the median number of steps to the goal. The results are not very good. For this reason, the authors experimented with a variation of the linear Q-learning algorithms. Now their Q-values were seeded with the Q-values computed by the Q_{MDP} method before learning started. Furthermore, a variation of Q_{MDP} was investigated where the "stay in place" action was removed, because this action can never be learned by Q_{MDP}. Generally speaking, these variations improved the results, but the goal was still only reached in at best 58.6% of the episodes. Because it is difficult to determine the optimal policy for this task, Littman et al. (1995a, 1995b) also include the results of a human (one of the authors) trained on this same task using a virtual reality environment. The human reached the goal in 100% of the episodes, showing that there is still a lot of room for improvement for the machine learning methods.

Loch and Singh (1998) used a model-free, table-based, memoryless policy learning technique, Sarsa(λ), to demonstrate that memoryless policies may still have good performance in partially observable domains, when eligibility traces are used during learning. In terms of the percentage of episodes where the goal is reached, their method outperforms all algorithms described by Littman et al. (1995a, 1995b). The median of the number of steps to the goal is high, however. This makes sense if we consider that a memoryless policy cannot distinguish between many positions in this environment, and must therefore take many suboptimal actions even if eventually the goal is reached.

The RL-LSTM system used here had the same basic architecture as before. There were 4 input units, one for each cardinal direction (as in the cheese maze), 3 memory cells (as in the cheese maze and all T-mazes), 15 standard hidden units, and 5 output

units. $\alpha = .01$, $\kappa = .5$, $\lambda = .9$. Exactly the same directed exploration mechanism was used as before. The success criterion required the agent to reach the goal in at least 90% of the episodes using the agent's stochastic action selection. 7 out of 10 test runs reached this success criterion, in an average of 14,253,085 iterations. The other 3 test runs did not reach the success criterion, but they all learned to reach the goal in at least 60% of the episodes. In the 7 successful runs, using greedy action selection, the average percentage of reaching the goal and median number of steps are 93.9 and 61, respectively. This is subtantially better than any of the other reported algorithms.

In many of the episodes, the agent first checks the upper left "room" which the agent initially cannot distinguish from the goal location, due to the environment's symmetry. Once it finds out that this is not the goal location, the agent moves more or less directly to the lower right room via the lower left corner. In those cases, it usually does not enter the lower left room—which is something a memoryless policy could not do.

Many authors (e.g. Kaelbling et al., 1998; Littman et al., 1995a; Thrun, 2000) have argued that the proper way to deal with the uncertainty caused by a POMDP is to explicitly *represent* the uncertainty, using belief states which define a probability distribution over all possible states. This experiment shows that RL-LSTM can deal with the uncertainty without explicitly representing the uncertainty and that it is apparently fairly robust against a large amount of stochasticity in the POMDP. The reason for this robustness probably lies in the inherent generalization capabilities of neural networks, combined with the inherently statistical nature of the learning algorithm. LSTM generalizes both over observations and over internal states. If either of them does not have the "expected" activation vector at a certain moment, either because of noise in the observation or because an intended action was not executed, LSTM does not completely collapse but rather tries to produce outputs (Advantage values) that best match this unexpected situation. Furthermore, the inherently statistical nature of the reinforcement learning algorithm implemented using gradient descent gives some robustness against noise during learning. Learning can still be successful as long as the learning algorithm can discover sufficient correlations between observations and internal states on the one hand, and outcomes on the other.

A particularly interesting result in this experiment is the finding that model-free RL-LSTM can outperform, in terms of quality of the learned policy, state of the art methods that already start out with a given, exact model of the environment! Thus, even with the given model, those methods perform worse than RL-LSTM, which has considerably less prior knowledge, and which was not even significantly adapted for this task compared to the networks used in the previous test problems. This result provides some further support for the claim, described extensively in chapter 3, that in some cases it may be easier to learn values or policies directly—model-free methods—than to use a model of the environment and apply a model-based method.

6.5 Discussion

The results presented in this chapter show that model-free reinforcement learning with a Long Short-Term Memory recurrent neural network (RL-LSTM) is a promising

approach to solving difficult POMDPs. The POMDPs may have many states, discrete as well as continuous states, a large amount of stochasticity, and, most importantly, complex and long-term dependencies between past events and current best actions.

RL-LSTM's main power is derived from LSTM's property of constant error flow, which other recurrent neural network architectures, such as the Elman networks of the previous chapters, do not have. Unlike other model-free POMDP solution methods which use short-term memory, such as a system based on memory bits or FSAs, RL-LSTM explicitly computes gradients of returns with respect to past experiences, which allows for effective, directed credit assignment.

For good performance in reinforcement learning tasks, the combination of LSTM networks with Advantage(λ) learning and directed exploration was crucial. The benefit of using Advantage(λ) learning instead of other value function-based reinforcement learning algorithms in POMDPs was discussed extensively in the previous chapter. Directed exploration is beneficial because it allows exploration to be low when it can, leading to more stable sequences of experiences in which LSTM can better find the temporal regularities which are used to reconstruct the environmental state signal.

Chapter 7

Reinforcement learning with unsupervised event extraction

Summary

This chapter describes how the techniques presented in the previous chapters can be applied to more realistic robot learning tasks, by combining those techniques with an unsupervised learning mechanism that does event extraction. The event extraction mechanism, ARAVQ, transforms the robot's continuous, noisy, high-dimensional sensory input stream into a compact sequence of high-level "events". The resulting overall system is a hierarchical control system in which the reinforcement learning component, an LSTM recurrent neural network, learns high-level actions in response to the history of high-level events. The high-level actions select low-level behaviors which take care of real-time motor control. Experiments are presented based on a Khepera mobile robot simulator which illustrate the ideas.

7.1 Introduction

The techniques described in the previous chapters constitute an intuitively appealing approach to developing control programs for real robots. Real robots are notoriously difficult to program. The techniques of the previous chapters do not require a person to program the desired behavior by hand, or even show the robot desired actions in different situations. Instead, they potentially allow the robot to learn correct behavior virtually autonomously, based only on scalar reward signals if the robot reaches particular goals. Furthermore, the fact that in most realistic robot domains the state is only partially observable (see section 3.3.4) is not a problem for the techniques of the previous chapters, which were explicitly designed to deal with partial observability.

However, the environments considered in the previous chapters were fairly small, and usually discrete. In contrast, a robot's environment is inherently large and continuous. Combined with partial observability this can lead to extremely long time lags, either between actions and finally achieving a goal, or between events that need

to be remembered and actions that depend on them. As we saw before, long time lags make temporal credit assignment very difficult: which of the many past actions are to be credited for good behavior, and which of the many past events need to be remembered? In the previous chapter, we saw that LSTM networks can be used to learn long-term dependencies in POMDPs. However, as was also apparent, there is still a limit to the time lag that can be bridged, even with LSTM. For larger and more realistic robot problems, more is needed.

This chapter attempts to make some progress on the problem of learning in realistic robot domains by combining different techniques. The main idea is to combine reinforcement learning, as it was used in the previous chapters, with unsupervised learning. This idea was also suggested, in a different context, by Hinton (1999) when he said about the brain:

> There's probably some basic way of unsupervised learning and once it constructed representations that way, there's probably some basic way of reinforcement learning based on these representations.

In this chapter, the unsupervised learning method accomplishes both spatial and temporal abstraction (or compression) of the robot's sensory inputs. A high-level representation of the environment is thus obtained in which reinforcement learning becomes much more feasible than in the raw, low-level representation. The unsupervised learning method does not in itself solve the problem of partial observability, but it does make it easier. In the spatially and temporally compressed representation it is easier for the reinforcement learning system to find the temporal dependencies which can disambiguate the state. As in the previous chapter, the reinforcement learning component is based on an LSTM recurrent neural network, which was shown in the previous chapter to be good at learning such temporal dependencies in partially observable reinforcement learning tasks.

The next section describes the overall system in more detail, and discusses related work. Section 7.3 presents results on experiments done using a Khepera robot simulator. Section 7.4, finally, contains a general discussion.

7.2 The learning system

7.2.1 Unsupervised event extraction

One component of the overall learning system is an unsupervised learning system doing *event extraction*. Robots typically have many, noisy sensors, sampled at a high rate, e.g. 20 Hz. From this timeseries of many noisy, high-dimensional data points, the event extraction mechanism extracts a compact sequence of high-level "events" or "concepts". Figure 7.1 shows the general idea. The mechanism accomplishing this is called an Adaptive Resource Allocation Vector Quantization (ARAVQ) network (Linåker & Niklasson, 2000; Linåker & Jacobsson, 2001). It basically matches incoming sensor vectors, i.e. the values coming from the different sensors, to stored model vectors in a manner similar to Kohonen maps (Kohonen, 1995), and it dynamically allocates new model vectors when it encounters novel and stable situations.

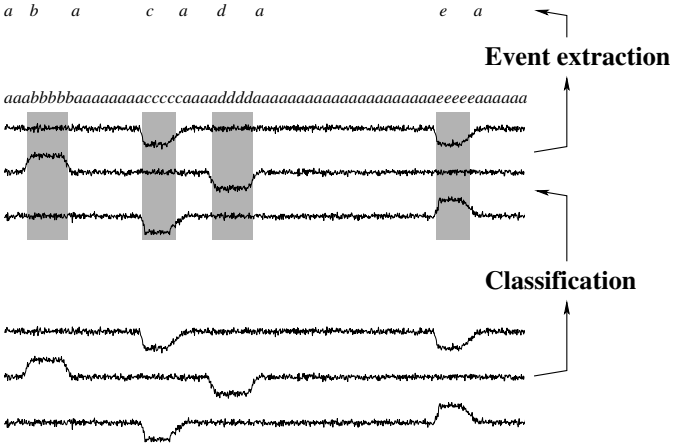

Figure 7.1: A schematic representation of event extraction. The horizontal dimension represents time. First, the continuous sensor vector is classified. Next, only changes in classification are passed on as significant events.

ARAVQ stores the last n input vectors in an input buffer. The values in the input buffer are averaged to create a more reliable, filtered input $\overline{x}(t)$ to the rest of the ARAVQ network. See figure 7.2 for the architecture. There is a set $M(t)$ of model vectors (each representing an event class), which is initially empty. The ARAVQ network operates in three stages:

Event class incorporation. Additional model vectors are only allocated when *stable* and *novel* inputs are encountered, i.e., when the following criteria are fulfilled. The input is considered stable if $d_{\overline{x}(t)}$, the average Euclidian distance between $\overline{x}(t)$ and the last n inputs, is below a threshold parameter ϵ. The input is considered novel if $d_{M(t)}$, the average Euclidean distance between the best matching stored model vector and the last n inputs, is larger than $d_{\overline{x}(t)}$ plus a distance parameter δ. If both of these criteria are met, the filtered input is incorporated as an additional model vector:

$$M(t+1) = \begin{cases} M(t) \cup \overline{x}(t) & d_{\overline{x}(t)} \leq min(\epsilon, d_{M(t)} - \delta) \\ M(t) & \text{otherwise.} \end{cases} \tag{7.1}$$

Classification. Each time-step, a winning model vector $v(t)$ is selected, indicating which class the filtered input currently matches:

$$v(t) = \arg \min_{1 \leq j \leq |M(t)|} \{||\overline{x}(t) - m_j||\}; m_j \in M(t). \tag{7.2}$$

Only when the winning model vector changes, an event is "thrown" (as depicted in figure 7.1, top layer).

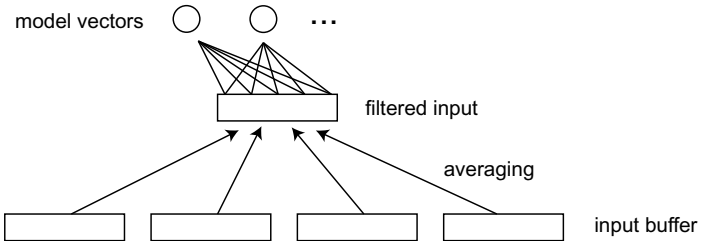

Figure 7.2: Schematic architecture of the ARAVQ network. The raw sensory inputs stored in the input buffer are first averaged to yield a filtered input. The filtered input is compared to the stored model vectors. If no good match is found, a new model vector is allocated corresponding to the filtered input.

Adaptation. If the winning model vector matches the filtered input very closely, the filtered input is considered to represent a "typical" instance of the class, and the model vector is modified to become even more like this input, in a manner similar to Kohonen maps:

$$\Delta m_{v(t)} = \begin{cases} \alpha[\overline{x}(t) - m_{v(t)}] & ||\overline{x}(t) - m_{v(t)}|| < \frac{\epsilon}{2} \\ 0 & \text{otherwise,} \end{cases} \tag{7.3}$$

where α is a learning rate parameter.

Figure 7.3 illustrates the functioning of ARAVQ on a simple, one-dimensional sensor signal. In summary, the ARAVQ network accomplishes two things that are good for reinforcement learning. First, it compresses and simplifies the high-dimensional sensor space and filters out much of the noise, producing clear-cut higher level concepts representing distinct, stable situations. Second, it compresses the long, high sampling rate timeseries into a short sequence of discrete events. In the present study, the ARAVQ learning phase precedes the reinforcement learning phase.

7.2.2 Reinforcement learning on the extracted concepts

The events or concepts coming out of the unsupervised learning system are the inputs to the reinforcement learning component. We are dealing with partially observable domains. The robot's sensory input is preprocessed using the event extraction mechanism, but different hallways may still look the same and lead to one extracted concept "hallway". This is why the reinforcement learning component is, as in the previous chapters, based on a recurrent neural network. The same basic architecture as in the previous chapter is used, an RL-LSTM network (depicted in figure 6.1). The RL-LSTM network again approximates the value function of Advantage(λ) learning. Furthermore, the same directed exploration technique is used as in the previous chapter.

The input of the RL-LSTM network basically consists of the output units of the ARAVQ network. Thus, each input is a vector with one unit set to 1 (the winning

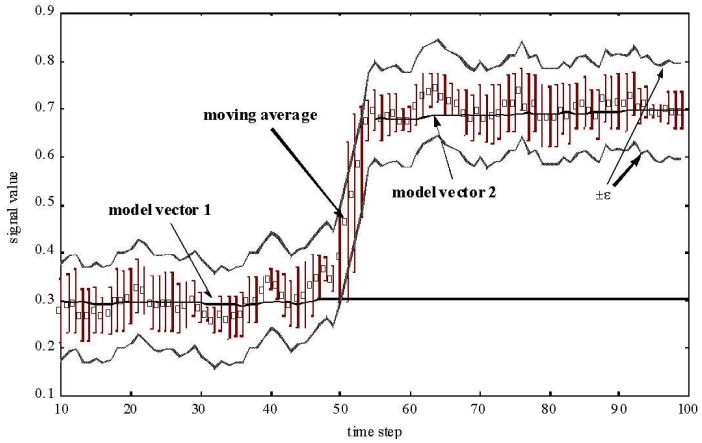

Figure 7.3: An illustration of the functioning of ARAVQ on a simple, one-dimensional sensor signal. For roughly the first half of this timeseries, the input is stable (falls within the boundaries set by ϵ). Then the input changes suddenly, and for a while ARAVQ considers the input too unstable. Next, the input more or less settles down and the stability criterion is again fulfilled. The new stable input is closer to stored model vector 2, therefore the categorization changes and a new event is thrown at that point.

concept) and the remaining ones set to 0. The robot's world state is approximated by the current event, provided by the ARAVQ network, together with the recurrent activations within the LSTM network, which can represent a complex function of the entire history of past events.

