Reinforcement Learning

edited by
Richard S. Sutton

Springer Science+Business Media, LLC

REINFORCEMENT LEARNING

edited by

Richard S. Sutton

GTE Laboratories

A Special Issue of Machine Learning on Reinforcement Learning

Reprinted from Machine Learning
Vol. 8, Nos. 3-4 (1992)

Springer Science+Business Media, LLC

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

KNOWLEDGE REPRESENTATION, LEARNING AND EXPERT SYSTEMS

Consulting Editor

Tom Mitchell
Carnegie Mellon University

UNIVERSAL SUBGOALING AND CHUNKING OF GOAL HIERARCHIES, J. Laird, P. Rosenbloom, A. Newell, ISBN: 0-89838-213-0
MACHINE LEARNING: A Guide to Current Research, T. Mitchell, J. Carbonell, R. Michalski, ISBN: 0-89838-214-9
MACHINE LEARNING OF INDUCTIVE BIAS, P. Utgoff, ISBN: 0-89838-223-8
A CONNECTIONIST MACHINE FOR GENETIC HILLCLIMBING, D. H. Ackley, ISBN: 0-89838-236-X
LEARNING FROM GOOD AND BAD DATA, P. D. Laird, ISBN: 0-89838-263-7
MACHINE LEARNING OF ROBOT ASSEMBLY PLANS, A. M. Segre, ISBN: 0-89838-269-6
AUTOMATING KNOWLEDGE ACQUISITION FOR EXPERT SYSTEMS, S. Marcus, Editor, ISBN: 0-89838-294-7
MACHINE LEARNING, META-REASONING AND LOGICS, P. B. Brazdil, K. Konolige, ISBN: 0-7923-9047-4
CHANGE OF REPRESENTATION AND INDUCTIVE BIAS: D. P. Benjamin, ISBN: 0-7923-9055-5
KNOWLEDGE ACQUISITION: SELECTED RESEARCH AND COMMENTARY, S. Marcus, Editor, ISBN: 0-7923-9062-8
LEARNING WITH NESTED GENERALIZED EXEMPLARS, S. L. Salzberg, ISBN: 0-7923-9110-1
INCREMENTAL VERSION-SPACE MERGING: A General Framework for Concept Learning, H. Hirsh, ISBN: 0-7923-9119-5
COMPETITIVELY INHIBITED NEURAL NETWORKS FOR ADAPTIVE PARAMETER ESTIMATION, M. Lemmon, ISBN: 0-7923-9086-5
STRUCTURE LEVEL ADAPTATION FOR ARTIFICIAL NEURAL NETWORKS, T.C. Lee, ISBN: 0-7923-9151-9
CONNECTIONIST APPROACHES TO LANGUAGE LEARNING, D. Touretzky, ISBN: 0-7923-9216-7
AN INTRODUCTION TO FUZZY LOGIC APPLICATIONS IN INTELLIGENT SYSTEMS, R. Yager, L. Zadeh, ISBN: 0-7923-9191-8

MACHINE LEARNING/*Vol. 8 Nos. 3/4 May 1992*

REINFORCEMENT LEARNING

A Special Issue of Machine Learning on Reinforcement Learning

Library of Congress Cataloging-in-Publication Data

Reinforcement learning / edited by Richard S. Sutton.
p. cm. -- (The Kluwer international series in engineering and computer science ; SECS 173)
"A special issue of Machine learning ... reprinted from ... vol. 8, nos. 3-4 (1992)."
Includes bibliographical references and index.
ISBN 978-1-4613-6608-9 ISBN 978-1-4615-3618-5 (eBook)
DOI 10.1007/978-1-4615-3618-5
1. Reinforcement learning (Machine Learning) I. Sutton, Richard S. II. Series.
Q325.6.R45 1992
006.3'1--dc20 92-7567
CIP

Printed on acid-free paper.

Machine Learning, 8, 225-227 (1992)

Introduction: The Challenge of Reinforcement Learning

Reinforcement learning is the learning of a mapping from situations to actions so as to maximize a scalar reward or reinforcement signal. The learner is not told which action to take, as in most forms of machine learning, but instead must discover which actions yield the highest reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate's reward, but also the next situation, and through that all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Reinforcement learning is both a new and very old topic in AI. The term appears to have been coined by Minsky (1961), and independently in control theory by Waltz and Fu (1965). The earliest machine learning research now viewed as directly relevant was Samuel's (1959) checker player, which used temporal-difference learning to manage delayed reward much as it is used today. Of course learning and reinforcement have been studied in psychology for almost a century, and that work has had a very strong impact on the AI/engineering work. One could in fact consider all of reinforcement learning to be simply the reverse engineering of certain psychological learning processes (e.g., operant conditioning and secondary reinforcement.)[1]

Despite the early papers mentioned above, reinforcement learning was largely forgotten in the late 1960s and the 1970s. Not until the early 1980s did it gradually become an active and identifiable area of machine learning research (Barto, et al., 1981, 1983; see also Hampson, 1983). Research in genetic algorithms and classifier systems, initiated by John Holland (1975, 1986), has also been an influential part of reinforcement learning research, as has learning automata theory (see Narendra & Thathachar, 1974). Most recently, Chris Watkins (1989) and Paul Werbos (1987), among others, have invigorated theoretical research in reinforcement learning by linking it to optimal control theory and dynamic programming.

The seven articles of this special issue are representative of the excellent reinforcement learning research ongoing today. Some are theoretical, some empirical. Most of them use some form of connectionist network as part of their learning method.[2] The article by Williams introduces a gradient theory of reinforcement learning analogous to that available for connectionist supervised learning. Whereas Williams' theory treats the case of immediate reward, the article by Tesauro focusses on delayed reward. Tesauro compares temporal-difference and supervised-learning approaches to learning to play backgammon. Among other surprising results, his temporal-difference program learns to play significantly better than the previous world-champion program and as well as expert human players.

Closely related to temporal-difference learning is Q-learning (Watkins, 1989), currently the most well-understood and widely-used reinforcement learning algorithm. The technical note by Watkins and Dayan presents for the first time a complete proof of the convergence of Q-learning, a landmark result in reinforcement learning theory. The papers by Lin and by Singh take up the broader challenge of extending and scaling Q-learning and other simple reinforcement-learning methods so that they are applicable to larger and harder tasks. In one of the largest systematic comparisons of learning methods, Lin demonstrates significantly accelerated learning using novel methods for teaching-by-example and re-using prior experience. Singh's article opens up an important new direction in which to extend reinforcement learning methods—structuring them to permit transfer from simple tasks to larger, composite tasks. The next paper, by Dayan, uses Q-learning techniques to extend the theory of temporal-difference learning methods and weaken their reliance on a Markov-world assumption. Finally, Millan and Torras's paper on path-finding is noteworthy for its use of continuous rather than discrete state and action spaces. This is the first work that I know of to combine continuous actions with temporal-difference learning.

It gives me great pleasure to have assembled and to present this set of papers. These works constitute an excellent introduction to modern reinforcement learning research, but they are by no means complete. I would be remiss if I did not mention at least some of the other ongoing reinforcement-learning work, including that by Barto, et al. (1991) on dynamic programming, by Whitehead and Ballard (1991) on active perception, by Mahadevan and Connell (1990) on Q-learning in robots, and by Booker (1988) and Grefensteete, et al. (1990) on reinforcement learning in genetic systems. Many more interesting papers can be found in the proceedings of recent machine learning meetings. An excellent tutorial introduction to reinforcement learning remains to be written, but the best choices for a place to start are either the theses by Kaelbling (1990) or Watkins (1989), or else the early papers by Barto, et al. (1981).

Part of the appeal of reinforcement learning is that it is in a sense the whole AI problem in a microcosm. The task is that of an autonomous learning agent interacting with its world to achieve a goal. The framework permits the simplifications necessary in order to make progress, while at the sime time including and highlighting cases that are clearly beyond our current capabilities, cases that we will not be able to solve effectively until many key problems of learning and representation have been solved. That is the challenge of reinforcement learning.

Richard S. Sutton
GTE Laboratories Incorporated
Waltham, MA 02254
(SUTTON@GTE.COM)

Notes

1. Psychologists do not use exactly the term "reinforcement learning," so we can feel free to use it, as I do here, to refer exclusively to the engineering enterprise.
2. This is typical, but by no means necessary, as is shown by reinforcement learning research that instead uses genetic algorithms (e.g., Grefenstette, et al., 1990).

References

Barto, A.G. Bradtke, S.J. & Singh, S.P. (1991). *Real-time learning and control using asynchronous dynamic programming* (Technical Report 91-57). Amherst, MA: University of Massachusetts, Computer Science Department.

Barto, A.G. & Sutton, R.S. (1981). Landmark learning: An illustration of associative search. *Biological Cybernetics, 42,* 1-8.

Barto, A.G., Sutton, R.S. & Anderson, C.W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Trans. on Systems, Man, and Cybernetics, SMC-13,* 834-846.

Barto, A.G., Sutton, R.S. & Brouwer, P.S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics, 40,* 201-211.

Booker, L.B. (1988). Classifier systems that learn world models. *Machine Learning, 3,* 161-192.

Grefenstette, J.J., Ramsey, C.L. & Schultz, A.C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning, 5,* 355-382.

Hampson, S.E. (1983). *A neural model of adaptive behavior.* Ph.D. dissertation, Dept. of Information and Computer Science, Univ. of Calif., Irvine (Technical Report #213). A revised edition appeared as *Connectionist Problem Solving,* Boston: Birkhäuser, 1990.

Holland, J.H. (1975). *Adaptation in natural and artificial systems.* Ann Arbor, MI: Univ. of Michigan Press.

Holland, J.H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In: R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine learning, An artificial intelligence approach, Volume II,* 593-623, Los Altos, CA: Morgan Kaufman.

Kaelbling, L.P. (1990). *Learning in embedded systems.* Ph.D. dissertation, Computer Science Dept, Stanford University.

Mahadevan, S. & Connell, J. (1990). Automatic programming of behavior-based robots using reinforcement learning. IBM technical report. To appear in *Artificial Intelligence.*

Minsky, M.L. (1961). Steps toward artificial intelligence. *Proceedings IRE, 49,* 8-30. Reprinted in E.A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought,* 406-450, New York: McGraw-Hill, 1963.

Narendra, K.S. & Thathachar, M.A.L. (1974). Learning automata—a survey. *IEEE Transactions on Systems, Man, and Cybernetics, 4,* 323-334. (Or see their textbook, *Learning Automata: An Introduction,* Englewood Cliffs, NJ: Prentice Hall, 1989.)

Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development, 3,* 210-229. Reprinted in E.A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought,* 71-105, New York: McGraw-Hill, 1963.

Waltz, M.D. & Fu, K.S. (1965). A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Automatic Control, AC-10,* 390-398.

Watkins, C.J.C.H. (1989). *Learning with delayed rewards.* Ph.D. dissertation, Psychology Department, Cambridge University.

Werbos, P.J. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man and Cybernetics,* Jan-Feb.

Whitehead, S.D. & Ballard, D.H. (1991). Learning to perceive and act by trial and error. *Machine Learning, 7,* 45-84.

Machine Learning, 8, 229-256 (1992)

Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

RONALD J. WILLIAMS rjw@corwin.ccs.northeastern.edu
College of Computer Science, 161 CN, Northeastern University, 360 Huntington Ave., Boston, MA 02115

Abstract. This article presents a general class of associative reinforcement learning algorithms for connectionist networks containing stochastic units. These algorithms, called REINFORCE algorithms, are shown to make weight adjustments in a direction that lies along the gradient of expected reinforcement in both immediate-reinforcement tasks and certain limited forms of delayed-reinforcement tasks, and they do this without explicitly computing gradient estimates or even storing information from which such estimates could be computed. Specific examples of such algorithms are presented, some of which bear a close relationship to certain existing algorithms while others are novel but potentially interesting in their own right. Also given are results that show how such algorithms can be naturally integrated with backpropagation. We close with a brief discussion of a number of additional issues surrounding the use of such algorithms, including what is known about their limiting behaviors as well as further considerations that might be used to help develop similar but potentially more powerful reinforcement learning algorithms.

Keywords. Reinforcement learning, connectionist networks, gradient descent, mathematical analysis

1. Introduction

The general framework of reinforcement learning encompasses a broad variety of problems ranging from various forms of function optimization at one extreme to learning control at the other. While research in these individual areas tends to emphasize different sets of issues in isolation, it is likely that effective reinforcement learning techniques for autonomous agents operating in realistic environments will have to address all of these issues jointly. Thus while it remains a useful research strategy to focus on limited forms of reinforcement learning problems simply to keep the problems tractable, it is important to keep in mind that eventual solutions to the most challenging problems will probably require integration of a broad range of applicable techniques.

In this article we present analytical results concerning certain algorithms for tasks that are *associative*, meaning that the learner is required to perform an input-output mapping, and, with one limited exception, that involve *immediate reinforcement*, meaning that the reinforcement (i.e., payoff) provided to the learner is determined by the most recent input-output pair only. While *delayed reinforcement* tasks are obviously important and are receiving much-deserved attention lately, a widely used approach to developing algorithms for such tasks is to combine an immediate-reinforcement learner with an adaptive predictor or *critic* based on the use of *temporal difference methods* (Sutton, 1988). The *actor-critic* algorithms investigated by Barto, Sutton and Anderson (1983) and by Sutton (1984) are clearly of this form, as is the *Q-learning* algorithm of Watkins (1989; Barto, Sutton, & Watkins, 1990).

A further assumption we make here is that the learner's search behavior, always a necessary component of any form of reinforcement learning algorithm, is provided by means of randomness in the input-output behavior of the learner. While this is a common way to achieve the desired exploratory behavior, it is worth noting that other strategies are sometimes available in certain cases, including systematic search or consistent selection of the apparent best alternative. This latter strategy works in situations where the goodness of alternative actions is determined by estimates which are always overly optimistic and which become more realistic with continued experience, as occurs for example in A^* search (Nilsson, 1980).

In addition, all results will be framed here in terms of connectionist networks, and the main focus is on algorithms that follow or estimate a relevant gradient. While such algorithms are known to have a number of limitations, there are a number of reasons why their study can be useful. First, as experience with backpropagation (leCun, 1985; Parker, 1985; Rumelhart, Hinton & Williams, 1986; Werbos, 1974) has shown, the gradient seems to provide a powerful and general heuristic basis for generating algorithms which are often simple to implement and surprisingly effective in many cases. Second, when more sophisticated algorithms are required, gradient computation can often serve as the core of such algorithms. Also, to the extent that certain existing algorithms resemble the algorithms arising from such a gradient analysis, our understanding of them may be enhanced.

Another distinguishing property of the algorithms presented here is that while they can be described roughtly as statistically climbing an appropriate gradient, they manage to do this without explicitly computing an estimate of this gradient or even storing information from which such an estimate could be directly computed. This is the reason they have been called *simple* in the title. Perhaps a more informative adjective would be *non-model-based*. This point is discussed further in a later section of this paper.

Although we adopt a connectionist perspective here, it should be noted that certain aspects of the analysis performed carry over directly to other ways of implementing adaptive input-ouput mappings. The results to be presented apply in general to any learner whose input-output mappings consists of a parameterized input-controlled distribution function from which outputs are randomly generated, and the corresponding algorithms modify the learner's distribution function on the basis of performance feedback. Because of the gradient approach used here, the only restriction on the potential applicability of these results is that certain obvious differentiability conditions must be met.

A number of the results presented here have appeared in various form in several earlier technical reports and conference papers (Williams, 1986; 1987a; 1987b; 1988a; 1988b).

2. Reinforcement-learning connectionist networks

Unless otherwise specified, we assume throughout that the learning agent is a feedforward network consisting of several individual units, each of which is itself a learning agent. We begin by making the additional assumption that all units operate stochastically, but later it will be useful to consider the case when there are deterministic units in the net as well. The network operates by receiving external input from the environment, propagating the

corresponding activity through the net, and sending the activity produced at its output units to the environment for evaluation. The evaluation consists of the scalar reinforcement signal r, which we assume is broadcast to all units in the net. At this point each unit performs an appropriate modification of its weights, based on the particular learning algorithm in use, and the cycle begins again.

The notation we use throughout is as follows: Let y_i denote the output of the ith unit in the network, and let $\mathbf{x}^i$ denote the pattern of input to that unit. This pattern of input $\mathbf{x}^i$ is a vector whose individual elements (typically denoted x_j) are either the outputs of certain units in the network (those sending their output directly to the ith unit) or certain inputs from the environment (if that unit happens to be connected so that it receives input directly from the environment). Then output y_i is drawn from a distribution depending on $\mathbf{x}^i$ and the weights w_{ij} on input lines to this unit. For each i, let $\mathbf{w}^i$ denote the weight vector consisting of all the weights w_{ij}. The let $\mathbf{W}$ denote the weight matrix consisting of all weights w_{ij} in the network. In a more general setting, $\mathbf{w}^i$ can be viewed as the collection of all parameters on which the behavior of the ith unit (or agent) depends, while $\mathbf{W}$ is the collection of parameters on which the behavior of the entire (or collection of agents) depends.

In addition, for each i let $g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = Pr\{y_i = \xi \,|\, \mathbf{w}^i, \mathbf{x}^i\}$, so that g_i is the probability mass function determining the value of y_i as a function of the parameters of the unit and its input. (For ease of exposition, we consistently use terminology and notation appropriate for the case when the set of possible output values y_i is discrete, but the results to be derived also apply to continuous-valued units when g_i is taken to be the corresponding probability density function.) Since the vector $\mathbf{w}^i$ contains all network parameters on which the input-output behavior of the ith unit depends, we could just as well have defined g_i by $g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = Pr\{y_i = \xi \,|\, \mathbf{W}, \mathbf{x}^i\}$.