7.2.3 Hierarchical control

The outputs of the reinforcement learning LSTM network represent Advantage values of high-level actions in response to event changes detected by the ARAVQ network. To provide a connection to the low level, real-time motor control of the robot, the whole system is organized as a hierarchical control system. The entire hierarchical control system, which has obvious similarities to Brooks' subsumption architecture (Brooks, 1991a), is depicted in figure 7.4.

Low level, real-time motor control is handled by a simple, handcrafted reactive controller. On this level, there are three behaviors: going forward, following the corridor; following the left wall and turning left when possible; and following the right wall and turning right when possible. Each can correct for some noise in the execution of motor commands and avoid collisions, by directly using the raw sensory inputs as feedback. A low level behavior is executed until a new event is detected by the ARAVQ network. At such a point, the high-level RL-LSTM controller *selects* (or gates) one of the three low level behaviors for execution until the next event.

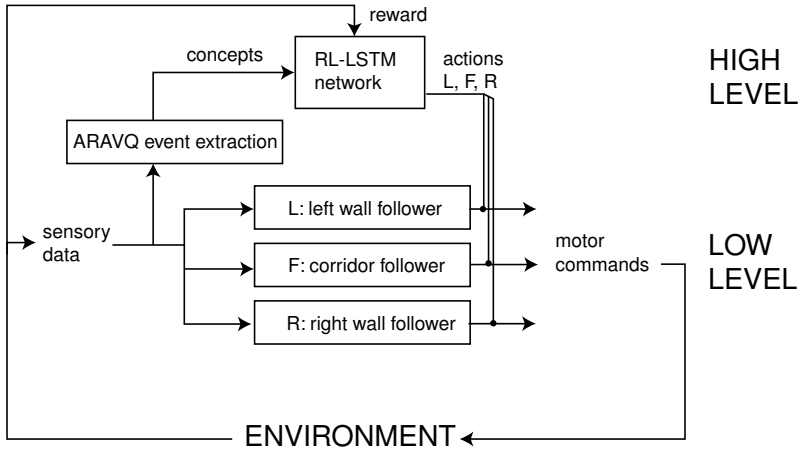

Figure 7.4: The complete hierarchical control system. In response to events detected by ARAVQ, RL-LSTM produces high-level actions which select low-level behaviors.

7.2.4 Related work

The ARAVQ unsupervised event extraction mechanism (Linåker & Niklasson, 2000; Linåker & Jacobsson, 2001) was inspired by Tani & Nolfi's (1998) event extraction mechanism , which in turn was inspired by Schmidhuber's (1992a) history compression mechanism. Tani & Nolfi's system attempts to predict the next input given the complete history of past inputs, whereas ARAVQ simply attempts to detect changes from the last (stable) input to the next. Tani & Nolfi's system is sensitive to the density of patterns: sometimes a very distinct but rare situation does not get its own concept. Furthermore, they required many iterations of learning. ARAVQ can often learn good concepts in just a single pass through the environment.

The idea of preprocessing raw sensory data for reinforcement learning using an unsupervised learning mechanism has been explored by a number of authors. For instance, Fernández & Borrajo (1999) use a standard vector quantization technique to categorize high-dimensional input vectors into a fixed number of concepts, before doing Q-learning on the concepts. Arleo, Smeraldi, Hug, and Gerstner (2001) extract high-level visual features from video images using unsupervised learning and do Q(λ)-learning using those features. None of these authors perform the type of temporal abstraction employed here; they only perform spatial abstraction over the state space, classifying the current raw sensory input. In fact, such spatial abstraction without temporal abstraction is the focus of a large number of studies in reinforcement learning on generalization (Sutton & Barto, 1998, and see chapter 3). However, in most of that work, unlike our work and that of Fernández & Borrajo (1999) and Arleo et al. (2001), there is no separate unsupervised learning component.

Temporal abstraction in reinforcement learning is the focus of another related body of literature, which is concerned with hierarchy. In hierarchical Q-learning (Wiering

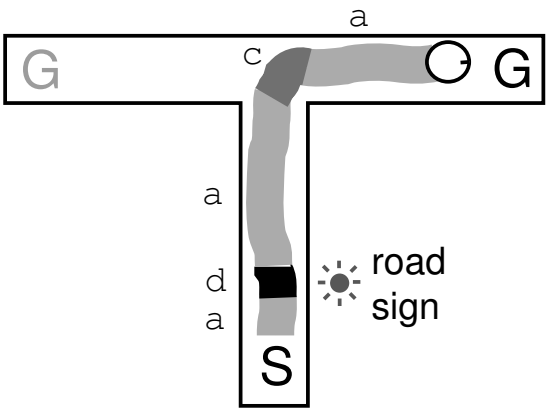

Figure 7.5: The T-maze, depicted together with the simulated Khepera robot and the events it detects along the way from the starting position S to the current goal position G (the "wrong" goal position is shown in gray).

& Schmidhuber, 1997, see section 3.6.3.6), the framework of options (Sutton, Precup, & Singh, 1999), MAXQ (Dietterich, 2000), Hierarchical Abstract Machines (Parr & Russell, 1998), and Feudal reinforcement learning (Dayan & Hinton, 1993), high-level actions correspond to entire policies at a lower level, which are executed until a termination criterion is fulfilled. Hierarchical Q-learning is also aimed specifically at POMDPs. Hernandez-Gardiol & Mahadevan (2001) make a similar argument to the one made in this chapter, stating that temporal abstraction is particularly important in partially observable reinforcement learning domains in order to deal with long time lags between significant past events and current decisions. Where we use an LSTM recurrent neural network to learn the temporal dependencies between high-level concepts, they use the Nearest Sequence Memory and Utile Suffix Memory algorithms (see sections 3.6.3.4 and 3.6.3.5). Another important difference is that they use *hand-coded* high-level concepts, whereas we use unsupervised learning to let the robot develop the high-level concepts autonomously.

7.3 Experiments

7.3.1 T-maze

As a first illustration of the ideas presented above, let us consider a partially observable T-maze navigation task, similar to the one investigated in the previous chapter. The version investigated here is implemented in a simulation of a Khepera mobile robot equipped with 8 noisy infrared proximity sensors and 2 light sensors, driving around in a continuous environment (Linåker & Jacobsson, 2001).

The robot's task is to move from the fixed starting position to a goal position (see figure 7.5). The goal position can either be to the left of the T-junction or to the right.

The goal position can itself not be perceived—this makes the task partially observable. However, the goal position depends on a road sign that the robot can perceive when it traverses the corridor. If the robot reaches the goal position, it receives a reward: $r = 4$. If, instead, it reaches the opposite position, it receives a negative reward: $r = -.5$. In both cases, the episode ends and a new one starts, with the goal position assigned randomly to the left or the right. No other rewards are available.

As before, the main difficulty in this task lies in discovering the temporal dependency between the road sign and the action at the T-junction, storing the road sign information reliably in short-term memory, and using that information at the correct time, while ignoring the irrelevant intermediate sensory inputs. In this continuous version, there are even more intermediate sensory inputs than in the version studied in the previous chapter, because sensors are sampled at a high rate. Furthermore, the sensory inputs of the Khepera robot are very noisy, which makes it more difficult still. Thus, despite its conceptual simplicity this is a difficult long time lag problem. Even though RL-LSTM was shown, in the previous chapter, to be capable of solving this problem with time lags of up to 50 to 70 timesteps, when the time lag becomes larger, which is easily the case with sensors sampled at 20 Hz, learning becomes very difficult.

This is a case where the value of event extraction is particularly clear. Figure 7.5 shows the extracted events during a sample path. Reinforcement learning is done on the basis of the extracted events rather than the raw sensory inputs, using the system described in the previous section. The time lags in the problem are significantly reduced and most of the sensory noise is filtered out. The RL-LSTM network can find the temporal dependency between the road sign and the T-junction relatively easily, because at this higher abstraction level the time lag between them is only two events.

10 runs were performed, using RL-LSTM networks with 4 input units (one for each concept), 3 output units (one for each action), 12 hidden units, and 3 memory cells. The following parameter values were used: $\delta = .7$, $\epsilon = .2$, $n = 5$, $\alpha = .05$, $\gamma = .98$, $\lambda = .8$, $\kappa = .2$, and $\beta = .01$.

All runs converged to optimal policies, using 19,082 iterations on average. Figure 7.6 shows the probability of reaching the goal as a function of the number of iterations, for one typical run (averaging over all 10 runs is not very insightful because different runs discover and rapidly converge to the correct policy at different moments). It is based on the robot choosing high-level actions according to its exploration mechanism, which is why it does not reach 100% correct; the derived greedy policy is optimal. Figure 7.6 also shows the average number of high-level actions to the goal under the online exploration strategy, provided the goal is reached, as a function of the number of iterations for the same run.

7.3.2 Complex maze

The second test problem is a more complex maze navigation task (see figure 7.7). The principle, however, is the same. The agent must move to one of two possible goal positions, depending on a road sign it can only observe near the starting position. Again, $r = 4$ when the robot reaches the goal, and $r = -.5$ when it reaches the opposite position. However, now there are many more states and longer paths to the goal, and ARAVQ finds more concepts. Furthermore, even if the robot reliably learns

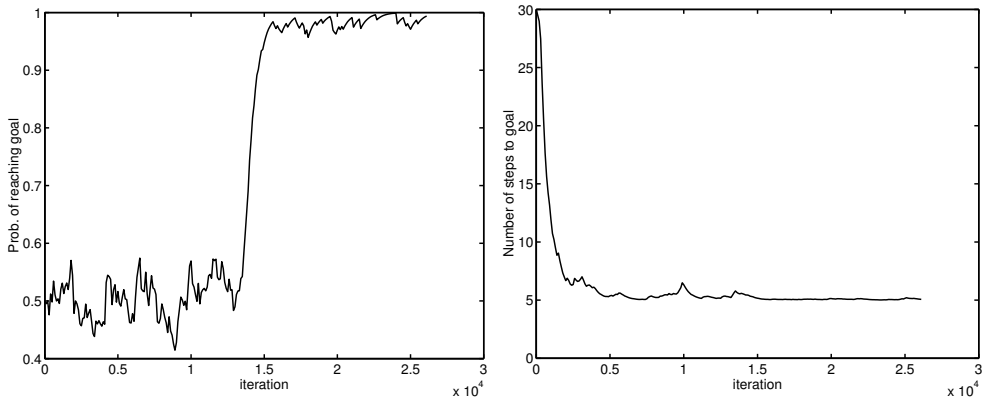

Figure 7.6: Results for the T-maze task. Fig. a (left). Probability of reaching the goal in the T-maze task as a function of the number of iterations for one run. Fig. b (right). Average number of high-level actions to the goal, provided the goal is reached, as a function of the number of iterations for one run.

to move from the starting position to the final T-junction before the goal position, the minimum time lag between the road sign and the final T-junction is 6. Thus, although the time lags are considerably reduced thanks to ARAVQ, this is still a relatively long-term dependency POMDP—but now even on the level of extracted events rather than the level of low-level sensory inputs. A similar task was investigated in Linåker and Jacobsson (2001), using a supervised learning Elman recurrent neural network to learn the high-level actions. The Elman network could not solve the task when the time lag exceeded a few events. LSTM, on the other hand, can solve such tasks, as we saw in the previous chapter, and it does so in this particular task.

10 runs were performed, using the same RL-LSTM architecture as in the simpler T-maze, except the number of input units is now 10, because there are now 10 concepts. All runs converged to optimal greedy policies, always reaching the correct goal position in the lowest possible number of high-level actions. The system needed considerably more time than in the simple T-maze, taking 7,283,302 iterations on average. Figure 7.8 shows the probability of reaching the correct goal position as a function of the number of iterations for one typical run. Figure 7.8 also shows the average number of high-level actions to the goal under the online exploration strategy, provided the goal is reached, as a function of the number of iterations for the same run. Note that this number remains much higher than the number under the greedy policy, because there is a lot of exploration, and even just one exploring suboptimal action often leads to many extra actions to arrive at the goal.

For the run depicted in figure 7.8, figure 7.9 shows the values of the memory cells' internal states z_j during the episode, *after* the optimal policy has been learned. The robot first encounters an episode where the goal is in the left position, and then an episode where the goal is in the right position. It is apparent that when the robot sees the "left" road sign the internal state of memory cell 2 takes on a different value

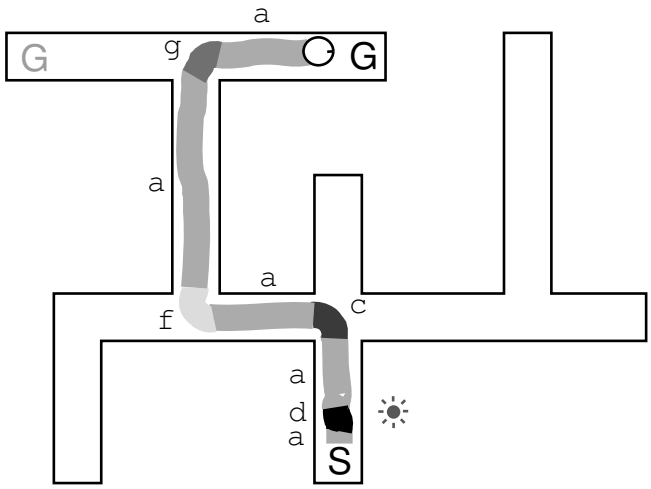

Figure 7.7: The more complex maze, depicted together with the robot taking the optimal path to the right goal position. The number of events between the road sign and the final T-junction is 6.

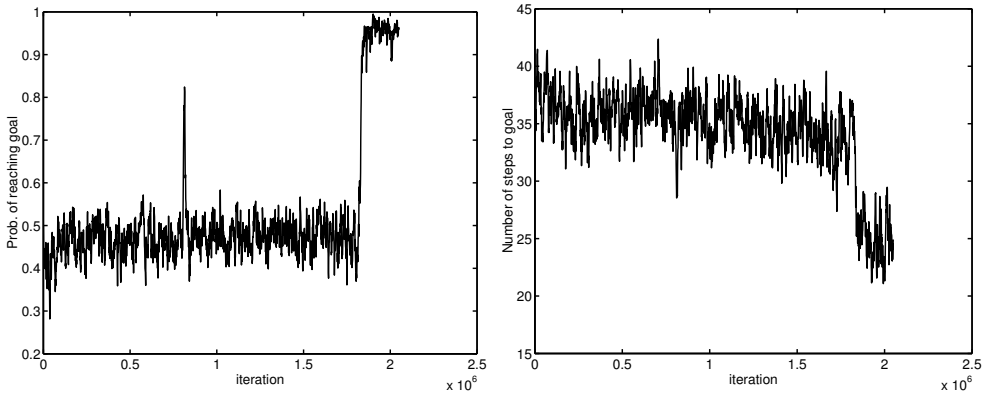

Figure 7.8: Results for the complex maze task. Fig. a (left). Probability of reaching the goal in the complex maze task as a function of the number of iterations for one run. Fig. b (right). Average number of high-level actions to the goal, provided the goal is reached, as a function of the number of iterations for one run.

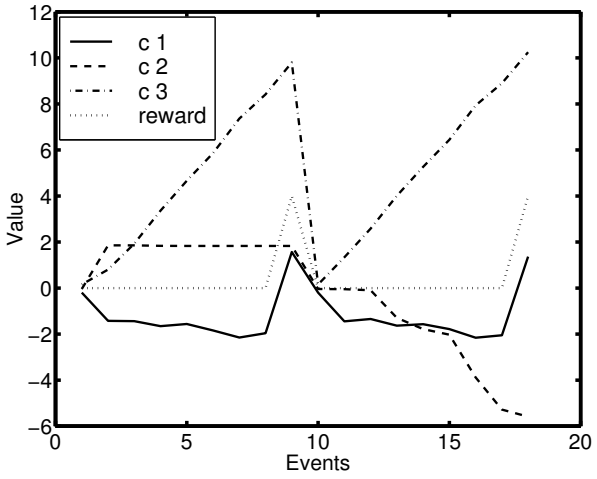

Figure 7.9: The memory cells' internal states z_j during two episodes, the first with the goal to the left and the second with the goal to the right. The reward signal is also shown. After the first reward, the new episode starts (event 10).

than when it sees the "right" road sign. This internal state uniquely characterizes the two conditions throughout the episode, and it is used by the network at the final T-junction to decide whether to turn left or right.