Note that many of the quantities we have named here, such as r, y_i, and $\mathbf{x}^i$, actually depend on time, but it is generally convenient in the sequel to suppress explicit reference to this time dependence, with the understanding that when several such quantities appear in a single equation they represent the values for the same time step t. We assume that each new time step begins just before external input is presented to the network. In the context of immediate-reinforcement tasks we also call each time step's cycle of network-environment interaction a *trial*.

To illustrate these definitions and also introduce a useful subclass, we define a *stochastic semilinear unit* to be one whose output y_i is drawn from some given probability distribution whose mass function has a single parameter p_i, which is in turn computed as

$$p_i = f_i(s_i), \tag{1}$$

where f_i is a differentiable squashing function and

$$s_i = \mathbf{w}^{i\mathrm{T}}\mathbf{x}^i = \sum_j w_{ij}\, x_j, \tag{2}$$

the inner product of $\mathbf{w}^i$ and $\mathbf{x}^i$. This can be viewed as a semilinear unit, as widely used in connectionist networks, followed by a singly parameterized random number generator.

A noteworthy special case of a stochastic semilinear unit is a *Bernoulli semilinear unit*, for which the output y_i is a Bernoulli random variable with parameter p_i, which means that the only possible output values are 0 and 1, with $Pr\{y_i = 0 | \mathbf{w}^i, \mathbf{x}^i\} = 1 - p_i$ and $Pr\{y_i = 1 | \mathbf{w}^i, \mathbf{x}^i\} = p_i$. Thus, for a Bernoulli semilinear unit,

$$g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = \begin{cases} 1 - p_i & \text{if } \xi = 0 \\ p_i & \text{if } \xi = 1, \end{cases}$$

where p_i is computed via equations (1) and (2). This type of unit is common in networks using stochastic units; it appears, for example, in the Boltzmann machine (Hinton & Sejnowski, 1986) and in the reinforcement learning networks explored by Barto and colleagues (Barto & Anderson, 1985; Barto & Jordan, 1987; Barto, Sutton, & Brouwer, 1981). While the name *Bernoulli semilinear unit* may thus appear to be simply a fancy new name for something quite familiar, use of this term is intended to emphasize its membership in a potentially much more general class.

A particular form of squashing function commonly used is the *logistic function*, given by

$$f_i(s_i) = \frac{1}{1 + e^{-s_i}}. \tag{3}$$

A stochastic semilinear unit using both the Bernoulli random number generator and the logistic squashing function will be called a *Bernoulli-logistic* unit.

Now we observe that the class of Bernoulli semilinear units includes certain types of units whose computation is alternatively described in terms of a linear threshold computation together with either additive input noise or a noisy threshold. This observation is useful because this latter formulation is the one used by Barto and colleagues (Barto, 1985; Barto & Anandan, 1985; Barto & Anderson, 1985; Barto, Sutton, & Anderson, 1983; Barto, Sutton, & Brouwer, 1981; Sutton, 1984) in their investigations. Specifically, they assume a unit computes it output y_i by

$$y_i = \begin{cases} 1 & \text{if } \sum_j w_{ij} x_j + \eta > 0 \\ 0 & \text{otherwise,} \end{cases}$$

where η is drawn randomly from a given distribution Ψ. To see that such a unit may be viewed as a Bernoulli semilinear unit, let

$$s_i = \sum_j w_{ij} x_j$$

and observe that

$$
\begin{aligned}
Pr\{y_i = 1 \mid \mathbf{w}^i, \mathbf{x}^i\} &= Pr\{y_i = 1 \mid s_i\} \\
&= Pr\{s_i + \eta > 0 \mid s_i\} \\
&= 1 - Pr\{s_i + \eta \leq 0 \mid s_i\} \\
&= 1 - Pr\{\eta \leq -s_i \mid s_i\} \\
&= 1 - \Psi(-s_i).
\end{aligned}
$$

Thus, as long as Ψ is differentiable, such a unit is a Bernoulli semilinear unit with squashing function f_i given by $f_i(s_i) = 1 - \Psi(-s_i)$.

3. The expected reinforcement performance criterion

In order to consider gradient learning algorithms, it is necessary to have a performance measure to optimize. A very natural one for any immediate-reinforcement learning problem, associative or not, is the expected value of the reinforcement signal, conditioned on a particular choice of parameters of the learning system. Thus, for a reinforcement-learning network, our performance measure is $E\{r \mid \mathbf{W}\}$, where E denotes the expectation operator, r the reinforcement signal, and $\mathbf{W}$ the network's weight matrix. We need to use expected values here because of the potential randomness in any of the following: (1) the environment's choice of input to the network; (2) the network's choice of output corresponding to any particular input; and (3) the environment's choice of reinforcement value for any particular input/output pair. Note that it only makes sense to discuss $E\{r \mid \mathbf{W}\}$ independently of time if we assume that the first and third sources of randomness are determined by stationary distributions, with the environment's choice of input pattern to the net also determined independently across time. In the absence of such assumptions, the expected value of r for any given time step may be a function of time as well as of the history of the system. Thus we tacitly assume throughout that these stationarity and independence conditions hold.

Note that, with these assumptions, $E\{r \mid \mathbf{W}\}$ is a well-defined, deterministic function of $\mathbf{W}$ (but one which is unknown to the learning system). Thus, in this formulation, the objective of the reinforcement learning system is to search the space of all possible weight matrices $\mathbf{W}$ for a point where $E\{r \mid \mathbf{W}\}$ is maximum.

4. REINFORCE algorithms

Consider a network facing an associative immediate-reinforcement learning task. Recall that weights are adjusted in this network following receipt of the reinforcement value r at each trial. Suppose that the learning algorithm for this network is such that at the end of each trial each parameter w_{ij} in the network is incremented by an amount

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij})e_{ij},$$

where α_{ij} is a *learning rate factor*, b_{ij} is a *reinforcement baseline*, and $e_{ij} = \partial \ln g_i/\partial w_{ij}$ is called the *characteristic eligibility* of w_{ij}. Suppose further that the reinforcement baseline b_{ij} is conditionally independent of y_i, given $\mathbf{W}$ and $\mathbf{x}^i$, and the rate factor α_{ij} is nonnegative and depends at most on $\mathbf{w}^i$ and t. (Typically, α_{ij} will be taken to be a constant.) Any learning algorithm having this particular form will be called a *REINFORCE* algorithm. The name is an acronym for "*RE*ward *I*ncrement = *N*onnegative *F*actor × *O*ffset *R*einforcement × *C*haracteristic *E*ligibility," which describes the form of the algorithm.

What makes this class of algorithms interesting is the following mathematical result:

Theorem 1. For any REINFORCE algorithm, the inner product of $E\{\Delta\mathbf{W} \mid \mathbf{W}\}$ and $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\}$ is nonnegative. Furthermore, if $\alpha_{ij} > 0$ for all i and j, then this inner product is zero only when $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} = 0$. Also, if $\alpha_{ij} = \alpha$ is independent of i and j, then $E\{\Delta\mathbf{W} \mid \mathbf{W}\} = \alpha \nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\}$.

This results relates $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\}$, the gradient in weight space of the performance measure $E\{r \mid \mathbf{W}\}$, to $E\{\Delta\mathbf{W} \mid \mathbf{W}\}$, the average update vector in weight space, for any REINFORCE algorithm.[1] Specifically, it says that for any such algorithm the average update vector in weight space lies in a direction for which this performance measure is increasing. The last sentence of the theorem is equivalent to the claim that for each weight w_{ij} the quantity $(r - b_{ij})\,\partial \ln g_i/\partial w_{ij}$ represents an unbiased estimate of $\partial E\{r \mid \mathbf{W}\}/\partial w_{ij}$. A proof of this theorem is given in Appendix A.

There are a number of interesting special cases of such algorithms, some of which coincide with algorithms already proposed and explored in the literature and some of which are novel. We begin by showing that some existing algorithms are REINFORCE algorithms, from which it follows immediately that Theorem 1 applies to them. Later we will consider some novel algorithms belonging to this class.

Consider first a Bernoulli unit having no (nonreinforcement) input and suppose that the parameter to be adapted is $p_i = Pr\{y_i = 1\}$. This is equivalent to a two-action stochastic learning automaton (Narendra & Thathatchar, 1989) whose actions are labeled 0 and 1. The probability mass function g_i is then given by

$$g_i(y_i, p_i) = \begin{cases} 1 - p_i & \text{if } y_i = 0 \\ p_i & \text{if } y_i = 1, \end{cases} \tag{4}$$

from which it follows that the characteristic eligibility for the parameter p_i is given by

$$\frac{\partial \ln g_i}{\partial p_i}(y_i, p_i) = \begin{cases} -\dfrac{1}{1 - p_i} & \text{if } y_i = 0 \\ \dfrac{1}{p_i} & \text{if } y_i = 1 \end{cases} = \frac{y_i - p_i}{p_i(1 - p_i)} \tag{5}$$

assuming p_i is not equal to 0 or 1.