7.4 Discussion

The results presented in this chapter show that certain continuous, partially observable robot learning tasks which are very difficult in their "raw" version can be solved using the combination of reinforcement learning with unsupervised learning. The key elements are the ARAVQ event extraction network that accomplishes a spatial and temporal abstraction of the environment, the RL-LSTM recurrent neural network that can learn the POMDP's temporal dependencies, and their combination into a hierarchical control system.

However, there are still several limitations. First of all, learning is still fairly slow, if we were to apply the algorithms to real robots. This is a problem for most reinforcement learning algorithms and most neural networks. This probably reflects more the generality and lack of "bias" or built-in knowledge of the used algorithms than it reflects inherent problems in the algorithms per se; but this issue still needs to be addressed if we are to apply these techniques to real robots in real-world tasks. One approach would be to use *experience replay* (Lin & Mitchell, 1993), which stores past experiences and uses them to keep training the neural network concurrently with new experiences. Another approach would be to introduce bias or a priori knowledge in some way, for example by providing the system with approximate or partial policies. Both of these approaches are related to what seems like a generally sensible approach

to apply learning to real robots, which is to do a lot of training of the robot controller off-line in simulation, and only then to continue training or to finetune the controller on the actual robot (see section 3.2.7).

A second limitation of the system proposed in this chapter has to do with the fact that here unsupervised learning was done independently from reinforcement learning. The reinforcement learning system operated on the assumption that the concepts extracted by the unsupervised system were sufficient for the task. However, we can easily imagine cases where the unsupervised system fails to distinguish perceptually similar but conceptually different situations, or conversely, distinguishes perceptually different but conceptually similar situations. Ideally, we would want the event extraction mechanism to make only the necessary and sufficient distinctions, in the same way as the RL-LSTM network attempts to do only useful state estimation. This could conceivably be achieved by having some kind of feedback from the reinforcement learning system to the event extraction system, telling it how "useful" the currently extracted concepts are. In the extreme, this would go back to a monolithic reinforcement learning system in which rewards determine how the system generalizes over its observations, e.g. a system based on a neural network as described in the previous chapters (see also sections 3.5.2.9 and 3.6.3.8). This is an approach that is hard to scale up to more realistic robot tasks, which was the very reason for proposing the dual system described in this chapter. However, we could envision an intermediate type of system in which, similar to this chapter, a separate component does fairly aggressive clustering of raw inputs, receiving only limited feedback concerning obtained rewards from the reinforcement learning component which finetunes the extracted concepts.

In general, this chapter was concerned with what I perceive as very important and strongly related issues: POMDPs, hierarchy, and concept formation. Hierarchical control is necessary for dealing with complex problems, because it facilitates the tasks for all levels in the hierarchy. Higher levels benefit from using more abstract inputs than lower levels; this is where concept formation comes in. However, abstraction over inputs implies that sensory information is thrown away and that therefore the task may effectively become a POMDP at the higher level. Conversely, in POMDPs it is important to consider past events at an appropriate grain size—suggesting the benefit of a hierarchy of control in which each level can have memory appropriate for its own level. Furthermore, a hierarchy of control may facilitate the solution of POMDPs because it may cut up the overall task into (more) Markovian subtasks (Wiering & Schmidhuber, 1997). In conclusion, the issues of POMDPs, hierarchy, and concept formation, and especially their combination, deserve much further study.

Chapter 8

Conclusions

8.1 Contributions of this thesis

The contributions of this thesis (and any thesis) can be classified into two categories: technical and conceptual contributions. As with so many distinctions, this distinction is somewhat arbitrary and fuzzy, but useful nevertheless. By technical contributions I mean new or improved algorithms and experiments showing that particular algorithms can be used to solve certain challenging problems or are better than other algorithms for those problems. Conceptual contributions are contributions that are not technical ones in the sense defined above, but that are still contributions because they consist of arguments and demonstrations of interesting points and principles.

8.1.1 Technical contributions

8.1.1.1 Advantage(λ) learning

An extension of the Advantage learning reinforcement learning algorithm (Harmon & Baird, 1996), Advantage(λ) learning, was described in chapter 5. A proof was presented of the equivalence of the forward and backward view of Advantage(λ) learning (appendix B). It was shown how Advantage(λ) learning and the corresponding eligibility traces (the backward view) can be implemented in various types of recurrent neural networks (chapters 5 and 6). The power of Advantage(λ) learning combined with recurrent neural networks in both continuous-time and discrete-time partially observable reinforcement learning tasks was demonstrated empirically. Among other experiments, Advantage(λ) learning was compared with Q(λ) learning, and the effect of λ was systematically investigated.

8.1.1.2 Input-output FSA extraction

An algorithm was described in chapter 4, and more extensively in appendix A, to extract input-output Finite State Automata (Moore and Mealy machines) from recurrent neural networks. It extends existing input FSA extraction algorithms, and it uses a modification of the Hopcroft FSA minimization procedure. The usefulness of this

algorithm lies in its application to the analysis and understanding of recurrent neural networks. This was demonstrated in chapters 4 and 5, where the algorithm was used to gain a better understanding of the functioning of Elman recurrent neural networks trained using reinforcement learning.

8.1.1.3 Reinforcement learning with Long Short-Term Memory

Reinforcement learning was combined with Long Short-Term Memory (LSTM) recurrent neural networks in chapters 6 and 7. Specifically, LSTM networks were used to approximate the value function of Advantage(λ) learning. This combination allows for successful learning in partially observable tasks with complex and long-term dependencies between relevant events, and with significant stochasticity. Empirical comparisons were made with Elman networks, systems based on memory bits, and also with model-based approaches. The results show that the reinforcement learning LSTM networks compare favorably with those alternatives in the investigated settings.

8.1.1.4 Non-regular reinforcement learning

One of the tasks the reinforcement learning LSTM network learned was a task in which it effectively had to recognize, in its history of observations, a non-regular language to learn the optimal policy (section 6.4.3). Non-regular languages cannot be recognized by finite state automata but require at least an automaton with a stack or counter. To the best of my knowledge, this is the first time a reinforcement learning system has learned, based only on rewards obtained at a final goal state, to induce an (admittedly imperfect) automaton of a higher computational class than a finite state automaton.

8.1.1.5 Adaptive exploration based on estimating uncertainty of values

A new adaptive, directed exploration method was described in chapter 6. It uses a feedforward neural network which learns to identify those states about which the system is uncertain in terms of expected returns from those states. This leads to more stable action selection in states with predictable values, and consequently, to more stable sequences of observation-action pairs, in which the reinforcement learning LSTM networks of chapters 6 and 7 could more easily detect the temporal regularities in the POMDP and reconstruct useful state signals.

8.1.1.6 Reinforcement learning with unsupervised event extraction

A system was described in chapter 7 which combines an unsupervised learning event extraction component with a reinforcement learning component. The event extraction component feeds a spatially and temporally compressed representation of the agent's environment into the reinforcement learning component. The reinforcement learning learns high-level actions on the basis of this abstract representation, selecting low-level behaviors for execution until the next high-level action. In this way, it is possible for the overall system to learn policies in more complex and realistic domains.

8.1.2 Conceptual contributions

Besides the technical contributions, described above, there are conceptual contributions. Conceptual contributions are always harder to pinpoint exactly, but it presents the opportunity to highlight a few general points that were made in this thesis and that I think are interesting. Several relate to specific issues discussed before in the general chapters about adaptive behavior research (chapter 2) and reinforcement learning (chapter 3), especially the issues mentioned in chapter 3's final discussion section.

8.1.2.1 A defense of the adaptive behavior approach

Chapter 2 presented an extensive description of the ideas underlying adaptive behavior research and arguments for its relevance as an approach to cognitive science and artificial intelligence. The rest of the thesis can be read as an illustration of many of these ideas and arguments. In line with adaptive behavior principles, agents were constructed with complete perception to action loops and close interaction with their environments. The agents were developed using learning, their decision making processes were decentralized and distributed, and post hoc analyses were done to understand the learned mechanisms.

8.1.2.2 Extending agents with internal state

The main focus of this thesis, reinforcement learning in POMDPs, relates directly to a central debate in adaptive behavior research concerning representation-hungry tasks, the nature of representations, and the usefulness of world models. With respect to this debate, this thesis makes the following contribution. Even though the classical adaptive behavior argument is true that agents can often obtain good performance by exploiting regularities of the world and "using the world as its own best model", there are also many cases where "using the world" directly, reactively, is not enough and the agent must use some form of internal state. In reinforcement learning terms, these "representation-hungry" cases correspond to POMDPs. However, in contrast with traditional artificial intelligence and cognitive science ideas, this internal state does not have to correspond to full-blown, symbolic world models. Instead, agents may learn to successfully deal with representation-hungry tasks by inducing subsymbolic, possibly continuous-valued, possibly distributed internal states which are geared towards particular tasks, without resorting to traditional AI's full-blown, symbolic world models.

8.1.2.3 Cognitive maps

One of the domains in which the need for a world model seems most pronounced is navigation. The world model in this domain would correspond to a "cognitive map". Experiments and analysis suggested that the reinforcement learning agents based on Elman recurrent neural networks of chapter 5 learned cognitive map-like capabilities, without a straightforward or explicit cognitive map being identifiable in the internal mechanisms. This does not show that animals' and humans' brains do not contain straightforward cognitive maps, but it does show that it is possible that a neural system

has similar capabilities using different mechanisms than a straightforward cognitive map. In general, it suggests that one should be careful in deducing from behavioral data conclusions regarding the type of cognitive/neural mechanisms to be found in brains.

8.1.2.4 A model-free approach to solving POMDPs

On a more general level than cognitive maps, the results of this thesis demonstrate that a model-free reinforcement learning approach can work well for challenging POMDPs with complex, long-term temporal dependencies and significant stochasticity. Rather than using or learning a predictive model of the environment and applying model-based POMDP solution techniques to that model, the model-free approach taken in this thesis takes a more direct route to finding good internal state-based policies. The agents learned to exploit the history of observations and actions experienced during an episode and induce short-term memory (internal state) which helped to solve the specific tasks at hand, without inducing full-blown world models. In a way, the agents only learned algorithms for solving tasks, rather than first learning to predict the world and then deriving algorithms for solving the tasks from those predictions. An important advantage of this approach is that it may be very difficult to induce full-blown world models, whereas learning what bits of information to put selectively into short-term memory to solve certain tasks may be feasible and require far less memory.

8.1.2.5 Utility-based state estimation

All agents proposed in this thesis employ a unified approach, based on recurrent neural networks, to the problem of having too little and/or too much perceptual information coming from the environment. The network attempts to generalize over current observations and throw away irrelevant details when it can, using its feedforward connections; and at the same time it attempts to compensate for incomplete information in single observations by exploiting memory of past events, using its recurrent connections. The "decisions" on how to generalize and what information to store in and use from short-term memory are based on experienced rewards. Thus, the interesting end result is a system which attempts to construct, both from its current observation and from past observations and actions, a state signal which is "useful" with respect to its goals: utility-based state estimation.

8.1.2.6 The trade-off between perception and internal state

The work of chapter 4 is related to this idea of utility-based state estimation. This chapter showed that even in tasks where internal state is not strictly necessary (MDPs), internal state can still make learning easier. In particular, this is the case when using the current observation requires the agent to solve a complex pattern recognition problem, as is often the case in MDPs with many states and factored state representations, which must be generalized over. This point was experimentally demonstrated. First of all, this provides an argument against the tendency in the reinforcement learning community to practically equate MDPs with perception-based, memoryless policies. Of course, the complexity of inducing useful internal states is also variable, depending

on temporal regularities in the environment. We can conclude that there is a trade-off between perception and internal state in many reinforcement learning and other machine learning tasks.

8.1.2.7 Relative value function approximation

In this thesis, the estimated states were used for value function approximation. Arguments were presented in chapter 3, specifically in sections 3.6.3.8 and 3.7.3, why value function approximation is a good approach in reinforcement learning tasks in general and even in the particular case of POMDPs, and the subsequent chapters illustrated these arguments in tasks with various properties and difficulties. In fact, recent neuroscience studies suggest that the brain uses some kind of value functions (Schultz et al., 1997; Houk et al., 1995).

However, the value function of the reinforcement learning algorithm that in many examples in this thesis worked best, Advantage(λ) learning, is different from the typical or "real" value functions of, for instance, dynamic programming, TD(λ)-learning, and Q(λ)-learning. Advantage values are something like "relative Q-values". In a way, they express the "preference" for certain actions in a state, relatively independent of the value of the state, and in this sense go some way toward direct policy search methods. Several recent studies using what are considered direct policy methods similarly use a kind of values or relative values to guide the policy search toward a good policy, which itself represents independent preferences for actions (Sutton et al., 2000; Baird & Moore, 1998; Meuleau et al., 1999). It may be that such an intermediate approach between a pure value function approach and a pure direct policy search approach is a fruitful one, especially in POMDPs, because in this way one can have the benefit of a value function but avoid some of its problems, e.g. convergence problems or sensitivity to small errors in the value function.

8.1.2.8 Bootstrapping state and value estimation

Value estimation requires reasonable state estimation, while the state estimation mechanism used in this thesis depends, in turn, on estimated values. Furthermore, temporal difference-based value estimation, in general, depends on subsequent values, which the system also estimates itself. Thus, we have an overall system that uses heavy bootstrapping. One might expect or fear that this would lead to the system never getting "off the ground", or to instabilities. In practice, however, it turns out to work satisfactorily, at least for the tasks investigated in this thesis. This probably relates to the fact that in these tasks, as in many learning tasks, even though the state is only partially observable, many observations still contain considerable information about the state. For this reason, initial estimates of the state and of values can be made, at least for certain states in the environment. These initial, very rough estimates can serve as the starting points and anchors for the bootstrapping process. Subsequently, state estimation can benefit from increasingly accurate value estimation, and value estimation can benefit from increasingly accurate state estimation.

8.1.2.9 Gradient descent

Both the value functions and the utility-based state estimation (i.e. how to generalize over current observations and what to put into and use from short-term memory) were learned using gradient descent (variations of backpropagation). The successful results of this thesis illustrate the power of gradient descent as a search technique in reinforcement learning (see also Baird, 1999). A problem of gradient descent that has been pointed out by many researchers is that it can lead to local optima. This is true, but it is hard to envision search techniques that are practical, that work for complex tasks and sophisticated function approximators, and that do not suffer from the problem of local optima. In practice, it may be that locally optimal solutions are all we can hope for, and it may also be that this is good enough for many real-world problems. In the context of this thesis, an important advantage of gradient descent, compared to many alternatives, is that it does relatively focused credit assignment, because it explicitly computes gradients of performance with respect to all parameters, including the parameters responsible for storing information regarding events far back in the past.

8.1.2.10 Hierarchical control

Chapter 7 described a hierarchical control architecture. The reinforcement learning component learned to choose high-level actions which selected low-level behaviors. The high-level action decisions were made based on (the history of) high-level concepts or events, which were extracted by an unsupervised component preprocessing the raw sensory input. Hierarchical control seems to be an important principle for dealing with more complex tasks, and chapter 7 presented an illustration of this. In the context of POMDPs, in particular, hierarchical control seems to be vital. When actions should sometimes depend on past events, the considered past events should be at the proper level of abstraction. For example, a decision maker selecting which high-level route to take in navigation should not have to consider large numbers of noisy, raw sensory inputs from the past which contain little information, but only higher-level past events (encountered landmarks, such as corners and junctions) that are essentially at the same abstraction level as the higher-level actions.