A particular REINFORCE algorithm for such a unit can be obtained by choosing $b_i = 0$ for the reinforcement baseline and by using as the rate factor $\alpha_i = \rho_i p_i(1 - p_i)$, where $0 < \rho_i < 1$. This gives rise to an algorithm having the form

$$\Delta p_i = \rho_i \, r(y_i - p_i),$$

using the result (5) above. The special case of this algorithm when the reinforcement signal is limited to 0 and 1 coincides with the 2-action version of the *linear reward-inaction* (L_{R-I}) *stochastic learning automaton* (Narendra & Thathatchar, 1989). A "network" consisting of more than one such unit constitutes a *team* of such learning automata, each using its own individual learning rate. The behavior of teams of L_{R-I} automata has been investigated by Narendra and Wheeler (1983; Wheeler & Narendra, 1986).

Now consider a Bernoulli semilinear unit. In this case, $g_i(y_i, \mathbf{w}^i, \mathbf{x}^i)$ is given by the right-hand side of (4) above, where p_i is expressed in terms of $\mathbf{w}^i$ and $\mathbf{x}^i$ using equations (1) and (2). To compute the characteristic eligibility for a particular parameter w_{ij}, we use the chain rule. Differentiating the equations (1) and (2) yields $dp_i/ds_i = f_i'(s_i)$ and $\partial s_i/\partial w_{ij} = x_j$. Noting that $\partial \ln g_i/\partial p_i \, (y_i, \mathbf{w}^i, \mathbf{x}^i)$ is given by the right-hand side of (5) above, we multiply these three quantities to find that the characteristic eligibility for the weight w_{ij} is given by

$$\frac{\partial \ln g_i}{\partial w_{ij}} \, (y_i, \mathbf{w}^i, \mathbf{x}^i) = \frac{y_i - p_i}{p_i(1 - p_i)} \, f_i'(s_i) \, x_j, \tag{6}$$

as long as p_i is not equal to 0 or 1. In the special case when f_i is the logistic function, given by equation (3), p_i is never equal to 0 or 1 and $f_i'(s_i) = p_i(1 - p_i)$, so the characteristic eligibility of w_{ij} is simply

$$\frac{\partial \ln g_i}{\partial w_{ij}} \, (y_i, \mathbf{w}^i, \mathbf{x}^i) = (y_i - p_i) \, x_j. \tag{7}$$

Now consider an arbitrary network of such Bernoulli-logistic units. Setting $\alpha_{ij} = \alpha$ and $b_{ij} = 0$ for all i and j gives rise to a REINFORCE algorithm having the form

$$\Delta w_{ij} = \alpha r(y_i - p_i) \, x_j, \tag{8}$$

using the result (7) above. It is interesting to compare this with the *associative reward-penalty* (A_{R-P}) algorithm (Barto, 1985; Barto & Anandan, 1985; Barto & Anderson, 1985; Barto & Jordan, 1987), which, for $r \in [0, 1]$, uses the learning rule

$$\Delta w_{ij} = \alpha[r(y_i - p_i) + \lambda(1 - r)(1 - y_i - p_i)] \, x_j.$$

where α is a positive learning rate parameter and $0 < \lambda \leq 1$. If $\lambda = 0$, this is called the *associative reward-inaction* (A_{R-I}) algorithm, and we see that the learning rule reduces to equation (8) in this case. Thus A_{R-I}, when applied to a network of Bernoulli-logistic units, is a REINFORCE algorithm.

In all the examples considered so far, the reinforcement baseline is 0. However, the use of *reinforcement comparison* (Sutton, 1984) is also consistent with the REINFORCE formulation. For this strategy one maintains an adaptive estimate $\bar{r}$ of upcoming reinforcement based on past experience. As a particular example, for a network of Bernoulli-logistic units one may use the learning rule

$$\Delta w_{ij} = \alpha(r - \bar{r})(y_i - p_i)\, x_j, \tag{9}$$

which is then a REINFORCE algorithm as long as the computation of $\bar{r}$ is never based on the current value of y_i (or the current value of r). One common approach to computing $\bar{r}$ is to use the exponential averaging scheme

$$\bar{r}(t) = \gamma r(t - 1) + (1 - \gamma)\bar{r}(t - 1), \tag{10}$$

where $0 < \gamma \leq 1$. More sophisticated strategies are also consistent with the REINFORCE framework, including making $\bar{r}$ a function of the current input pattern $\mathbf{x}^i$ to the unit.

While the analytical results given here offer no basis for comparing various choices of reinforcement baseline in REINFORCE algorithms, it is generally believed that the use of reinforcement comparison leads to algorithms having superior performance in general. We discuss questions of REINFORCE algorithm performance at greater length below.

5. Episodic REINFORCE algorithms

Now we consider how the REINFORCE class of algorithms can be extended to certain learning problems having a temporal credit-assignment component, as may occur when the network contains loops or the environment delivers reinforcement values with unknown, possibly variable, delays. In particular, assume a net N is trained on an episode-by-episode basis, where each episode consists of k time steps, during which the units may recompute their outputs and the environment may alter its non-reinforcement input to the system at each time step. A single reinforcement value r is delivered to the net at the end of each episode.

The derivation of this algorithm is based on the use of the "unfolding-in-time" mapping, which yields for any arbitrary network N operating through a fixed period of time another network N^* having no cycles but exhibiting corresponding behavior. The unfolded network N^* is obtained by duplicating N once for each time step. Formally, this amounts to associating with each time-dependent variable v in N a corresponding time-indexed set of variables $\{v^t\}$ in N^* whose values do not depend on time, and which have the property that $v(t) = v^t$ for all appropriate t. In particular, each weight w_{ij} in N gives rise to several weights w_{ij}^t in N^*, all of whose values happen to be equal to each other and to the value of w_{ij} in N since it is assumed that w_{ij} is constant over the episode.

The form of algorithm to be considered for this problem is as follows: At the conclusion of each episode, each parameter w_{ij} is incremented by

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij}) \sum_{t=1}^{k} e_{ij}(t) \tag{11}$$

where all notation is the same as that defined earlier, with $e_{ij}(t)$ representing the characteristic eligibility for w_{ij} evaluated at the particular time t. By definition, $e_{ij}(t) = e^t_{ij}$, where this latter makes sense within the acyclic network N^*. For example, in a completely interconnected recurrent network of Bernoulli-logistic units that is updated synchronously, $e_{ij}(t) = (y_i(t) - p_i(t))x_j(t - 1)$. All quantities are assumed to satisfy the same conditions required for the REINFORCE algorithm, where, in particular, for each i and j, the reinforcement baseline b_{ij} is independent of any of the output values $y_i(t)$ and the rate factor α_{ij} depends at most on $\mathbf{w}^i$ and episode number. Call any algorithm of this form (and intended for such a learning problem) an *episodic REINFORCE* algorithm.

For example, if the network consists of Bernoulli-logistic units an episodic REINFORCE algorithm would prescribe weight changes according to the rule

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij}) \sum_{t=1}^{k} [y_i(t) - p_i(t)]\, x_j(t - 1).$$

The following result is proved in Appendix A:

Theorem 2. For any episodic REINFORCE algorithm, the inner product of $E\{\Delta \mathbf{W} \mid \mathbf{W}\}$ and $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\}$ is nonnegative. Furthermore, if $\alpha_{ij} > 0$ for all i and j, then this inner product is zero only when $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} = 0$. Also, if $\alpha_{ij} = \alpha$ is independent of i and j, then $E\{\Delta \mathbf{W} \mid \mathbf{W}\} = \alpha \nabla E\{r \mid \mathbf{W}\}$.

What is noteworthy about this algorithm is that it has a plausible on-line implementation using a single accumulator for each parameter w_{ij} in the network. The purpose of this accumulator is to form the eligibility sum, each term of which depends only on the operation of the network as it runs in real time and not on the reinforcement signal eventually received.

A more general formulation of such an episodic learning task is also possible, where reinforcement is delivered to the network at each time step during the episode, not just at the end. In this case the appropriate performance measure is $E\{\Sigma_{t=1}^{k} r(t) \mid \mathbf{W}\}$. One way to create a statistical gradient-following algorithm for this case is to simply replace r in (11) by $\Sigma_{t=1}^{k} r(t)$, but it is interesting to note that when r is *causal*, so that it depends only on network inputs and outputs from earlier times, there is a potentially better way to perform the necessary credit assignment. Roughly, the idea is to treat this learning problem over the k-time-step interval as k different but overlapping episodic learning problems, all starting at the beginning of the episode. We omit further discussion of the details of this approach.