8.1.2.11 Analysis of the behavior and mechanisms of agents

The conceptual contributions described above are all related to the *construction* of intelligent agents. This thesis also makes a contribution related to the question of how to *analyze* intelligent agents. The types of analyses done in this thesis include behavioral experiments, FSA extraction, and analysis of neural network activation patterns over time. They all helped in understanding how the agents solved their tasks and in understanding why the learning algorithms worked, if they worked. In much reinforcement learning and neural networks research, researchers take a black box attitude toward the learned (or evolved) systems. It is important to also analyze the learned systems on a deeper level than noting that algorithm A worked and algorithm B did not work. A deeper analysis can yield insights into the strengths and weaknesses

of different algorithms and architectures and provide clues how to improve them, and it can also suggest ways of thinking about the functioning of biological intelligence.

8.2 Interesting directions for future research

This section describes a few directions for future research that may be interesting, given the findings of this thesis. Most of them tie in fairly directly with the contributions described above.

8.2.1 Model-based versus model-free reinforcement learning

This thesis used model-free reinforcement learning techniques, but much can also be said for model-based approaches (see section 3.7.1). It would be interesting to investigate more systematically when and why model-based approaches work better than model-free approaches and vice versa, in particular when the task is a POMDP and/or the model must be learned. Presumably, factors such as the difficulty of predicting next observations play a role, but also the extent to which predicting next observations improves the utility of the estimated state signal with respect to the objectives of the agent.

Furthermore important is the extent to which the bootstrapping process for simultaneously estimating the state and the values, as described above and used throughout this thesis, will work in larger and more complex problems. It may be that in more complex problems it is better to separate the state estimation and value estimation processes, as would be the case in a model-based approach. However, it may be possible to retain some of the advantages of the model-free approach to state estimation. For instance, the learned model may be purposively incomplete and mainly focus on predictions of the world that are useful with respect to the agent's objectives and, similarly, focus mainly on trajectories in the world that the policy is likely to follow. On the other hand, perhaps the model should not be too focused on a specific objective, in order to allow reuse of the model: learning and planning for different objectives.

Another intermediate approach would be to apply model-free reinforcement learning methods (e.g. Q-learning or Advantage learning) to the model, rather than a pure model-based method such as value iteration. Advantages may include the ability to focus computational resources on likely trajectories rather than having to sweep through the whole state space, and less strict requirements as to what information the model should contain (e.g. concerning exact state transition probabilities and reward probabilities).

This thesis used LSTM recurrent neural networks in a model-free approach. It would be interesting to use LSTM networks as the substrate for predictive models of the environment, so as to exploit LSTM's desirable properties of non-vanishing gradients in that context.

8.2.2 Value functions versus direct policy search

The debate concerning value functions versus direct policy search (section 3.7.3) deserves more systematic study as well. This is especially true for the case of POMDPs,

in which the situation is least clear. We should obtain more precise insights into when either approach is better, not only because this will allow us to select approaches more appropriately, but also because it may lead to better approaches. In the conceptual contributions section above it was suggested that an intermediate approach between pure value functions and pure direct policy search might be good, and this suggestion could be investigated further.

An interesting direct policy search approach that received some attention some time ago, but that since then has apparently not been investigated extensively, is the backpropagation through a model approach (Werbos, 1990; Jordan & Rumelhart, 1992; Kawato, 1990; Nguyen & Widrow, 1990; Schmidhuber, 1991c, see sections 3.5.1.3 and 3.6.2.3). This approach does have certain attractive properties, however, justifying further research. It allows for relatively directed credit assignment to actions, because explicit gradients of the error with respect to action vectors are computed. For the same reason, this approach can be used for problems with many or high-dimensional actions, such as robots with many degrees of freedom; this is a problem area that has been relatively neglected in the reinforcement learning community.

The backpropagation through a model approach can also be combined with value functions (see section 3.5.1.3). This approach can be called backpropagation through a critic, and it has rarely been tried. It might work well, because much of the burden of temporal credit assignment is taken over by the value function, and it still has the attractive property of directed credit assignment to actions.

Backpropagation through a model or critic requires a differentiable model and controller, and it is usually done with recurrent neural networks. It may be particularly interesting to try backpropagation through a model or critic approaches with LSTM networks, both for the model or critic part and for the controller part, to see if LSTM's non-vanishing gradient computation is fruitful here as well.

8.2.3 Hierarchical reinforcement learning

The hierarchical control architecture of chapter 7 was still fairly unsophisticated. The high-level sensory concepts and high-level actions were learned, but the low-level behaviors were handcoded. The low-level behaviors were, partly because they were handcoded, very simple (wall following etc.). Furthermore, the event extraction mechanism yielding the high-level concepts did not receive any feedback about the usefulness of its concepts; the parameters of the event extraction mechanism were handcoded such that the extracted concepts were useful. Finally, there were only two levels in the hierarchy.

It would be interesting to get rid of these limitations. Ideally, the hierarchical reinforcement learning system should learn at all levels of the control hierarchy. Thus, it should learn low-level policies, and it should learn high-level policies coordinating the low-level policies. It should also determine the control hierarchy itself: the number of levels and the conditions which prompt decisions in the higher levels of the control hierarchy. Related to the latter problem is the problem of determining the abstraction level of the "conditions" considered by higher levels. In other words, this leads to the problem of concept formation: how should we abstract over raw sensory information to yield higher-level concepts that are useful inputs to higher-level policies? How should

we abstract over those concepts to yield still higher-level concepts? In general, we want to filter out a lot of information about the complete environmental state for the high-level decision maker, in order to reduce its search space. For this reason, at the higher level of decision making the decision problem usually becomes a POMDP even if the overall problem is an MDP. Hierarchical methods should take this into account. Furthermore, how can we select, preferably automatically, what information is useful for the higher-level decision maker? In some cases, apparently low-level sensory "details" may be very useful for solving a high-level task, e.g. when a robot using vision can disambiguate between two office hallways by looking at a small sign indicating the floor of this hallway. These are all questions that warrant further research.

In general, it seems that hierarchical control is one of the keys to more advanced artificial intelligence, which can be used in more complex and realistic domains. The brains of animals and humans actually provide good inspiration for this: there are numerous, classical studies that suggest that brains are built and operate in a hierarchical way (e.g. Jackson, 1870). And, as in other cases, it may be that making progress on hierarchical artificial intelligence techniques gives additional insights into the functioning of biological intelligence.

8.2.4 Practical applications

The future research directions suggested above all concern *theoretical* issues in reinforcement learning: the use of models, value functions versus direct policy search, and hierarchy. It is also important to attempt more *practical applications* of the methods discussed in this thesis, and of reinforcement learning methods in general. It is important because not enough convincing applications exist, and because much can be learned from such real-world "tests". Application areas include robots, scheduling problems, games, industrial controllers, and internet agents.

When thinking about practical applications, we immediately run into the problem of the typically long training times associated with reinforcement learning algorithms. I do not believe there are magical reinforcement learning algorithms "out there", waiting to be discovered, which will, by themselves, lead to very fast and effective learning in a wide variety of domains. This relates directly to bias arguments (Mitchell, 1980; Geman et al., 1992) and no free lunch arguments (Wolpert & Macready, 1997). In short, fast and effective learning requires significant bias, or a priori knowledge, for the specific task at hand, and bias towards some tasks implies that the method will work relatively badly for other tasks.

This is not necessarily a problem. First of all, we may be able and willing to introduce bias for the specific classes of tasks we would like to apply reinforcement learning to. Important research questions include the question how to add bias or a priori knowledge to reinforcement learning methods, how to possibly use, learn, and evolve useful bias (Minsky, 1961; Peshkin & de Jong, 2002; Schmidhuber, 2002) for classes of tasks, and even the fundamental question which types of biases are good for which types of tasks.

Secondly, it is useful at this point to recall the distinction between online reinforcement learning and offline reinforcement learning (section 3.2.7). Let us consider a robot application. In summary, in online reinforcement learning the goal is to build

a reinforcement learning robot, while in offline reinforcement learning the goal is to build a robot that can perform some task, and to develop the robot controller using reinforcement learning. Both online and offline reinforcement learning for real-world applications are interesting future research directions in their own right, but one should keep in mind that they have different constraints. Online reinforcement learning will only work when we build in a large amount of bias, such that the number of learning experiences can be very low. Offline reinforcement learning may use more learning experiences and for this reason does not require as much bias; but it will not, by itself, lead to a robot that can adapt itself online.

As noted in section 3.2.7, an interesting approach may be to combine offline and online reinforcement learning. Offline learning is used to arrive at a reasonable controller which works well in simulation but not perfectly in the real robot. This controller is then used as the starting point for online reinforcement learning, which can use basically the same learning algorithm as was used during offline learning. In this way, offline reinforcement learning can provide the necessary bias to make online reinforcement learning feasible.

In any case, in practical applications many of the problems discussed in this thesis are relevant: the problems of large state spaces, high-dimensional sensory inputs, stochasticity, partial observability, long time lags, etc. Therefore, practical applications provide good test beds for the proposed solutions, in particular the use of (recurrent) neural networks for state estimation, the use of LSTM and relative value functions for dealing with long time lags, and the use of hierarchical control for learning and planning in complex tasks.

8.3 Final remarks

This thesis started with an argument for using adaptive behavior research as an approach to studying both artificial and biological intelligence. Looking back at this thesis, to what extent is this argument supported by this thesis' results?

The contributions of this thesis are summarized above, and I think they may be interesting both from an artificial intelligence point of view and from a biological intelligence point of view. For example, the ideas of value function approximation, hierarchical control, the trade-off between perception and internal state, and utility-based state estimation provide powerful methods for artificial intelligence and at the same time provide suggestions as to how biological systems solve the problems of intelligence.

It is hard to see how these types of ideas could be developed in other contexts, for example by studies that identify specific brain region activation patterns, studies that collect and model simple responses of humans in very contrained experimental tasks, studies that focus on abstract logic, or studies that are concerned with static pattern recognition problems. Each of these alternative approaches has its merits (some a bit more than others, perhaps), but adaptive behavior research has its own unique perspective.

One of adaptive behavior's most important points is its focus on the fact that different elements of intelligence, i.e. perception, memory, and action, are intimately

connected, and that they are intimately connected to the environment of the agent. This focus can be seen to return in this thesis' contributions. Adaptive behavior research provides formal suggestions how the different elements of intelligence work, and in particular, how they work *together*. Because adaptive behavior research proposes formal—i.e. mathematical and computational—models and robots, we can check if they can indeed, and to what extent, perform the tasks they are claimed to perform. Thus, in a way, adaptive behavior does exploratory research into "holistic" but formal theories of intelligence (without necessarily converging to just one theory, see section 2.3.10). I believe that such a holistic view is essential in understanding individual subsystems (such as particular brain regions) and individual processes (such as static pattern recognition), exactly because their context is taken into account; and it is essential in understanding how the various subsystems and processes interact. For these reasons, I believe that adaptive behavior research has a crucial role to play in the continuing study of artificial and biological intelligence.

Appendix A

Input-output FSA extraction and minimization

This appendix describes algorithms for extraction of input-output FSAs from recurrent neural networks and the subsequent minimization of the extracted input-output FSAs. We assume that the internal state at time t of the recurrent neural network is an n-dimensional continuous vector $v(t)$ where each element lies in a fixed range, say $[0, 1]$. This means that the internal state space is the n-dimensional hypercube $X = [0, 1]^n$. We also assume that the inputs and outputs are discrete and countable. In this thesis, the recurrent neural network actually outputs continuous values, but for FSA extraction the output is considered to be the action chosen by greedy action selection. The input (observation) at time t is $o(t)$, the output (action) is $a(t)$. The last timestep is T.

Firstly, the continuous internal state space X is "discretized" at resolution r by dividing it into r^n hypercubes of equal size. See figure A.1 for an illustration in 2 dimensions, with $r = 5$. The pseudocode in Algorithm A.1 describes the algorithm used to extract the FSA.

Figure A.1 shows the idea of the algorithm schematically. Each hypercube that contains at least one point of the internal state trajectory becomes an FSA state. In general, this will be true for only a fraction of the total number of hypercubes. The FSA's edges are determined by recording which hypercubes are visited in succession during normal operation of the recurrent neural network. Going from the current internal state to the next, the associated input and output (action) are recorded and stored with the edge. This is straightforward here because both input and output are discrete and countable.

Note that this algorithm extracts a Mealy machine, i.e. an input-output FSA where the outputs are stored with the edges (see section 3.3.2). In the Elman recurrent neural networks considered in chapters 4 and 5, this corresponds to considering the context unit activations as the internal state, and considering the output function a function of this internal state and the current input of the network. Alternatively, one could extract a Moore machine, i.e. an input-output FSA where the outputs are stored with

```
j=0
for all t do
   determine the hypercube h(t) in which v(t) falls
   if h(t) is not labeled then
      ++j
      label h(t) as FSA state j
   end if
end for

for all t < T do
   determine h(t)'s FSA state j
   in table entry for FSA state j: store o(t), a(t), next FSA state i
end for

for all j do
   estimate probability of each FSA edge by computing, for each input, the proportion of each output-next state pair
end for
```

Algorithm A.1: Input-output FSA extraction

the states, by considering the hidden layer activation vector of the Elman network as the internal state of the system. In that case, the output function of the system is a function only of this internal state; after all, the output of an Elman network can be considered to be a function only of the current hidden unit activations.

Even though we consider recurrent neural networks with deterministic action selection, the extracted FSA can be stochastic. This is caused by the discretization method, which groups internal states fairly arbitrarily. For the same reason, the procedure will often yield an FSA that has many states that do not differ "functionally", i.e. different states which produce the same output for the same input and lead to the same subsequent states. For standard input FSAs (or output FSAs), Hopcroft and Ullman (1979) describe a *minimization* procedure, so-called Hopcroft minimization, which identifies and merges functionally equivalent states. Here a minimization procedure is described for the input-output FSAs extracted from the recurrent neural networks.

The minimization procedure described here is somewhat different from the one described by Hopcroft and Ullman (1979) for three reasons. First of all, the extracted FSAs are input-output FSAs rather than input (or output) FSAs. Secondly, in contrast to Hopcroft and Ullman (1979), here the FSAs have many different inputs. Thirdly, many inputs occur only when the FSA is in certain states and not in others.

There are basically two options for minimization of the extracted FSAs, given these differences. One option would be to restrict the consideration to merge states to pairs of states that have only common inputs eminating from the states. However, an important goal in this thesis is to distinguish systems that use internal state-based policies from systems that use perception-based policies, i.e. systems that could be replaced by, for example, a feedforward neural network; and this is not possible if

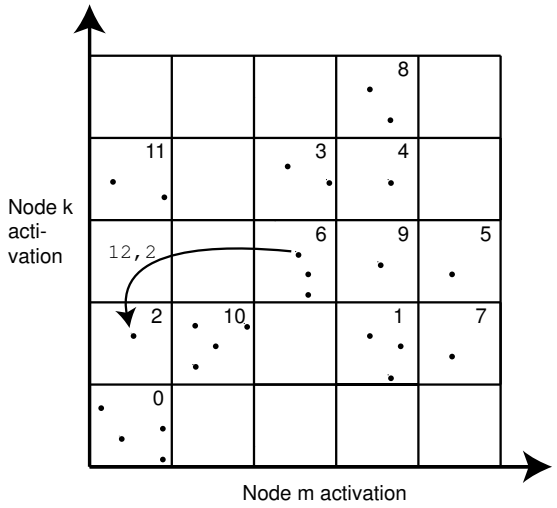

Figure A.1: Discretization of a two-dimensional internal state space. Each dimension is partitioned into 5 equal regions. The FSA states that are extracted have numbers in their corresponding grid cells. One state transition is indicated: the FSA is in state 6, it receives input 12, emits output 2, and goes to FSA state 2.

states can only be merged if they share all inputs. Thus, we want to also be able to merges states when they occur with different, non-overlapping inputs.

The criterion to consider states as different is: two states are different if for at least one input, the output is different or the next state is different. In contrast to standard Hopcroft minimization (Hopcroft & Ullman, 1979), the order in which states are merged matters. This is so because it may be the case that FSA state 1 can be merged with 2, and 1 can also be merged with 3, but 2 cannot be merged with 3. This can occur when 1 and 2 do not share inputs, and neither do 1 and 3, but 2 and 3 do. It is unclear at this point how to determine the order of merging that is optimal, i.e. how the smallest possible FSA can be guaranteed to be found.

In the algorithm used here, there is a first pass in which only pairs of states are considered for merging that share all inputs. The resulting FSA, which is already minimized to some extent, then enters the second phase. In the second phase, the FSA is incrementally minimized by considering also pairs of states where at least one input is not shared. To avoid the problem described above, two states that can be merged are "flagged", and in the current iteration of the algorithm they are no longer considered for merging with other states. In the next iteration, these two states are actually merged and become a single state, and now this new state can once again be considered for merging with other states. This process continues until no new merges can be found.

Like the standard Hopcroft minimization procedure, sometimes the only difference between two FSA states is that for the same input (and output) they lead to different

subsequent states. This presents a problem, because during the minimization process we do not yet know which states are functionally different. The same method is employed here as in standard Hopcroft minimization. A table is created with entries for all pairs of states. When two states are found to be different, the entry is "marked". When the distinction between two states depends, as explained above, on whether two other, subsequent states are different, the indices of the first two states are placed on a list associated with the entry of the second pair of states. Once a pair of states is found to be different and its corresponding entry is marked, also the entries corresponding to pairs of states on the list of this entry are marked. The complete minimization algorithm is described as pseudocode in Algorithm A.2.

By increasing the resolution of discretization of FSA extraction from the recurrent neural network, an FSA can be extracted that more and more precisely and eventually perfectly models the behavior of the network (Blair & Pollack, 1997; Casey, 1996). Increasing the resolution of the discretization process does not necessarily lead to more FSA states, thanks to the minimization procedure which gets rid of superfluous states. In this thesis, the discretization resolution was increased until the size of the minimized FSA did not vary significantly anymore and the FSA perfectly modeled the network's behavior.

```
for all pairs of states (i,j) in the FSA do
  if an input is different or for an input an output is different then
    mark (i,j) and recursively mark all entries on the list of (i,j)
  else if for the same input and output the next states (k,m) are different then
    if (k,m) are marked then
      mark (i,j) and recursively mark all entries on the list of (i,j)
    else
      put (i,j) on the list of (k,m)
    end if
  end if
end for
merge all unmarked state pairs and create the corresponding FSA

repeat
  for all pairs of states (i,j) in the FSA do
    p(i)=−1 and p(j)=−1
  end for
  for all pairs of randomly selected states (i,j) in the FSA where p(i)==−1 and
  p(j)==−1 do
    if for an input an output is different then
      mark (i,j) and recursively mark all entries on the list of (i,j)
    else if for the same input and output the next states (k,m) are different then
      if (k,m) are marked then
        mark (i,j) and recursively mark all entries on the list of (i,j)
      else
        put (i,j) on the list of (k,m)
        p(i)=j and p(j)=i
      end if
    else
      p(i)=j and p(j)=i
    end if
  end for
  merge unmarked state pairs where p(i)==j and p(j)==i and create the corre-
  sponding FSA
until the FSA is not changing any more
```

Algorithm A.2: Input-output FSA minimization

Appendix B

Equivalence of the forward and backward view of Advantage(λ) learning

This appendix shows the equivalence of the two views of Advantage(λ) learning, the forward and the backward view, with function approximators. More precisely, it shows that offline Advantage learning with the same eligibility traces as Q-learning, and on the basis of the one-step return (the backward view), leads to the same weight updates as offline Advantage learning without eligibility traces on the basis of the λ-return (the forward view). This is important because it is not obvious that Advantage learning can be combined with eligibility traces in the same way as Q-learning and lead to the same desired result, namely back-ups based on the λ-return. It is not obvious because the corresponding Bellman equations differ for the two algorithms, and the known results for Q(λ) and TD(λ) depend on the corresponding Bellman equations.

The following proof makes use of ideas from Watkins (1989), Sutton (1988), Sutton (1989), and Sutton and Barto (1998). It is based on the proof presented in Sutton (1988) and Sutton and Barto (1998) regarding the equivalence of the forward and backward view of TD(λ) prediction using a tabular representation. Compared to their proof, the proof presented here is not only adapted for Advantage learning's different Bellman equation, it is also slightly more general, because it allows for λ to be set to 0 occasionally during the episode (as is necessary during exploration in off-policy control methods like Q-learning and Advantage learning), and it allows for function approximators, of which the tabular representation is a special case.

Let $\Delta w_{i,t}^{\lambda}$ denote the weight update at time t according to the λ-return algorithm, and let $\Delta w_{i,t}^{TD}$ denote the weight update according to the algorithm with eligibility traces. Let the episode start at time $t = 0$ and end at $t = T$. What we need to establish, then, is that the sum of all weight updates over an episode is always the

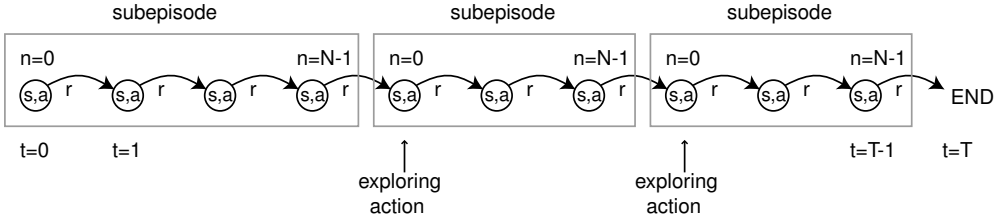

Figure B.1: Schematic representation of how an episode is made up of subepisodes. The moments at which exploring actions are taken mark the beginning of a new subepisode.

same for both algorithms:

$$\sum_{t=0}^{T-1} \Delta w_{i,t}^{\lambda} = \sum_{t=0}^{T-1} \Delta w_{i,t}^{TD}. \tag{B.1}$$

The first step is to define a *subepisode* as a substring of the entire string of state-action pairs making up the episode, such that the first element in the substring is either the first state-action pair of the episode ($t = 0$) or a state-action pair in which the action is exploratory, and the last element is either the last state-action pair before the end ($t = T - 1$) or the last state-action pair before the next exploratory action. Let $n = 0$ denote the first element of the subepisode and $n = N - 1$ the last element. Figure B.1 shows how an episode is composed of subepisodes. The entire episode is simply the concatenation of all subepisodes. Therefore, if we want to show equivalence of the weight updates over an entire episode, it is sufficient to show equivalence over each subepisode. Thus, we must establish that

$$\sum_{n=0}^{N-1} \Delta w_{i,n}^{\lambda} = \sum_{n=0}^{N-1} \Delta w_{i,n}^{TD}. \tag{B.2}$$

Let us start with the lefthand side of equation B.2:

$$\sum_{n=0}^{N-1} \Delta w_{i,n}^{\lambda} = \sum_{n=0}^{N-1} \alpha E_n^{\lambda} \frac{\partial A(s_n, a_n)}{\partial w_i} \tag{B.3}$$

where

$$E_n^{\lambda} = V(s_n) + \frac{R_n^{\lambda} - V(s_n)}{\kappa} - A(s_n, a_n). \tag{B.4}$$

R_n^{λ} is the λ-return, and in off-policy control methods, such as Q-learning and Advantage-learning, it is defined as (Watkins, 1989):

$$R_n^{\lambda} = (1 - \Lambda_{n+1}) R_n^{(1)} + \Lambda_{n+1}(1 - \Lambda_{n+2}) R_n^{(2)} + \Lambda_{n+1}\Lambda_{n+2}(1 - \Lambda_{n+3}) R_n^{(3)} + \dots \tag{B.5}$$

where $R_n^{(p)}$ is the p-step return, and

$$\Lambda_n = \begin{cases} \lambda & a_n = \arg\max_a A(s_n, a) \\ 0 & \text{otherwise.} \end{cases} \tag{B.6}$$

Thus, the λ-return is truncated at the point where an exploring action is chosen. This corresponds to the notion that rewards after that point no longer reflect the value of the currently estimated best policy. From the definition of a subepisode it follows that $\Lambda_n = \lambda$ for all $0 < n < N$ and $\Lambda_N = 0$, so equation B.5 can be rewritten as

$$R_n^\lambda = (1-\lambda)R_n^{(1)} + (1-\lambda)\lambda R_n^{(2)} + (1-\lambda)\lambda^2 R_n^{(3)} + ... + \lambda^{N-1-n} R_n^{(N)}. \tag{B.7}$$

This means that $R_n^\lambda - V(s_n)$ from equation B.4 can be written as

$$\begin{aligned} R_n^\lambda - V(s_n) = & -V(s_n) \\ & + (1-\lambda)\lambda^0(r_{n+1} + \gamma V(s_{n+1})) \\ & + (1-\lambda)\lambda^1(r_{n+1} + \gamma r_{n+2} + \gamma^2 V(s_{n+2})) \\ & + (1-\lambda)\lambda^2(r_{n+1} + \gamma r_{n+2} + \gamma^2 r_{n+3} + \gamma^3 V(s_{n+3})) \\ & + ... \\ & + \lambda^{N-1-n}(r_{n+1} + \gamma r_{n+2} + \gamma^2 r_{n+3} + ... + \gamma^{N-1-n} r_N + \gamma^{N-n} V(s_N)). \end{aligned} \tag{B.8}$$

Now, R_n^λ was designed as a weighted sum of p-step returns. Thus, $(1-\lambda)\lambda^0 + (1-\lambda)\lambda^1 + ... + \lambda^{N-1-n}$ sums to 1. This means that we can pull out the first column inside the brackets and get an unweighted term for that sum. We can do a similar thing for the second column, etc. In all, we can rewrite equation B.8 as

$$\begin{aligned} R_n^\lambda - V(s_n) = & -V(s_n) \\ & + (\gamma\lambda)^0(r_{n+1} + \gamma V(s_{n+1}) - \gamma\lambda V(s_{n+1})) \\ & + (\gamma\lambda)^1(r_{n+2} + \gamma V(s_{n+2}) - \gamma\lambda V(s_{n+2})) \\ & + (\gamma\lambda)^2(r_{n+3} + \gamma V(s_{n+3}) - \gamma\lambda V(s_{n+3})) \\ & + ... \\ & + (\gamma\lambda)^{N-1-n}(r_N + \gamma V(s_N)) \\ = & \quad (\gamma\lambda)^0(r_{n+1} + \gamma V(s_{n+1}) - V(s_n)) \\ & + (\gamma\lambda)^1(r_{n+2} + \gamma V(s_{n+2}) - V(s_{n+1})) \\ & + (\gamma\lambda)^2(r_{n+3} + \gamma V(s_{n+3}) - V(s_{n+2})) \\ & + ... \\ & + (\gamma\lambda)^{N-1-n}(r_N + \gamma V(s_N) - V(s_{N-1})) \\ = & \sum_{k=n}^{N-1} (\gamma\lambda)^{k-n}(r_{k+1} + \gamma V(s_{k+1}) - V(s_k)). \end{aligned} \tag{B.9}$$

For convenience, let us define

$$\delta_k = r_{k+1} + \gamma V(s_{k+1}) - V(s_k). \tag{B.10}$$

Now we can rewrite equation B.4 as

$$E_n^\lambda = V(s_n) + \frac{\sum_{k=n}^{N-1}(\gamma\lambda)^{k-n}\delta_k}{\kappa} - A(s_n, a_n) \tag{B.11}$$

and equation B.3 as

$$\sum_{n=0}^{N-1} \Delta w_{i,n}^{\lambda} = \sum_{n=0}^{N-1} \alpha \frac{\partial A(s_n, a_n)}{\partial w_i} (V(s_n) + \frac{\sum_{k=n}^{N-1} (\gamma\lambda)^{k-n} \delta_k}{\kappa} - A(s_n, a_n)). \tag{B.12}$$

This is what the backward, mechanistic view of Advantage(λ) learning must realize using eligibility traces. So let us turn now to the righthand side of equation B.2, which corresponds to weight updates according to the backward view:

$$\sum_{n=0}^{N-1} \Delta w_{i,n}^{TD} = \sum_{n=0}^{N-1} \alpha E_n^{TD} e_{i,n} \tag{B.13}$$

where

$$\begin{aligned} E_n^{TD} &= V(s_n) + \frac{r_{n+1} + \gamma V(s_{n+1}) - V(s_n)}{\kappa} - A(s_n, a_n) \\ &= V(s_n) + \frac{\delta_n}{\kappa} - A(s_n, a_n) \end{aligned} \tag{B.14}$$

and the eligibility traces for function approximators $e_{i,n}$ (Sutton, 1989) are defined as

$$e_{i,n} = \gamma \Lambda_n e_{i,n-1} + \frac{\partial A(s_n, a_n)}{\partial w_i}. \tag{B.15}$$

For reasons that will become apparent shortly, we decompose the sum in equation B.13 into two parts:

$$\sum_{n=0}^{N-1} \Delta w_{i,n}^{TD} = \sum_{n=0}^{N-1} \alpha \frac{\delta_n}{\kappa} e_{i,n} + \sum_{n=0}^{N-1} \alpha (V(s_n) - A(s_n, a_n)) e_{i,n}. \tag{B.16}$$

Let's consider the second part of this sum. Note that $e_{i,0} = \frac{\partial A(s_0, a_0)}{\partial w_i}$. Furthermore, note that for $0 < n < N$, $A(s_n, a_n) = V(s_n)$, because these are all exploiting actions. Thus, equation B.16 can be rewritten as

$$\sum_{n=0}^{N-1} \Delta w_{i,n}^{TD} = \sum_{n=0}^{N-1} \alpha \frac{\delta_n}{\kappa} e_{i,n} + \sum_{n=0}^{N-1} \alpha (V(s_n) - A(s_n, a_n)) \frac{\partial A(s_n, a_n)}{\partial w_i}. \tag{B.17}$$

We know that within a subepisode, $\Lambda_n = \lambda$ for all $0 < n < N$. The recursive definition of equation B.15 can then be rewritten in the following non-recursive form (Watkins, 1989; Sutton & Barto, 1998):

$$e_{i,n} = \sum_{k=0}^{n} (\gamma\lambda)^{n-k} \frac{\partial A(s_k, a_k)}{\partial w_i}. \tag{B.18}$$

Thus, equation B.17 becomes

$$\begin{aligned}
\sum_{n=0}^{N-1} \Delta w_{i,n}^{TD} &= \sum_{n=0}^{N-1} \alpha \frac{\delta_n}{\kappa} \sum_{k=0}^{n} (\gamma\lambda)^{n-k} \frac{\partial A(s_k, a_k)}{\partial w_i} + \sum_{n=0}^{N-1} \alpha (V(s_n) - A(s_n, a_n)) \frac{\partial A(s_n, a_n)}{\partial w_i} \\
&= \sum_{k=0}^{N-1} \alpha \sum_{n=0}^{k} (\gamma\lambda)^{k-n} \frac{\partial A(s_n, a_n)}{\partial w_i} \frac{\delta_k}{\kappa} + \sum_{n=0}^{N-1} \alpha (V(s_n) - A(s_n, a_n)) \frac{\partial A(s_n, a_n)}{\partial w_i} \\
&= \sum_{n=0}^{N-1} \alpha \sum_{k=n}^{N-1} (\gamma\lambda)^{k-n} \frac{\partial A(s_n, a_n)}{\partial w_i} \frac{\delta_k}{\kappa} + \sum_{n=0}^{N-1} \alpha (V(s_n) - A(s_n, a_n)) \frac{\partial A(s_n, a_n)}{\partial w_i} \\
&= \sum_{n=0}^{N-1} \alpha \frac{\partial A(s_n, a_n)}{\partial w_i} \sum_{k=n}^{N-1} (\gamma\lambda)^{k-n} \frac{\delta_k}{\kappa} + \sum_{n=0}^{N-1} \alpha (V(s_n) - A(s_n, a_n)) \frac{\partial A(s_n, a_n)}{\partial w_i} \\
&= \sum_{n=0}^{N-1} \alpha \frac{\partial A(s_n, a_n)}{\partial w_i} (V(s_n) + \frac{\sum_{k=n}^{N-1} (\gamma\lambda)^{k-n} \delta_k}{\kappa} - A(s_n, a_n))
\end{aligned} \tag{B.19}$$

which is equal to equation B.12. This proves the equivalence of the forward and backward view.