6. REINFORCE with multiparameter distributions

An interesting application of the REINFORCE framework is to the development of learning algorithms for units that determine their scalar output stochastically from multiparameter

distributions rather than the single-parameter distributions used by stochastic semilinear units, for example. One way such a unit may compute in this fashion is for it to first perform a deterministic computation, based on its weights and input, to obtain the values of all parameters controlling the random number generation process, and then draw its output randomly from the appropriate distribution. As a particular example, the normal distribution has two parameters, the mean μ and the standard deviation σ. A unit determining its output according to such a distribution would first compute values of μ and σ deterministically and then draw its output from the normal distribution with mean equal to this value of μ and standard deviation equal to this value of σ.

One potentially useful feature of such a *Gaussian unit* is that the mean and variance of its output are individually controllable as long as separate weights (or perhaps inputs) are used to determine these two parameters. What makes this interesting is that control over σ is tantamount to control over the unit's exploratory behavior. In general, random units using multiparameter distributions have the potential to control their degree of exploratory behavior independently of where they choose to explore, unlike those using single-parameter distributions.

Here we note that REINFORCE algorithms for any such unit are easily derived, using the particular case of a Gaussian unit as an example. Rather than commit to a particular means of determining the mean and standard deviation of such a unit's output from its input and its weights, we will simply treat this unit as if the mean and standard deviation themselves served as the adaptable parameters of the unit. Any more general functional dependence of these parameters on the actual adaptable parameters and input to the unit simply requires application of the chain rule. One particular approach to computation of these parameters, using separate weighted sums across a common set of input lines (and using a somewhat different learning rule), has been explored by Gullapalli (1990). To simplify notation, we focus on one single unit and omit the usual unit index subscript throughout.

For such a unit the set of possible outputs is the set of real numbers and the density function g determining the output y on any single trial is given by

$$g(y, \mu, \sigma) = \frac{1}{(2\pi)^{1/2}\sigma} e^{-(y-\mu)^2/2\sigma^2}.$$

The characteristic eligibility of μ is then

$$\frac{\partial \ln g}{\partial \mu} = \frac{y - \mu}{\sigma^2} \tag{12}$$

and the characteristic eligibility of σ is

$$\frac{\partial \ln g}{\partial \sigma} = \frac{(y - \mu)^2 - \sigma^2}{\sigma^3}.$$

A REINFORCE algorithm for this unit thus has the form

$$\Delta\mu = \alpha_\mu(r - b_\mu)\frac{y - \mu}{\sigma^2} \tag{13}$$

and

$$\Delta\sigma = \alpha_\sigma(r - b_\sigma)\frac{(y - \mu)^2 - \sigma^2}{\sigma^3}, \tag{14}$$

where α_μ, b_μ, α_σ, and b_σ are chosen appropriately. A reasonable algorithm is obtained by setting

$$\alpha_\mu = \alpha_\sigma = \alpha\sigma^2,$$

where α is a suitably small positive constant,[2] and letting $b_\mu = b_\sigma$ be determined according to a reinforcement comparison scheme.

It is interesting to note the resemblance between equation (12), giving the characteristic eligibility for the parameter μ of the normal distribution, and equation (5), giving the characteristic eligibility for the parameter p of the Bernoulli distribution. Since p is the mean and $p(1 - p)$ the variance of the corresponding Bernoulli random variable, both equations have the same form. In fact, the characteristic eligibility of the mean parameter has this form for an even wider variety of distributions, as stated in the following result:

Proposition 1. Suppose that the probability mass or density function g has the form

$$g(y, \mu, \theta_2, \ldots, \theta_k) = \exp[Q(\mu, \theta_2, \ldots, \theta_k)y + D(\mu, \theta_2, \ldots, \theta_k) + S(y)]$$

for some functions Q, D, and S, where $\mu, \theta_2, \ldots, \theta_k$ are parameters such that μ is the mean of the distribution. Then

$$\frac{\partial \ln g}{\partial \mu} = \frac{y - \mu}{\sigma^2},$$

where σ^2 is the variance of the distribution.

Mass or density functions having this form represent special cases of *exponential families of distributions* (Rohatgi, 1976). It is easily checked that a number of familiar distributions, such as the Poisson, exponential, Bernoulli, and normal distributions, are all of this form. A proof of this proposition is given in Appendix B.

7. Compatibility with backpropagation

It is useful to note that REINFORCE, like most other reinforcement learning algorithms for networks of stochastic units, works essentially by measuring the correlation between variations in local behavior and the resulting variations in global performance, as given by the reinforcement signal. When such algorithms are used, all information about the

effect of connectivity between units is ignored; each unit in the network tries to determine the effect of changes of its output on changes in reinforcement independently of its effect on even those units to which it is directly connected. In contrast, the backpropagation algorithm works by making use of the fact that entire chains of effects are predictable from knowledge of the effects of individual units on each other. While the backpropagation algorithm is appropriate only for supervised learning in networks of deterministic units, it makes sense to also use the term *backpropagation* for the single component of this algorithm that determines relevant partial derivatives by means of the backward pass. (In this sense it is simply a computational implementation of the chain rule.) With this meaning of the term we can then consider how backpropagation might be integrated into the statistical gradient-following reinforcement learning algorithms investigated here, thereby giving rise to algorithms that can take advantage of relevant knowledge of network connectivity where appropriate. Here we examine two ways that backpropagation can be used.

7.1. Networks using deterministic hidden units

Consider a feedforward network having stochastic output units and deterministic hidden units. Use of such a network as a reinforcement learning system makes sense because having randomness limited to the output units still allows the necessary exploration to take place.

Let $\mathbf{x}$ denote the vector of network input and let $\mathbf{y}$ denote the network output vector. We can the define $g(\xi, \mathbf{W}, \mathbf{x}) = Pr\{\mathbf{y} = \xi \mid \mathbf{W}, \mathbf{x}\}$ to be the overall probability mass function describing the input-output behavior of the entire network. Except for the fact that the output of the network is generally vector-valued rather than scalar-valued, the formalism and arguments used to derive REINFORCE algorithms apply virtually unchanged when this global rather than local perspective is taken. In particular, a simple extension of the arguments used to prove Theorem 1 to the case of vector-valued output shows that, for any weight w_{ij} in the network, $(r - b_{ij})\, \partial \ln g / \partial w_{ij}$ represents an unbiased estimate of $\partial E\{r \mid \mathbf{W}\}/\partial w_{ij}$.

Let O denote the index set for output units. Because all the randomness is in the output units, and because the randomness is independent across these units, we have

$$Pr\{\mathbf{y} = \xi \mid \mathbf{W}, \mathbf{x}\} = \prod_{k \in O} Pr\{y_k = \xi_k \mid \mathbf{W}, \mathbf{x}\}$$

$$= \prod_{k \in O} Pr\{y_k = \xi_k \mid \mathbf{w}^k, \mathbf{x}^k\},$$

where, for each k, $\mathbf{x}^k$ is the pattern appearing at the input to the kth unit as a result of presentation of the pattern $\mathbf{x}$ to the network. Note that each $\mathbf{x}^k$ depends deterministically on $\mathbf{x}$.

Therefore,

$$\ln g(\xi, \mathbf{W}, \mathbf{x}) = \ln \prod_{k \in O} g_k(\xi_k, \mathbf{w}^k, \mathbf{x}^k) = \sum_{k \in O} \ln g_k(\xi_k, \mathbf{w}^k, \mathbf{x}^k),$$

so that

$$\frac{\partial \ln g}{\partial w_{ij}} (\xi, \mathbf{W}, \mathbf{x}) = \sum_{k \in O} \frac{\partial \ln g_k}{\partial w_{ij}} (\xi_k, \mathbf{w}^k, \mathbf{x}^k).$$

Clearly, this sum may be computed via backpropagation. For example, when the output units are Bernoulli semilinear units, we can use the parameters p_k as intermediate variables and write the characteristic eligibility of any weight w_{ij} as

$$\frac{\partial \ln g}{\partial w_{ij}} = \sum_{k \in O} \frac{\partial \ln g_k}{\partial p_k} \frac{\partial p_k}{\partial w_{ij}},$$

and this is efficiently computed by "injecting"

$$\frac{\partial \ln g_k}{\partial p_k} = \frac{y_k - p_k}{p_k(1 - p_k)}$$

just after the kth unit's squashing function, for each k, and then performing the standard backward pass. Note that if w_{ij} is a weight attached to an output unit, this backpropagation computation simply gives rise to the result (6) derived earlier. For that result we essentially backpropagated the characteristic eligibility of the Bernoulli parameter p_i through the sub-units consisting of the "squasher" and the "summer."

While we have restricted attention here to networks having stochastic output units only, it is not hard to see that such a result can be further generalized to any network containing an arbitrary mixture of stochastic and deterministic units. The overall algorithm in this case consists of the use of the correlation-style REINFORCE computation at each stochastic unit, whether an output unit or not, with backpropagation used to compute (or, more precisely, estimate) all other relevant partial derivatives.