This result only holds for offline updating, i.e. updating after each episode (or, in our case, after each subepisode), otherwise $A(s_n, a_n)$, $V(s_n)$, and $\frac{\partial A(s_n, a_n)}{\partial w_i}$ are not necessarily exactly equal in both cases. However, since weight changes during an episode using online updating will in general be small, we can expect a close approximation in the online case. In summary, the result says that if we simply use the same type of eligibility traces with Advantage learning as we do with Q-learning, we obtain a similar end result as with Q-learning, namely gradient descent in weight space based on the λ-return. In fact, just as Q-learning without eligibility traces is a special case of Advantage learning without eligibility traces, Q(λ)-learning is a special case of Advantage(λ) learning, when $\kappa = 1$.

References

Abul, A., Polat, F., & Alhaji, R. (2000). Multiagent reinforcement learning using function approximation. *IEEE Transactions on Systems, Man, and Cybernetics. Part C: Applications and reviews*, *30*, 485–497.

Acharyya, S. (2000). *Learning radial basis function based soccer strategies using ideal opponent model.* MSc thesis, Center for Robotics and Mechatronics, Indian Institute of Technology, Kanpur.

Anderson, C. (2000). *Approximating a policy can be easier than approximating a value function* (Technical report No. CS-00-101). Fort Collins, CO, 80523: Department of Computer Science, Colorado State University.

Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. In *Proceedings of the fourth international workshop on machine learning* (pp. 103–114). San Mateo, CA: Morgan Kaufmann.

Anderson, C. W. (1993). Q-learning with hidden-unit restarting. In S. J. Hanson, J. D. Cowan, & C. L. Giles (Eds.), *Advances in neural information processing systems* (Vol. 5, pp. 81–88). Morgan Kaufmann, San Mateo, CA.

Arbib, M. A. (1989). *The metaphorical brain 2: Neural networks and beyond.* New York: John Wiley and Sons.

Arbib, M. A. (1995). Road maps: Learning in artificial neural networks. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks.* Cambridge, MA: MIT Press.

Arleo, A., Smeraldi, F., Hug, S., & Gerstner, W. (2001). Place cells and spatial navigation based on 2d visual feature extraction, path integration, and reinforcement learning. In *Nips 13.*

Ashby, W. R. (1960). *Design for a brain. (Second, revised edition).* London: Chapman and Hall.

Ashcraft, M. H. (1998). *Fundamentals of cognition.* New York: Addison-Wesley.

Baddeley, A. (1990). *Human memory.* London: Lawrence Erlbaum Associates.

Baird, L. C. (1994). Reinforcement learning in continuous time: Advantage updating. In *Proceedings of the international conference on neural networks.* Orlando, FL.

Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the twelfth international conference on machine learning* (pp. 30–37). San Francisco: Morgan Kaufmann.

Baird, L. C. (1999). *Reinforcement learning through gradient descent.* PhD thesis, Carnegie Mellon University, Pittsburgh.

Baird, L. C., & Klopf, A. H. (1993). *Reinforcement learning with high-dimensional, continuous actions* (Technical report No. WL-TR-93-1147). Wright-Patterson Air Force Base Ohio: Wright Laboratory.

Baird, L. C., & Moore, A. (1998). Gradient descent for general reinforcement learning. In *Advances in neural information processing systems* (Vol. 11).

Barto, A. G. (1990). Connectionist learning for control. In W. T. Miller, R. S. Sutton, & P. J. Werbos (Eds.), *Neural networks for control* (pp. 5–58). Cambridge, Mass: M.I.T. Press.

Barto, A. G. (1995). Reinforcement learning in motor control. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks.* Cambridge, MA: MIT Press.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, *13*, 835–846.

Baum, E. B. (1999). Toward a model of intelligence as an economy of agents. *Machine Learning*, *35*, 155.

Beer, R. D. (1990). *Intelligence as adaptive behavior: An experiment in computational neuroethology.* San Diego, CA: Academic Press.

Beer, R. D. (1995). Computational and dynamical languages for autonomous agents. In R. Port & T. van Gelder (Eds.), *Mind as motion.* MIT Press.

Beer, R. D., Chiel, H., Quinn, K., Espenschied, S., & Larsson, P. (1992). A distributed neural network architecture for hexapod robot locomotion. *Neural Computation, 4 (3)*, 56–365.

Beer, R. D., & Gallagher, J. C. (1992). Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior, 1*, 91–122.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. In *Advances in neural information processing systems 6* (pp. 75–82). San Mateo, CA: Morgan Kaufmann.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming.* Belmont, MA: Athena Scientific.

Blair, A. D., & Pollack, J. B. (1997). Analysis of dynamical recognizers. *Neural Computation*, *9*, 1127–1142.

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, & T. K. Leen (Eds.), *Advances in neural information processing systems 7* (pp. 369–376). Cambridge, MA: The MIT Press.

Braitenberg, V. (1984). *Vehicles: Experiments in synthetic psychology.* Cambridge, MA: MIT Press.

Brinkers, M., & den Dulk, P. (1999). The evolution of non-reciprocal altruism. In D. Floreano, J.-D. Nicoud, & F. Mondada (Eds.), *Advances in artificial life, ecal'99* (pp. 499–503). Berlin: Springer.

Brooks, R. A. (1989). A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation, 1*, 253–262.

Brooks, R. A. (1991a). Intelligence without reason. In *Proceedings of the 12th internal joint conference on artificial intelligence.* San Mateo, CA: Morgan Kauffman.

Brooks, R. A. (1991b). Intelligence without representation. *Artificial Intelligence*, *47*, 139–159.

Casey, M. (1996). The dynamics of discrete-time computation, with application to recurrent neural networks and finite state automata. *Neural Computation*, *8*, 1135–1178.

Cassandra, A., Littman, M. L., & Zhang, N. L. (1997). Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In D. Geiger & P. P. Shenoy (Eds.), *Proceedings of the thirteenth annual conference on uncertainty in artificial intelligence (uai–97)* (pp. 54–61). San Francisco, CA: Morgan Kaufmann Publishers.

Chalmers, D. J. (1990). Syntactic transformations on distributed representations. *Connection Science, 2*, 53–62.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In J. Mylopoulos & R. Reiter. (Eds.), *Proceedings of the twelfth international joint conference on artificial intelligence (ijcai-91)* (pp. 726–731). San Mateo, Ca.: Morgan Kaufmann.

Cheng, H.-T. (1988). *Algorithms for partially observable Markov decision processes.* Unpublished doctoral dissertation, University of British Columbia, British Columbia, Canada.

Chomsky, N. (1959). A review of B.F. Skinner's Verbal Behavior. *Language*, *35 (1)*, 26–58.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the tenth national conference on artificial intelligence.* San Jose, CA: AAAI Press.

Churchland, P. S. (1986). *Neurophilosophy: Toward a unified science of mind-brain.* Cambridge, MA: MIT Press.

Clark, A. (1997). *Being there: Putting mind, body, and world together again.* Cambridge, MA: MIT Press.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, *1*, 372–381.

Cliff, D. (1991). Computational neuroethology: A provisional manifesto. In J.-A. Meyer & S. Wilson (Eds.), *Proceedings of the first international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Cliff, D., & Ross, S. (1994). Adding temporary memory to ZCS. *Adaptive Behavior*, *3:2*, 101–150.

Crick, F. (1988). *What mad pursuit: A personal view of scientific discovery.* New York: Basic Books.

Crites, R. H., & Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In D. S. T. amd M. C. Mozer & M. E. Hasselmo (Eds.), *Advances in neural information processing systems: Proceedings of the 1995 conference* (pp. 1017–1023). Cambridge, MA: MIT Press.

Crutchfield, J. P. (1994). The calculi of emergence: Computation, dynamics, and intuition. *Physica D, Special issue on the Proc. of the Oji International Seminar Complex Systems — from Complex Dynamics to Artificial Reality.*

Dayan, P., & Hinton, G. E. (1993). Feudal reinforcement learning. In S. J. Hanson, J. D. Cowan, & C. L. Giles (Eds.), *Advances in neural information processing systems 5: Proceedings of the 1992 conference.* San Mateo, Ca.: Morgan Kaufmann Publishers.

de Jong, E. D. (1997). An accumulative exploration method for reinforcement learning. In S. Sen (Ed.), *Proceedings of the aaai'97 workshop on multiagent learning, available as aaai technical report ws-97-03* (p. 19-24). Menlo Park, California: AAAI Press.

de Jong, E. D. (1999). Autonomous concept formation. In T. Dean (Ed.), *Proceedings of the sixteenth international joint conference on artificial intelligence ijcai'99* (pp. 344–349). San Francisco, CA: Morgan Kaufmann.

Dellaert, F., Fox, D., Burgard, W., & Thrun, S. (1999). Monte carlo localization for mobile robots. In *IEEE international conference on robotics and automation (ICRA99).*

Dennett, D. C. (1991). *Consciousness explained.* Boston: Little, Brown.

Dennett, D. C. (1994). Cognitive science as reverse engineering: Several meanings of "top-down" and "bottom-up". In D. Prawitz, B. Skyrms, & D. Westerstahl (Eds.), *Proceedings of the 9th international congress of logic, methodology, and philosophy of science.*

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Donders, F. C. (1862). Die schnelligkeit psychischer processe. [the speed of psychological processes]. *Arch. Anat. Physiol.*, 657–681.

Dorigo, M., & Bersini, H. (1994). A comparison of Q-learning and classifier systems. In *Proceedings of From Animals to Animats, third international conference on simulation of adaptive behavior.*

Doya, K. (1992). Bifurcations in the learning of recurrent neural networks. In *Proc. of 1992 IEEE int. symposium on circuits and systems* (p. 2777-2780).

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation*, *12 (1)*, 219–245.

Eck, D., & Schmidhuber, J. (2002). Learning the long-term structure of the blues. In J. Dorronsoro (Ed.), *Artificial Neural Networks – ICANN 2002* (pp. 284–289). Berlin: Springer.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*, 179–211.

Fernández, F., & Borrajo, D. (1999). VQQL. Applying vector quantization to reinforcement learning. In *Ijcai'99 workshop on robocup.*

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, *4*(1), 1–58.

Gers, F., Schmidhuber, J., & Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, *12 (10)*, 2451–2471.

Gers, F. A., Eck, D., & Schmidhuber, J. (2001). Applying LSTM to time series predictable through time-window approaches. In *Proc. ICANN 2001, Int. Conf. on Artificial Neural Networks.* Vienna, Austria: IEE, London.

Gers, F. A., & Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proc. ijcnn'2000, int. joint conf. on neural networks.*

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., & Lee, Y. C. (1992). Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, *4*, 393–405.

Glaser, M. O., & Düngelhoff, F.-J. (1984). The time-course of picture-word interference. *Journal of Experimental Psychology: Human Perception and Performance, 10*, 640–654.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning.* Reading: Addison-Wesley.

Gomez, F. J., & Miikkulainen, R. (1999). Solving non-markovian control tasks with neuro-evolution. In D. Thomas (Ed.), *Proceedings of the 16th international joint conference on artificial intelligence (IJCAI-99-vol2)* (pp. 1356–1361). S.F.: Morgan Kaufmann Publishers.

Gullapalli, V. (1991). A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural networks*, *3*, 671–692.

Hallam, B. (1999). *Simulating animal conditioning: Investigating Halperin's neuro-connector model.* PhD thesis, University of Edinburgh.

Hansen, E. A., Barto, A. G., & Zilberstein, S. (1996). Reinforcement learning for mixed open-loop and closed-loop control. In *Advances in neural information processing systems (nips).* Cambridge, MA: MIT Press.

Happel, B. L. M., & Murre, J. M. J. (1994). Design and evolution of modular neural-network architectures. *Neural Networks*, *7*(6-7), 985–1004.

Harmon, M. E., & Baird, L. C. (1996). *Multi-player residual advantage learning with general function approximation* (Technical report No. WL-TR-1065). Wright-Patterson Air Force Base Ohio: Wright Laboratory.

Hauskrecht, M. (2000). Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research*, *13*, 33-94.

Hernandez-Gardiol, N., & Mahadevan, S. (2001). Hierarchical memory-based reinforcement learning. In *Advances in neural information processing systems, NIPS'2000* (Vol. 13).

Hertz, J., Krogh, A., & Palmer, R. G. (1991). *Introduction to the theory of neural computation* (1st ed.). Reading, MA: Addison-Wesley.

Hinton, G. (1999). Invited talk. In *Workshop "the future and prospects of neural networks" at icann'99.*

Hinton, G. E. (1987). *Connectionist learning procedures* (Tech. Rep. No. Computer Science Technical Report). Pittsburgh, PA.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen.* Diploma thesis, Fakultät für Informatik, Technische Universität München.

Hochreiter, S., Bengio, Y., Frasconi, P., & Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer & J. F. Kolen (Eds.), *A field guide to dynamical recurrent neural networks.* IEEE Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9 (8)*, 1735–1780.

Holland, J. (1975). *Adaptation in natural and artificial systems.* Ann Arbor, MI: The University of Michigan Press.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation.* Reading, MA: Addison-Wesley.

Houk, J. C., Adams, J. L., & Barto, A. G. (1995). A model of how the basal ganglia generates and uses neural signals that predict reinforcement. In J. C. Houk, J. Davis, & D. Beiser (Eds.), *Models of information processing in the basal ganglia* (pp. 249–270). Cambridge, Mass: M.I.T. Press.

Humphrys, M. (1997). *Action selection methods using reinforcement learning.* Unpublished doctoral dissertation, University of Cambridge, Computer Laboratory.

Hutter, M. (2003). *A gentle introduction to the universal algorithmic agent aixi* (Technical report No. IDSIA-01-03). Manno-Lugano, Switzerland: IDSIA.

Jackson, J. H. (1870). A study of convulsions. In J. Taylor (Ed.), *Selected writings of john hughlings jackson* (Vol. 1, pp. 8–36). New York: Basic Books.

Jordan, M. I., & Rumelhart, D. E. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science*, *16*, 370–354.

Kaelbling, L. P. (1990). *Learning in embedded systems.* PhD thesis, Dept. of Computer Science, Stanford University.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101.*

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kakade, S., & Dayan, P. (2000). Dopamine bonuses. In *Advances in neural information processing systems (nips) 13.* Cambridge, MA: MIT Press.

Kawato, M. (1990). Computational schemes and neural network models for formation and control of multi-joint arm trajectory. In W. T. Miller, R. S. Sutton, & P. J. Werbos (Eds.), *Neural networks for control.* Cambridge, Mass: M.I.T. Press.

Keijzer, F. A. (2001). *Representation and behavior.* Cambridge, MA: MIT Press.

Kelso, J. A. S. (1995). *Dynamic patterns: The self-organization of brain and behavior.* Cambridge, MA: MIT Press.

Kohonen, T. (1995). *Self-organizing maps.* Berlin: Springer-Verlag.

Kretchmar, R. M., & Anderson, C. (1997). Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the international conference on neural networks, ICNN'97.*

Kröse, B. J. A., & van Dam, J. W. M. (1992). Learning to avoid collisions: A reinforcement learning paradigm for mobile robot navigation. In *Proceedings of the 1992 IFAC/IFIP/IMACS symposium on artificial intelligence in real-time control.* Delft.

Krotkov, E. P., & Simmons, R. G. (1996). Perception, planning and control for autonomous walking with the ambler planetary rover. *International Journal of Robotics Research, 15*, 155–180.

Kwee, I., Hutter, M., & Schmidhuber, J. (2001). Market-based reinforcement learning in partially observable worlds. *Proceedings of the International Conference on Artificial Neural Networks (ICANN-2001)*, 865–873.

Lanzi, P. L. (2000). Adaptive agents with reinforcement learning and internal memory. In J.-A. Meyer, D. Floreano, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 6: Proceedings of the sixth international conference on simulation of adaptive behavior* (pp. 333–342). Cambridge, MA: MIT Press.

Lenat, D. B., & Guha, R. V. (1990). *Building large knowledge-based systems.* Reading, MA: Addison-Wesley.

Levelt, W. J. M., Roelofs, A., & Meyer, A. S. (1999). A theory of lexical access in speech production. *Behavioral and Brain Sciences, 22*, 1–75.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning, 8*, 293–321.

Lin, L.-J., & Mitchell, T. (1992). *Memory approaches to reinforcement learning in non-markovian domains* (Technical report No. CMU-CS-92-138). Carnegie Mellon University, School of Computer Science.

Lin, L.-J., & Mitchell, T. (1993). Reinforcement learning with hidden states. In J.-A. Meyer, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 2: Proceedings of the second international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Linåker, F., & Jacobsson, H. (2001). Mobile robot learning of delayed response tasks through event extraction: A solution to the road sign problem and beyond. In *Proceedings of the International joint Conference on Artificial Intelligence, IJCAI'2001.*

Linåker, F., & Niklasson, L. (2000). Time series segmentation using an adaptive resource allocating vector quantization network based on change detection. In *Proceedings of the international joint conference on neural networks, ijcnn'2000* (pp. 323–328).

Littman, M. L. (1993). An optimization-based categorization of reinforcement learning environments. In J.-A. Meyer, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 2: Proceedings of the second international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Littman, M. L. (1994). Memoryless policies: theoretical limitations and practical results. In D. Cliff, P. Husbands, J.-A. Meyer, & S. W. Wilson (Eds.), *From animals to animats 3: Proceedings of the third international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995a). Learning policies for partially observable environments: Scaling up. In *Proceedings of the twelfth international conference on machine learning.*

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995b). *Learning policies for partially observable environments: Scaling up* (Tech. Rep.). Dept. of Computer Science, Brown University.

Loch, J., & Singh, S. (1998). Using eligibility traces to find the best memoryless policy in Partially Observable Markov Decision Processes. In *Proc. of icml'98.*

Malott, R. W., Whaley, D. L., & Malott, M. E. (1993). *Elementary principles of behavior.* Englewood CLiffs, NJ: Prentice Hall.

Mataric, M. (1997). Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence, 9: 2–3*, 323–336.

Mataric, M. J. (1991). Navigating with a rat brain: A neurobiologically-inspired model for robot spatial representation. In J.-A. Meyer & S. Wilson (Eds.), *Proceedings of the first international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the tenth international machine learning conference.*

McCallum, R. A. (1995). Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings of the twelfth international conference on machine learning* (pp. 387–395). San Francisco, CA: Morgan Kauffman.

McCallum, R. A. (1996). Learning to use selective attention and short-term memory in sequential tasks. In *From animals to animats 4: Proceedings of the fourth international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

McCallum, R. A. (1997). *Reinforcement learning with selective perception and hidden state.* Unpublished doctoral dissertation, University of Rochester, Computer Science Department.

Meeden, L., McGraw, G., & Blank, D. (1993). Emergent control and planning in an autonomous vehicle. In D. S. Touretsky (Ed.), *Proceedings of the fifteenth annual meeting of the cognitive science society* (pp. 735–740). Hillsdale, NJ: Lawrence Erlbaum Associates.

Meuleau, N., Peshkin, L., Kim, K. E., & Kaelbling, L. P. (1999). Learning finite-state controllers for partially observable environments. In *Proceedings of the fifteenth conference on uncertainty in artificial intelligence.*

Michie, D., & Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In D. E & M. D. (Eds.), *Machine intelligence 2* (pp. 137–152). Edinburgh: Oliver and Boyd.

Minsky, M. (1961). Steps towards artificial intelligence. *Proceedings of the IRE*, *49*, 8–30.

Minsky, M., & Papert, S. (1969). *Perceptrons.* Cambridge, MA: MIT Press.

Mitchell, T. M. (1980). *The need for biases in learning generalizations* (Technical Report No. CBM-TR-117). New Brunswick, New Jersey: Department of Computer Science, Rutgers University.

Moravec, H. P. (1982). The Stanford Cart and the CMU Rover. *Proceedings of the IEEE, 71 (7)*, 872–884.

Moriarty, D. E., & Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, *22*, 11.

Moriarty, D. E., Schultz, A. C., & Grefenstette, J. J. (1999). Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, *11*, 199–229.

Murphy, K. P. (2000). *A survey of POMDP solution techniques* (Tech. Rep.). Computer Science Division, University of California at Berkeley.

Murre, J. M. J. (1992). *Categorization and learning in neural networks.* PhD thesis, Leiden University.

Newell, A., & Simon, H. A. (1976). Computer science as empirical enquiry: Symbols and search. *Communications of the Association for Computing Machinery, 19*, 113-126.

Nguyen, D., & Widrow, B. (1990). The truck backer-upper: An example of self-learning in neural networks. In W. T. Miller, III, R. S. Sutton, & P. J. Werbos (Eds.), *Neural networks for control* (pp. 287–299). MIT Press.

Nilsson, N. J. (1984). *Shakey the robot* (Tech. Rep.). SRI A.I. Technical Note 323, April.

Nolfi, S., & Floreano, D. (1998). Co-evolving predator and prey robots: Do 'arms races' arise in artificial evolution? *Artificial Life, 4(4).*

O'Keefe, J., & Nadel, L. (1978). *The Hippocampus as a Cognitive Map.* Clarendon Press.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. In M. I. Jordan, M. J. Kearns, & S. A. Solla (Eds.), *Advances in neural information processing systems* (Vol. 10). The MIT Press.

Peng, J., & Williams, R. J. (1996). Technical note: Incremental multi-step Q-learning. *Machine Learning*, *22*, 283.

Peshkin, L., & de Jong, E. D. (2002). Context-based policy search: transfer of experience across problems. In *Proceedings of the ICML-2002 workshop on development of representations.*

Peshkin, L., Meuleau, N., & Kaelbling, L. P. (1999). Learning policies with external memory. In *Proceedings of the sixteenth international conference on machine learning.*

Pollack, J. (1991). The induction of dynamical recognizers. *Machine Learning*, *7*, 227–252.

Pylyshyn, Z. W. (1984). *Computation and cognition.* Cambridge, MA: MIT Press.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavior model. *Computer Graphics, 21(4)*, 25–34.

Ring, M. B. (1993a). Learning sequential tasks by incrementally adding higher orders. In C. L. Giles, S. J. Hanson, & J. D. Cowan (Eds.), *Advances in neural information processing systems 5* (pp. 115–122). San Mateo, California: Morgan Kaufmann Publishers.

Ring, M. B. (1993b). Two methods for hierarchy learning in reinforcement environments. In J. A. Meyer, H. Roitblat, & S. Wilson (Eds.), *From animals to animats 2: Proceedings of the second international conference on simulation of adaptive behavior* (pp. 148–155). MIT Press.

Rodriguez, P., Wiles, J., & Elman, J. (1999). A recurrent neural network that learns to count. *Connection Science*, *11*(1), 5-40.

Rosen, B. E., Goodwin, J. M., & Vidal, J. J. (1991). Adaptive range coding. In *Advances in neural information processing systems* (Vol. 3). MIT Press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1: Foundations). Cambridge, MA: MIT Press.

Rumelhart, D. E., & McClelland, J. L. (1986a). On learning the past tense of english verbs. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition. Volume 2: Psychological and biological models* (pp. 216–271). Cambridge, MA: MIT Press.

Rumelhart, D. E., & McClelland, J. L. (Eds.). (1986b). *Parallel distributed processing: Explorations in the microstructure of cognition.* Cambridge, MA: MIT Press.

Rummery, G. A., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems* (Technical report No. CUED/F-INFENG/TR 166). Engineering Department, Cambridge University.

Rylatt, R. M., & Czarnecki, C. A. (2000). Embedding connectionist autonomous agents in time: The 'road sign problem'. *Neural Processing Letters*, *12*, 145–158.

Schmidhuber, J. (1990). Networks adjusting networks. In *Proc. of distributed adaptive neural information processing.* St. Augustin.

Schmidhuber, J. (1991a). Curious model-building control systems. In *Proceedings of the international joint conference on neural networks, IJCNN'91* (Vol. 2, pp. 1458–1463). Singapore.

Schmidhuber, J. (1991b). A possibility for implementing curiosity and boredom in model-building neural controllers. In J. A. Meyer & S. W. Wilson (Eds.), *Proc. of the international conference on simulation of adaptive behavior: From animals to animats* (p. 222-227). MIT Press/Bradford Books.

Schmidhuber, J. (1991c). Reinforcement learning in Markovian and non-Markovian environments. In D. S. Touretzky (Ed.), *Advances in neural information processing systems 3* (pp. 500–506). San Mateo, CA: Morgan Kauffman.

Schmidhuber, J. (1992a). Learning complex, extended sequences using the principle of history compression [Letter]. *Neural Computation*, *4*(2), 234–242.

Schmidhuber, J. (1992b). Learning to control fast-weight memories: An alternative to dynamic recurrent networks [Letter]. *Neural Computation*, *4*(1), 131–139.

Schmidhuber, J. (2002). *Optimal ordered problem solver* (Technical report No. IDSIA-12-02). Manno-Lugano, Switzerland: IDSIA.

Schmidhuber, J., & Zhao, J. (1999). Direct policy search and uncertain policy evaluation. In *Aaai spring symposium on search under uncertain and incomplete information, stanford univ.* (p. 119-124). American Association for Artificial Intelligence, Menlo Park, Calif.

Schmidhuber, J., Zhao, J., & Schraudolph, N. (1997). Reinforcement learning with self-modifying policies. In S. Thrun & L. Pratt (Eds.), *Learning to learn* (pp. 293–309). Kluwer.

Schmidhuber, J., Zhao, J., & Wiering, M. (1996). *Simple principles of metalearning* (Tech. Rep. No. IDSIA-69-96). IDSIA.

Schultz, W., Dayan, P., & Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, *275*, 1593–1599.

Simmons, P., & Young, D. (1999). *Nerve cells and animal behavior.* Cambridge: Cambridge University Press.

Simon, H. S. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society, 106 (6)*, 467–482.

Singh, S. P., Jaakkola, T., & Jordan, M. I. (1994). Learning without state-estimation in partially observable Markovian decision processes. In *Proc. 11th international conference on machine learning* (pp. 284–292). Morgan Kaufmann.

Skinner, B. F. (1938, 1991). *The behavior of organisms: An experimental analysis.* Acton, MA: Copley.

Slocum, A. C., Downey, D. C., & Beer, R. D. (2000). Further experiments in the evolution of minimally cognitive behavior: From perceiving affordances to selective attention. In J.-A. Meyer, D. Floreano, H. L. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 6: Proceedings of the sixth international conference on simulation of adaptive behavior* (pp. 430–439). Cambridge, MA: MIT Press.

Steels, L. (1995). Intelligence - dynamics and representations. In L. Steels (Ed.), *The biology and technology of intelligent autonomous agents.* Berlin: Springer-Verlag.

Steels, L. (1997). Synthesising the origins of language and meaning using co-evolution, self-organisation and level formation. In J. Hurford, C. Knight, & M. Studdert-Kennedy (Eds.), *Evolution of human language.* Edinburgh: Edinburgh Univ. Press.

Stroop, J. R. (1935). Studies of interference in serial verbal reactions. *Journal of Experimental Psychology, 18*, 643–662.

Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning.* Unpublished doctoral dissertation, University of Massachusetts, Amherst.

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

Sutton, R. S. (1989). *Implementation details of the TD(λ) procedure for the case of vector predictions and backpropagation* (Technical report No. TN87-509.1). GTE Laboratories.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the seventh international conference on machine learning* (pp. 216–224). Austin, TX: Morgan Kaufmann.

Sutton, R. S. (1991). Reinforcement learning architectures. In J.-A. Meyer & S. Wilson (Eds.), *Proceedings of the third international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo (Eds.), *Advances in neural information processing systems: Proceedings of the 1995 conference* (pp. 1038–1044). San Francisco: Morgan Kauffman.

Sutton, R. S., & Barto, A. G. (1981). Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, *88*, 135–170.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems* (Vol. 12). MIT Press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Suykens, J. A. K., De Moor, B. L. R., & Vandewalle, J. (1997). NLq theory: A neural control framework with global asymptotic stability criteria. *Neural Networks*, *10*(4), 615–637.

Tani, J., & Nolfi, S. (1998). Learning to perceive as articulated. In *From animals to animats 5: Proceedings of the fifth international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

ten Hagen, S. H. G., & Kröse, B. J. A. (1998). Pseudo-parametric Q-learning using feedforward neural networks. In L. Niklasson, M. Bodén, & T. Ziemke (Eds.), *Icann'98, proceedings of the international conference on artificial neural networks* (pp. 449–454). Springer-Verlag.

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

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play [Note]. *Neural Computation*, *6*(2), 215-219.

Thorndike, E. L. (1911). *Animal intelligence.* Darien, CT: Hafner.

Thrun, S. (2000). Monte carlo POMDPs. In S. Solla, T. Leen, & K.-R. Müller (Eds.), *Advances in neural information processing systems 12* (pp. 1064–1070). MIT Press.

Thrun, S. B. (2000). Probabilistic algorithms in robotics. *AI Magazine*, *21 (4)*, 93–109.

Thrun, S. B., & Möller, K. (1992). Active exploration in dynamic environments. In J. E. Moody, S. J. Hanson, & R. P. Lippmann (Eds.), *Advances in neural information processing systems* (Vol. 4, pp. 531–538). Morgan Kaufmann Publishers, Inc.

Tolman, E. C. (1948). Cognitive maps in rats and men. *Psychological Review, 55*, 189–208.

Trullier, O., & Meyer, J. (1998). Animat navigation using a cognitive graph. In *From animals to animats 5: Proceedings of the fifth international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Tsitsiklis, J. N., & Roy, B. V. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Trans. Auto. Control*, *42*(5), 674–690.

Turing, A. M. (1950). Computing machinery and intelligence. *Proceedings of the American Philosophical Society*, *49*, 433–460.