Furthermore, it is not difficult to prove an even more general compatibility between computation of unbiased estimates, not necessarily based on REINFORCE, and backpropagation through deterministic functions. The result is, essentially, that when one set of variables depends deterministically on a second set of variables, backpropagating unbiased estimates of partial derivatives with respect to the first set of variables gives rise to unbiased estimates of partial derivatives with respect to the second set of variables. It is intuitively reasonable that this should be true, but we omit the rigorous mathematical details here since we make no use of the result.

7.2. Backpropagating through random number generators

While the form of algorithm just described makes use of backpropagation within deterministic portions of the network, it still requires a correlation-style computation whenever it is necessary to obtain partial derivative information on the input side of a random number generator. Suppose instead that it were possible to somehow "backpropagate through a random number generator." To see what this might mean, consider a stochastic semilinear unit and suppose that there is a function J having some deterministic dependence on the output y_i. An example of this situation is when the unit is an output unit and $J = E\{r \mid \mathbf{W}\}$, with reinforcement depending on whether the network output is correct or not. What we would like, roughly, is to be able to compute $\partial J/\partial p_i$ from knowledge of $\partial J/\partial y_i$. Because of the randomness, however, we could not expect there to be a deterministic relationship between these quantities. A more reasonable property to ask for is that $\partial E\{J \mid p_i\}/\partial p_i$ be determined by $E\{\partial J/\partial y_i \mid p_i\}$.

Unfortunately, even this property fails to hold in general. For example, in a Bernoulli unit, it is straightforward to check that whenever J is a nonlinear function of y_i there need be no particular relationship between these two quantities. However, if the output of the random number generator can be written as a differentiable function of its parameters, the approach just described for backpropagating through deterministic computation can be applied.

As an illustration, consider a normal random number generator, as used in a Gaussian unit. Its output y is randomly generated according to the parameters μ and σ. We may write

$$y = \mu + \sigma z,$$

where z is a standard normal deviate. From this we see that

$$\frac{\partial y}{\partial \mu} = 1$$

and

$$\frac{\partial y}{\partial \sigma} = z = \frac{y - \mu}{\sigma}.$$

Thus, for example, one may combine the use of backpropagation through Gaussian hidden units with REINFORCE in the output units. In this case the characteristic eligibility for the μ in such a unit is set equal to that computed for the output value y while the characteristic eligibility for the σ parameter is obtained by multiplying that for y by $(y - \mu)/\sigma$. It is worth noting that these particular results in no way depend on the fact that μ is the mean and σ the standard deviation; the identical result applies whenever μ represents a translation parameter and σ a scaling parameter for the distribution. More generally, the same technique can obviously be used whenever the output can be expressed as a function of the parameters together with some auxiliary random variables, as long as the dependence on the parameters is differentiable.

Note that the argument given here is based on the results obtained above for the use of backpropagation when computing the characteristic eligibility in a REINFORCE algorithm, so the conclusion is necessarily limited to this particular use of backpropagation here. Nevertheless, because it is also true that backpropagation preserves the unbiasedness of gradient estimates in general, this form of argument can be applied to yield statistical gradient-following algorithms that make use of backpropagation in a variety of other situations where a network of continuous-valued stochastic units is used. One such application is to supervised training of such networks.

8. Algorithm performance and other issues

8.1. Convergence properties

A major limitation of the analysis performed here is that it does not immediately lead to prediction of the asymptotic properties of REINFORCE algorithms. If such an algorithm does converge, one might expect it to converge to a local maximum, but there need be no such convergence. While there is a clear need for an analytical characterization of the asymptotic behavior of REINFORCE algorithms, such results are not yet available, leaving simulation studies as our primary source of understanding of the behavior of these algorithms. Here we give an overview of some relevant simulation results, some of which have been reported in the literature and some of which are currently only preliminary.

Sutton (1984) studied the performance of a number of algorithms using single-Bernoulli-unit "networks" facing both nonassociative and associative immediate-reinforcement tasks. Among the algorithms investigated were L_{R-I} and one based on equations (9) and (10), which is just REINFORCE using reinforcement comparison. In these studies, REINFORCE with reinforcement comparison was found to outperform all other algorithms investigated.

Williams and Peng (1991) have also investigated a number of variants of REINFORCE in nonassociative function-optimization tasks, using networks of Bernoulli units. These studies have demonstrated that such algorithms tend to converge to local optima, as one might expect of any gradient-following algorithm. Some of the variants examined incorporated modifications designed to help defeat this often undesirable behavior. One particularly interesting variant incorporated an entropy term in the reinforcement signal and helped enable certain network architectures to perform especially well on tasks where a certain amount of hierarchical organization during the search was desirable.

Other preliminary studies have been carried out using networks of Bernoulli units and using single Gaussian units. The Gaussian unit studies are described below. The network studies involved multilayer or recurrent networks facing supervised learning tasks but receiving only reinforcement feedback. In the case of the recurrent networks, the objective was to learn a trajectory and episodic REINFORCE was used. One of the more noteworthy results of these studies was that it often required careful selection of the reinforcement function to obtain solutions using REINFORCE. This is not surprising since it turns out that some of the more obvious reinforcement functions one might select for such problems tend to have severe false maxima. In contrast, A_{R-P} generally succeeds at finding solutions even

when these simpler reinforcement functions are used. Like A_{R-P}, REINFORCE is generally very slow even when it succeeds. Episodic REINFORCE has been found to be especially slow, but this, too, is not surprising since it performs temporal credit-assignment by essentially spreading credit or blame uniformly over all past times.

One REINFORCE algorithm whose asymptotic behavior is reasonably well understood analytically is 2-action L_{R-I}, and simulation experience obtained to date with a number of other REINFORCE algorithms suggests that their range of possible limiting behaviors may, in fact, be similar. The L_{R-I} algorithm is known to converge to a single deterministic choice of action with probability 1. What is noteworthy about this convergence is that, in spite of the fact that the expected motion is always in the direction of the best action, as follows from Theorem 1, there is always a nonzero probability of its converging to an inferior choice of action. A simpler example that exhibits the same kind of behavior is a biased random walk on the integers with absorbing barriers. Even though the motion is biased in a particular direction, there is always a nonzero probability of being absorbed at the other barrier.

In general, a reasonable conjecture consistent with what is known analytically about simple REINFORCE algorithms like L_{R-I} and what has been found in simulations of more sophisticated REINFORCE algorithms is the following: Depending on the choice of reinforcement baseline used, any such algorithm is more or less likely to converge to a local maximum of the expected reinforcement function, with some nonzero (but typically comfortably small) probability of convergence to other points that lead to zero variance in network behavior. For further discussion of the role of the reinforcement baseline, see below.

8.2. Gaussian unit search behavior

For the Gaussian unit studies mentioned above, the problems considered were nonassociative, involving optimization of a function of a single real variable y, and the adaptable parameters were taken to be μ and σ. From equations (13) and (14) it is clear that the reinforcement comparison version of REINFORCE for this unit behaves as follows: If a value y is sampled which leads to a higher function value than has been obtained in the recent past, then μ moves toward y; similarly, μ moves away from points giving lower function values. What is more interesting is how σ is adapted. If the sampled point y gives rise to a higher function value than has been obtained in the recent past, then σ will decrease if $|y - \mu| < \sigma$ but increase if $|y - \mu| > \sigma$. The change made to σ corresponds to that required to make the reoccurence of y more likely. There is corresponding behavior in the opposite direction if the sampled point leads to a lower value. In terms of a search, this amounts to narrowing the search around μ if a better point is found suitably close to the mean or a worse point is found suitably far from the mean, while broadening the search around μ if a worse point is found suitably close to the mean or a better point is found suitably far from the mean. Since the sampled points y are roughly twice as likely to lie within one standard deviation of the mean, it follows that whenever μ sits at the top of a local hill (of sufficient breadth with respect to σ), then σ narrows down to allow convergence to the local maximum. However it is also true that if the local maximum is very flat on top, σ will decrease to the point where sampling worse values becomes extremely unlikely

and then stop changing. Simulation studies using both deterministic and noisy reinforcement confirm this behavior. They also demonstrate that if r is always nonnegative and reinforcement comparison is not used (i.e., $b = 0$), REINFORCE may cause σ to converge to 0 before μ has moved to the top of any hill. This can be viewed as a generalization of the potential convergence to suboptimal performance described earlier for L_{R-I}.

It is interesting to compare REINFORCE for such a unit with an alternative algorithm for the adaptation of μ and σ that has been proposed by Gullapalli (1990). In this approach, μ is adapted in essentially the same manner as in REINFORCE but σ is adapted in a quite different manner. With reinforcement values r assumed to lie between 0 and 1, σ is taken to be proportional to $1 - r$. This strategy makes sense if one takes the point of view that σ is a parameter controlling the scale of the search being performed and the optimum value for the function is unknown. In those situations when it is known that unsatisfactory performance is being achieved it is reasonable to broaden this scale in order to take a coarse-grained view of the search space and identify a broad region in which the optimum has a reasonable chance of being found.

Also relevant here is the work of Schmidhuber and Huber (1990), who have reported successful results using networks having Gaussian output units in control tasks involving *backpropagating through a model* (Jordan & Rumelhart, 1990). In this work, backpropagation through random number generators was used to allow learning of a model and learning of performance to proceed simultaneously rather than in separate phases.

8.3. Choice of reinforcement baseline

One important limitation of the analysis given here is that it offers no basis for choosing among various choices of reinforcement baseline in REINFORCE algorithms. While Theorem 1 applies equally well to any such choice, extensive empirical investigation of such algorithms leads to the inescapable conclusion that use of an adaptive reinforcement baseline incorporating something like the reinforcement comparison strategy can greatly enhance convergence speed, and, in some cases, can lead to a big difference in qualitative behavior as well. One example is given by the Gaussian unit studies described above. A simpler example is provided by a single Bernoulli semilinear unit with only a bias weight and input with its output y affecting the reinforcement r deterministically. If r is always positive, it is easy to see that one obtains a kind of biased random walk behavior when $b = 0$, leading to nonzero probability of convergence to the inferior output value. In contrast, the reinforcement comparison version will lead to values of b lying between the two possible values of r, which leads to motion always toward the better output value.

However, this latter behavior will occur for any choice of b lying between the two possible values for r, so additional considerations must be applied to distinguish among a wide variety of possible adaptive reinforcement baseline schemes. One possibility, considered briefly by Williams (1986) and recently investigated more fully by Dayan (1990), is to pick a reinforcement baseline that minimizes the *variance* of the individual weight changes over time. This turns out to yield not the mean reinforcement as in the usual reinforcement comparison approach, but another quantity that is more difficult to estimate effectively. Dayan's simulation results seem to suggest that use of such a reinforcement baseline offers

a slight improvement in convergence speed over the use of mean reinforcement, but a more convincing advantage remains to be demonstrated.

8.4. Alternate forms for eligibility

REINFORCE, with or without reinforcement comparison, prescribes weight changes proportional to the product of a reinforcement factor that depends only on the current and past reinforcement values and another factor we have called the characteristic eligibility. A straightforward way to obtain a number of variants of REINFORCE is to vary the form of either of these factors. Indeed, the simulation study performed by Sutton (1984) involved a variety of algorithms obtained by systematically varying both of these factors. One particularly interesting variant having this form but not included in that earlier study has since been examined by several investigators (Rich Sutton, personal communication, 1986; Phil Madsen, personal communication, 1987; Williams & Peng, 1991) and found promising. These studies have been conducted only for nonassociative tasks, so this is the form of the algorithm we describe here. (Furthermore, because a principled basis for deriving algorithms of this particular form has not yet been developed, it is somewhat unclear exactly how it should be extended to the associative case.)

We consider specifically the case of a Bernoulli-logistic unit having only a bias weight w. Since the bias input is 1, a standard reinforcement-comparison version of REINFORCE prescribes weight increments of the form

$$\Delta w = \alpha(r - \bar{r})(y - p),$$

where $\bar{r}$ is computed according to the exponential averaging scheme

$$\bar{r}(t) = \gamma r(t - 1) + (1 - \gamma)\bar{r}(t - 1),$$

where $0 < \gamma < 1$. An alternative algorithm is given by the rule

$$\Delta w = \alpha(r - \bar{r})(y - \bar{y}),$$

where $\bar{y}$ is updated by

$$\bar{y}(t) = \gamma y(t - 1) + (1 - \gamma)\bar{y}(t - 1),$$

using the same γ as is used for updating $\bar{r}$. This particular algorithm has been found generally to converge faster and more reliably than the corresponding REINFORCE algorithm.

It is clear that the two algorithms bear some strong similarities. The variant is obtained by simply replacing p by $\bar{y}$, and each of these can be viewed as reasonable *a priori* estimates of the output y. Furthermore, the corresponding strategy can be used to generate variants of REINFORCE in a number of other cases. For example, if the randomness in the unit uses any distribution to which Proposition 1 applies, then the REINFORCE algorithm for adjusting its mean parameter μ will involve the factor $y - \mu$ and we can simply replace

this by $y - \bar{y}$. Such an algorithm for adapting the mean of a Gaussian unit has been tested and found to behave very well.

While some arguments can be given (Rich Sutton, personal communication, 1988) that suggest potential advantages of the use of $y - \bar{y}$ in such algorithms, a more complete analysis has not yet been performed. Interestingly, one possible analytical justification for the use of such algorithms may be found in considerations like those discussed next.

8.5. Use of other local gradient estimates

There are several senses in which it makes sense to call REINFORCE algorithms *simple*, as implied by the title of this paper. First, as is clear from examples given here, the algorithms themselves often have a very simple form. Also, they are simple to derive for essentially any form of random unit computation. But perhaps most significant of all is the fact that, in the sense given by Theorems 1 and 2, they climb an appropriate gradient without explicitly computing any estimate of this gradient or even storing information from which such an estimate could be directly computed. Clearly, there are alternative ways to estimate such gradients and it would be useful to understand how various such techniques can be integrated effectively.

To help distinguish among a variety of alternative approaches, we first define some terminology. Barto, Sutton, and Watkins (1990), have introduced the term *model-based* to describe what essentially correspond to *indirect* algorithms in the adaptive control field (Goodwin & Sin, 1984). These algorithms explicitly estimate relevant parameters underlying the system to be controlled and then use this learned model of the system to compute the control actions. The corresponding notion for an immediate-reinforcement learning system would be one that attempts to learn an explicit model of the reinforcement as a function of learning system input and output, and use this model to guide its parameter adjustments. If these parameter adjustments are to be made along the gradient of expected reinforcement, as in REINFORCE, then this model must actually yield estimates of this gradient. Such an algorithm, using backpropagation through a model, has been proposed and studied by Munro (1987).

This form of model-based approach uses a global model of the reinforcement function and its derivatives, but a more local model-based approach is also possible. This would involve attempting to estimate, at each unit, the expected value of reinforcement as a function of input and output of that unit, or, if a gradient algorithm like REINFORCE is desired, the derivatives of this expected reinforcement. An algorithm studied by Thathatchar and Sastry (1985) for stochastic learning automata keeps track of the average reinforcement received for each action and is thus of this general form. Q-learning (Watkins, 1989) can also be viewed as involving the learning of local (meaning, in this case, *per-state*) models for the cumulative reinforcement.

REINFORCE fails to be model-based even in this local sense, but it may be worthwhile to consider algorithms that do attempt to generate more explicit gradient estimates if their use can lead to algorithms having clearly identifiable strengths. One interesting possibility that applies at least in the nonassociative case is to perform, at each unit, a linear regression of the reinforcement signal on the output of the unit. It is suspected that algorithms

using the $y - \bar{y}$ form of eligibility described above may be related to such an approach but this has not been fully analyzed yet.

9. Conclusion

The analyses presented here, together with a variety of simulation experiments performed by this author and others, suggest that REINFORCE algorithms are useful in their own right and, perhaps more importantly, may serve as a sound basis for developing other more effective reinforcement learning algorithms. One major advantage of the REINFORCE approach is that it represents a prescription for devising statistical gradient-following algorithms for reinforcement-learning networks of units that compute their random output in essentially any arbitrary fashion. Also, because it is a gradient-based approach, it integrates well with other gradient computation techniques such as backpropagation. The main disadvantages are the lack of a general convergence theory applicable to this class of algorithms and, as with all gradient algorithms, an apparent susceptibility to convergence to false optima.

Acknowledgments

I have benefitted immeasurably from numerous discussions with Rich Sutton and Andy Barto on various aspects of the material presented herein. Preparation of this paper was supported by the National Science Foundation under grant IRI-8921275.

Notes

1. In more detail, $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\}$ and $E\{\Delta\mathbf{W} \mid \mathbf{W}\}$ are both vectors having the same dimensionality as $\mathbf{W}$, with the (i, j) coordinate of $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\}$ being $\partial E\{r \mid \mathbf{W}\}/\partial w_{ij}$ and the corresponding coordinate of $E\{\Delta\mathbf{W} \mid \mathbf{W}\}$ being $E\{\Delta w_{ij} \mid \mathbf{W}\}$.
2. Strictly speaking, there is no choice of α for this algorithm guaranteeing that σ will not become negative, unless the normal distribution has its tails truncated (which is necessarily the case in practice). Another approach is to take $\lambda = \ln \sigma$ as the adaptable parameter rather than σ, which leads to an algorithm guaranteed to keep σ positive.

Appendix A

This appendix contains proofs of Theorems 1 and 2 on REINFORCE and episodic REINFORCE algorithms, respectively. In addition to the notation introduced in the text, we symbolize some sets of interest by letting Y_i denote the set of possible output values y_i of the ith unit, with X_i denoting the set of possible values of the input vector $\mathbf{x}^i$ to this unit. Although it is not a critical assumption, we take Y_i and X_i to be discrete sets throughout. Also, we let I denote the index set for elements of $\mathbf{W}$, so that $(i, j) \in I$ if and only if w_{ij} is a parameter in the system.