Ulbricht, C. (1996). Handling time-warped sequences with neural networks. In *From animals to animats 4: Proceedings of the fourth international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Urzelai, J., & Floreano, D. (2001). Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments. *Evolutionary Computation*, *9*, 495–524.

van Dartel, M. (2001). *Internal states: What are they?* Master's thesis, Cognitive Psychology Department, University of Maastricht.

van Gelder, T. (1998). The dynamical hypothesis in cognitive science. *Behavioral and brain sciences*, *21 (5)*, 1–75.

Vlassis, N., Terwijn, B., & Kröse, B. J. A. (2002). Auxiliary particle filter robot localization from high-dimensional sensor observations. In *Proceedings of the IEEE international conference on robotics and automation, ICRA'02.*

Wagatsuma, H., & Yamaguchi, Y. (1999). A neural network model self-organizing a cognitive map using theta phase precession. In *Ieee smc'99 conference proceedings. 1999 ieee international conference on systems, man, and cybernetics.* (Vol. 3, pp. 199–204). Piscataway, NJ: IEEE Service Center.

Walter, W. G. (1950). An imitation of life. *Scientific American, 182(5)*, 42–45.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* PhD thesis, Cambridge University.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279-292.

Webb, B. (1994). Robotic experiments in cricket phonotaxis. In D. Cliff, P. Husbands, J.-A. Meyer, & S. Wilson (Eds.), *Proceedings of the third international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Werbos, P. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences.* PhD thesis, Harvard University, Cambridge, MA.

Werbos, P. J. (1990). A menu of designs for reinforcement learning over time. In W. T. Miller, R. S. Sutton, & P. J. Werbos (Eds.), *Neural networks for control* (pp. 67–95). Cambridge, Mass: M.I.T. Press.

Werbos, P. J. (1992). Approximate dynamic control for real-time control and neural modeling. In D. A. White & D. A. Sofge (Eds.), *Handbook of intelligent control—neural, fuzzy, and adaptive approaches* (pp. 493–526). Van Norstrand Reinhold.

Werner, G. M. (1994). Using second order neural connections for motivation of behavioral choices. In D. Cliff, P. Husbands, J.-A. Meyer, & S. Wilson (Eds.), *Proceedings of the third international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Whitley, D., Dominic, S., Das, R., & Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, *13*, 259.

Whitley, D., Gruau, F., & Pyeatt, L. (1995). Cellular encoding applied to neurocontrol. In L. Eshelman (Ed.), *Genetic algorithms: Proceedings of the sixth international conference (icga95)* (pp. 460–467). Pittsburgh, PA, USA: Morgan Kaufmann.

Wieland, A. P. (1991). Evolving neural network controllers for unstable systems. In *Proceedings of ijcnn-91* (Vol. II, pp. 667–673).

Wiering, M., & Schmidhuber, J. (1997). HQ-learning. *Adaptive Behavior*, *6:2*, 219–246.

Wiering, M., & Schmidhuber, J. (1998). Fast online Q(lambda). *Machine Learning*, *33 (1)*, 105–115.

Wiles, J., & Elman, J. (1995). Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *In proceedings of the seventeenth annual conference of the cognitive science society* (pp. pages 482 – 487). Cambridge, MA: MIT Press.

Williams, R. J. (1990). Adaptive state representation and estimation using recurrent connectionist networks. In W. T. Miller, R. S. Sutton, & P. J. Werbos (Eds.), *Neural networks for control.* Cambridge, MA: MIT Press.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running recurrent neural networks. *Neural Computation*, *1*(2), 270–280.

Wilson, S. W. (1991). The animat path to AI. In *From animals to animats 1: Proceedings of the first international conference on simulation of adaptive behavior.* Cambridge, MA: MIT Press.

Wilson, S. W. (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation*, *2*(1), 1–18.

Wilson, S. W. (1995). Classifier Fitness Based on Accuracy. *Evolutionary Computation*, *3*(2), 149–175.

Wilson, S. W. (1998). Generalization in the XCS classifier system. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, & R. Riolo (Eds.), *Genetic programming 1998: Proceedings of the third annual conference* (pp. 665–674). University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Inform. Control*, *34*, 286–295.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67-82.

Wyatt, J. (1995). Issues in putting reinforcement learning onto robots. In *Mobile robotics workshop, 10th biennial conference of the AISB.* Sheffield.

Yamauchi, B. M., & Beer, R. D. (1994). Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, *2*(3), 219–246.

Zhao, J., & Schmidhuber, J. (1998). Solving a complex prisoner's dilemma with self-modifying policies. In J. A. Meyer & S. W. Wilson (Eds.), *Proceedings of the sixth international conference on simulation of adaptive behavior: From animals to animats 6* (p. 177-182). MIT Press/Bradford Books.

Publications

Several chapters in this thesis are based on or formed the basis for refereed publications:

- Chapter 2: Bakker, B. (2000). The Adaptive Behavior Approach to Psychology. *Cognitive Processing, 1*, 39-70.
- Chapter 4: Bakker, B., and van der Voort van der Kleij, G. (2000). Trading off Perception with Internal State: Reinforcement Learning and Analysis of Q-Elman Networks in a Markovian Task. In S.-I. Amari, C.L. Giles, M. Gori, and V. Piuri (Eds.), *Proceedings of the International Joint Conference on Neural Networks 2000, Vol. III*, 213-218.
- Chapter 5: Bakker, B., and de Jong, M. (2000). The Epsilon State Count. In J.-A. Meyer, A. Berthoz, D. Floreano, H. Roitblat, and S.W. Wilson (Eds.), *From Animals to Animats 6: Proceedings of The Sixth International Conference on Simulation of Adaptive Behavior*, 51-60, Cambridge, MA: MIT Press.
- Chapter 6: Bakker, B. (2002). Reinforcement Learning with Long Short-Term Memory. In T.G. Dietterich, S. Becker, and Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems, 14*. Cambridge, MA: MIT Press.
- Chapter 7: Bakker, B., Linåker, F., and Schmidhuber, J. (2002). Reinforcement Learning in Partially Observable Mobile Robot Domains Using Unsupervised Event Extraction. In *Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS2002.*

Samenvatting

Het achterliggende idee bij het werk van dit proefschrift en soortgelijk werk is dat het mogelijk is, en zelfs belangrijk, om kennis te verwerven over zowel kunstmatige intelligentie als natuurlijke, biologische intelligentie door mathematische en computationele studies te doen. Er zijn algemene principes van intelligentie, en als we die beter leren begrijpen middels zulke studies, vergaren we kennis over zowel kunstmatige intelligentie als biologische intelligentie. In dit proefschrift is hiertoe werk gedaan binnen de context van *adaptive behavior* onderzoek. Meer specifiek gaat dit proefschrift over *reinforcement learning*, leren op basis van beloning en straf.

De hoofdvraag die het proefschrift behandelt is de vraag hoe reinforcement learning technieken gebruikt kunnen worden in leertaken waar observaties van de omgeving door het lerende systeem onvoldoende of moeilijk te interpreteren informatie verschaffen over de toestand van het systeem in de omgeving. Dit zijn met name leertaken die partieel observeerbare problemen (POMDPs) genoemd worden. In zulke situaties moet het lerende systeem variabele interne toestanden hebben die als korte-termijn geheugen fungeren, om op basis van herinneringen aan eerdere observaties en uitgevoerde acties te kunnen afleiden wat de huidige toestand in de omgeving is. Dit soort taken komt in realistische toepassingen en omgevingen veel voor, en is daarom van groot belang om te begrijpen en op te lossen. Dit geldt niet alleen voor het veld van reinforcement learning, maar voor het veld van adaptive behavior in het algemeen. Dit proefschrift onderzoekt met name hoe dit soort taken opgelost kan worden met behulp van recurrente neurale netwerken. De recurrente activatie-waardes, aanwezig in het netwerk via interne terugkoppelingslussen, leveren de benodigde variabele interne toestanden.

De recurrente neurale netwerken worden in dit proefschrift getraind met relatief standaard reinforcement learning-algoritmes die als doel hebben het schatten van *value* functies over de toestand-actie ruimte, en die dit doen door de zogenaamde temporele veranderingen in de geschatte value functie als voorspellings-fout te beschouwen en te minimaliseren. De toestand van de omgeving wordt hier dus impliciet geschat middels de huidige observatie (de huidige invoer van het netwerk) samen met de interne toestand van het netwerk (de recurrente activatie-waardes).

Het elegante aan deze benadering is dat aangezien de interne toestanden gecreëerd worden op basis van fouten in de value functie, de toestandsschatting gebaseerd is op *nut*. Dit betekent dat de toestandsschatting alleen dan aangepast wordt wanneer dat belangrijk is voor verbetering van het gedrag van het lerende systeem, dat wil zeggen als het systeem hiermee meer beloning kan krijgen. De toestandsschatting

wordt in principe niet veranderd als dit weliswaar nodig zou zijn om de absoluut correcte toestand van het systeem in de omgeving te bepalen, maar dit niet belangrijk is om meer beloning te krijgen. Dit principe kan belangrijk zijn in leeromgevingen die zeer complex zijn, en waarin daardoor de exacte toestandsschatting zeer moeilijk is, maar waarin door alleen te letten op nut van de toestandsschatting toch goed gedrag geleerd kan worden.

Na een korte inleiding in hoofdstuk 1 wordt in hoofdstuk 2 de adaptive behavior benadering beschreven, met speciale nadruk op verschillen met meer traditionele kunstmatige intelligentie en op de relevantie voor de cognitieve psychologie.

In hoofdstuk 3 wordt een overzicht gegeven van reinforcement learning. Hierbij is aandacht voor de algemene eigenschappen van reinforcement learning taken en voor gebruikelijke algoritmes, en er is speciaal veel aandacht voor die types problemen en algoritmes gerelateerd aan de hoofdvraag van dit proefschrift, partieel observeerbare problemen.

Hoofdstuk 4 is het eerste "technische" hoofdstuk. Hierin wordt een argument beschreven aangaande het gebruik van interne toestanden in taken waar dit strikt gesproken niet noodzakelijk is (volledig observeerbare taken, MDPs). Het idee is dat als observaties van de omgeving weliswaar in principe volledige informatie geven over de toestand in de omgeving, maar die informatie moeilijk te extraheren is doordat hiervoor complexe (statische) patroonherkenning nodig is, het soms beter is om toch een lerend systeem met interne toestanden te gebruiken. Het leren van nuttige interne toestanden kan dan makkelijker zijn dan het oplossen van het moeilijke statische patroonherkenningsprobleem. Experimenten worden gepresenteerd die dit argument illustreren en er volgt een uitgebreide analyse van de getrainde systemen, in dit hoofdstuk Elman recurrente neurale netwerken getraind met Q-learning.

Hoofdstuk 5 is het eerste hoofdstuk dat experimenten behandelt waarin interne toestanden niet alleen nuttig zijn, maar ook werkelijk nodig: partieel observeerbare problemen. Opnieuw worden Elman netwerken gebruikt, met ditmaal een nieuwe variatie op Q-learning, Advantage(λ) learning. Twee soorten taken worden onderzocht: een partieel observeerbare versie van balanceren van een geïnverteerde pendulum, en een partieel observeerbaar doolhof navigatie probleem. In het laatste probleem wordt een uitgebreide analyse gegeven van de capaciteiten en eigenschappen van het getrainde netwerk, en in het bijzonder wordt onderzocht in hoeverre we hier kunnen spreken over het geleerd hebben van een "cognitieve landkaart".

Hoofdstuk 6 behandelt het probleem waar Elman netwerken en sowieso de meeste recurrente neurale netwerkarchitecturen en andere architecturen met variabele toestanden aan leiden: het probleem van het leren van complexe en lange termijnafhankelijkheden. Als de huidige toestand van het systeem in de omgeving alleen bepaald kan worden door iets te onthouden van lang geleden of door een complexe temporele regelmatigheid uit de vroegere observaties en acties te detecteren, is dat moeilijk voor zulke architecturen. In het geval van recurrente neurale netwerken is dit in belangrijke mate terug te voeren op een snel verval van gradienten geschat ten opzichte van gebeurtenissen verder in het verleden. Dit hoofdstuk onderzoekt het gebruik in de context van reinforcement learning van een specifieke recurrente neurale netwerk architectuur die ontworpen is om dit probleem het hoofd te bieden bij supervised learning: het Long Short-Term Memory (LSTM) netwerk. In een aantal

verschillende experimenten wordt aangetoond dat LSTM geschikt is om met dergelijke POMDPs om te gaan. Hierbij is wel van belang dat meer gericht wordt geëxploreerd dan gebruikelijk is in reinforcement learning. Dit is met name zo omdat het voor het detecteren van complexe en lange termijn-afhankelijkheden belangrijk is dat de sequentie van invoersignalen redelijk stabiel is. Dit kan bereikt worden door selectief minder te exploreren in bepaalde toestanden waarvan de value redelijk goed bekend is, zodat de sequentie van acties en daarmee de sequentie van erop volgende observaties stabieler wordt. Een algoritme wordt gepresenteerd dat dit realiseert. Voor de verschillende experimenten worden verder analyses gepresenteerd die laten zien hoe de verschillende elementen van LSTM samenwerken in dit soort taken.

In hoofdstuk 7 wordt het lerende systeem van hoofdstuk 6 gecombineerd met een extra, unsupervised lerende component. Deze component doet *event extraction*: het extraheert uit een continue stroom van sensor-data, bijvoorbeeld afkomstig van een robot, significante, stabiele veranderingen. Zulke veranderingen zijn de momenten die in zekere zin "nieuw" en "verrassend" zijn, en daardoor van belang geacht worden om mogelijk het gedrag te veranderen. Dit is dan ook het moment dat het reinforcement lerende LSTM netwerk een nieuw gedrag selecteert. Het idee is dat op deze manier omgegaan kan worden met bepaalde soorten complexe leerproblemen die zonder deze event extraction zeer grote toestandsruimtes en zeer lange termijn-afhankelijkheden hebben. Experimenten worden beschreven die deze principes illustreren. Wat we hier feitelijk hebben is een hiërarchisch besturingssysteem waar lager-niveau gedragingen worden aangestuurd (geïnhibeerd en geëxciteerd) door een hoger-niveau systeem. Argumenten worden gegeven waarom zulke hiërarchische besturingssystemen belangrijk en krachtig zijn.

Het laatste hoofdstuk, hoofdstuk 8, behandelt puntsgewijs de technische en conceptuele bijdragen van dit proefschrift. Tenslotte worden nieuwe of, als ze niet nieuw zijn, belangrijke onderzoeksvragen voor de toekomst beschreven binnen de context van dit soort onderzoek en wordt kort teruggekeken op de doelen van het proefschrift zoals geformuleerd in het begin van het proefschrift.

Curriculum Vitae

Bram Bakker was born in Leidschendam, the Netherlands, on 3 September 1972. He attended V.W.O. ("Preparatory Scientific Education") high school at "het Loo" in Voorburg from 1984 to 1990. After graduating, he was first an *extraneus* of Aircraft and Space Technology and then a student of Technical Physics, both at the Technical University of Delft, before finally settling in as a student of Psychology at Leiden University in 1992. As part of his undergraduate studies in Psychology, he did a research project in the Brain Simulation Lab at the University of Southern California in Los Angeles. In 1997 he graduated *cum laude* in 1997 with a thesis on "biologically plausible reinforcement learning with feedforward neural networks". After graduation he became a PhD student (aio) in the Unit of Experimental and Theoretical Psychology (now Unit of Cognitive Psychology) of Leiden University, which culminated in this thesis. He is currently a part-time postdoctoral researcher at the *Istituto Dalle Molle di Studi sull'Intelligenza Artificiale* (IDSIA) in Lugano, Switzerland, and part-time postdoctoral researcher in the Intelligent Autonomous Systems group of the Computer Science department at the University of Amsterdam.