It should be remarked here that, in the interest of brevity, all the assertions proved in this appendix make use of a convention in which each unbound variable is implicitly assumed to be universally quantified over an appropriate set of values. For example, whenever i and j appear, they are to be considered arbitrary (subject only to $(i, j) \in I$).

A.1. Results for REINFORCE Algorithms

Fact 1.

$$\frac{\partial E\{r \mid \mathbf{W}, \mathbf{x}^i\}}{\partial w_{ij}} = \sum_{\xi \in Y_i} E\{r \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i).$$

Proof. Conditioning on the possible values of the output y_i, we may write

$$\begin{aligned} E\{r \mid \mathbf{W}, \mathbf{x}^i\} &= \sum_{\xi \in Y_i} E\{r \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \, Pr\{y_i = \xi \mid \mathbf{W}, \mathbf{x}^i\} \\ &= \sum_{\xi \in Y_i} E\{r \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \, g_i(\xi, \mathbf{w}^i, \mathbf{x}^i). \end{aligned}$$

Note that specification of the value of y_i causes w_{ij} to have no influence on the ultimate value of r, which means that $E\{r \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\}$ does not depend on w_{ij}. The result then follows by differentiating both sides of this last equation with respect to w_{ij}. ■

Fact 2.

$$\sum_{\xi \in Y_i} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i) = 0.$$

Proof.

$$\sum_{\xi \in Y_i} g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = \sum_{\xi \in Y_i} Pr\{x = \xi \mid \mathbf{w}^i, \mathbf{x}^i\} = 1,$$

and the result follows by differentiating with respect to w_{ij}. ■

Lemma 1. For any REINFORCE algorithm,

$$E\{\Delta w_{ij} \mid \mathbf{W}, \mathbf{x}^i\} = \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}, \mathbf{x}^i\}}{\partial w_{ij}}.$$

Proof. First note that the characteristic eligibility can be written

$$e_{ij} = \frac{\partial \ln g_i}{\partial w_{ij}} = \frac{1}{g_i} \frac{\partial g_i}{\partial w_{ij}}.$$

Although this fails to be defined when $g_i = 0$, it will still be the case that Δw_{ij} is well-defined for any REINFORCE algorithm as long as Y_i is discrete. This is because $g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = 0$ means that the value ξ has zero probability of occurrence as a value of the output y_i.

Then

$$E\{\Delta w_{ij} \mid \mathbf{W}, \mathbf{x}^i\} = \sum_{\xi \in Y_i} E\{\Delta w_{ij} \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \, Pr\{y_i = \xi \mid \mathbf{W}, \mathbf{x}^i\}$$

$$= \sum_{\xi \in Y_i} E\left\{ \frac{\alpha_{ij}(r - b_{ij})}{g_i(\xi, \mathbf{w}^i, \mathbf{x}^i)} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i) \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi \right\} g_i(\xi, \mathbf{w}^i, \mathbf{x}^i)$$

$$= \alpha_{ij} \sum_{\xi \in Y_i} E\{r \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i)$$

$$- \alpha_{ij} \sum_{\xi \in Y_i} E\{b_{ij} \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i),$$

making use of the fact that α_{ij} does not depend on the particular value of the output y_i. By Fact 1, the first term of this last expression is $\alpha_{ij}(\partial E\{r \mid \mathbf{W}, \mathbf{x}^i\}/\partial w_{ij})$. Consider the remaining term. Since $E\{b_{ij} \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} = E\{b_{ij} \mid \mathbf{W}, \mathbf{x}^i\}$, by assumption, we have

$$\sum_{\xi \in Y_i} E\{b_{ij} \mid \mathbf{W}, \mathbf{x}^i, y_i = \xi\} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i) = E\{b_{ij} \mid \mathbf{W}, \mathbf{x}^i\} \sum_{\xi \in Y_i} \frac{\partial g_i}{\partial w_{ij}} (\xi, \mathbf{w}^i, \mathbf{x}^i)$$

$$= 0$$

by Fact 2, and the Lemma is proved. ■

Fact 3.

$$\frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}} = \sum_{x \in X_i} \frac{\partial E\{r \mid \mathbf{W}, \mathbf{x}^i = \mathbf{x}\}}{\partial w_{ij}} \, Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\}.$$

Proof. Conditioning on the possible input patterns $\mathbf{x}^i$, we may write

$$E\{r \mid \mathbf{W}\} = \sum_{x \in X_i} \{r \mid \mathbf{W}, \mathbf{x}^i = \mathbf{x}\} \, Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\}.$$

Note that the weight w_{ij} lies downstream of all computation performed to determine $\mathbf{x}^i$. This means that $Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\}$ does not depend on w_{ij}, so the result follows by differentiating both sides of this last equation by w_{ij}. ■

Lemma 2. For any REINFORCE algorithm,

$$E\{\Delta w_{ij} \mid \mathbf{W}\} = \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}}.$$

Proof.

$$\begin{aligned} E\{\Delta w_{ij} \mid \mathbf{W}\} &= \sum_{x \in X_i} E\{\Delta w_{ij} \mid \mathbf{W}, \mathbf{x}^i = \mathbf{x}\} \, Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\} \\ &= \sum_{x \in X_i} \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}, \mathbf{x}^i = \mathbf{x}\}}{\partial w_{ij}} \, Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\} \\ &= \alpha_{ij} \sum_{x \in X_i} \frac{\partial E\{r \mid \mathbf{W}, \mathbf{x}^i = \mathbf{x}\}}{\partial w_{ij}} \, Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\} \\ &= \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}}, \end{aligned}$$

where the first equality is obtained by conditioning on the possible input patterns to the unit, the second equality follows from Lemma 1, the third equality follows from the assumption that α_{ij} does not depend on the input to the unit, and the last equality follows from Fact 3. ■

Establishing this last result, which is just like Lemma 1 except that the conditioning on input to unit i has been removed from both sides of the equation, is a key step. It relates two quantities that, unlike those of Lemma 1, would be quite messy to compute explicitly in general because $Pr\{\mathbf{x}^i = \mathbf{x} \mid \mathbf{W}\}$ can be quite complicated. From this lemma our main result follows easily.

Theorem 1. For any REINFORCE algorithm, $E\{\Delta \mathbf{W} \mid \mathbf{W}\}^{\mathrm{T}} \nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} \geq 0$. Furthermore, if $\alpha_{ij} > 0$ for all i and j, then equality holds if and only if $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} = 0$.

Proof.

$$E\{\Delta \mathbf{W} \mid \mathbf{W}\}^{\mathrm{T}} \nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} = \sum_{(i,j) \in I} E\{\Delta w_{ij} \mid \mathbf{W}\} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}}$$

$$= \sum_{(i,j)\in I} \alpha_{ij} \left(\frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}} \right)^2,$$

by Lemma 2, and the result is immediate. ■

A.2. Results for episodic REINFORCE algorithms

Analysis of the episodic REINFORCE algorithm is based on the unfolding-in-time mapping, which associates with the original net N its unfolded-in-time acyclic net N^*. The key observation is that having N face its learning problem is equivalent to having N^* face a corresponding associative learning problem. Let $\mathbf{W}^*$ denote the weight matrix for N^*, with its individual components being denoted w^t_{ij}. The weight w^t_{ij} in N^* corresponds to the weight w_{ij} in N at the tth time step, so that $w^t_{ij} = w_{ij}$ for all i, j, and t. Because of the correspondence between these nets, it should be noted that specifying $\mathbf{W}$ is equivalent to specifying $\mathbf{W}^*$. Also, the correspondence between the learning problems is such that we can consider the reinforcement r to be the same for both problems.

Fact 4.

$$\frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}} = \sum_{t=1}^{k} \frac{\partial E\{r \mid \mathbf{W}^*\}}{\partial w^t_{ij}}.$$

Proof. Using the chain rule, we have

$$\frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}} = \sum_{t=1}^{k} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w^t_{ij}} \frac{\partial w^t_{ij}}{\partial w_{ij}} = \sum_{t=1}^{k} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w^t_{ij}} = \sum_{t=1}^{k} \frac{\partial E\{r \mid \mathbf{W}^*\}}{\partial w^t_{ij}},$$

since $w^t_{ij} = w_{ij}$ for all t. ■

Lemma 3. For any episodic REINFORCE algorithm,

$$E\{\Delta w_{ij} \mid \mathbf{W}\} = \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}}.$$

Proof. Let $\Delta w^t_{ij} = \alpha_{ij}(r - b_{ij})e^t_{ij}$, so that $\Delta w_{ij} = \Sigma_{t=1}^{k} \Delta w^t_{ij}$. Note that this represents a REINFORCE algorithm in N^*, so it follows from Lemma 2 that

$$E\{\Delta w^t_{ij} \mid \mathbf{W}^*\} = \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}^*\}}{\partial w^t_{ij}}.$$

But then

$$E\{\Delta w_{ij} \mid \mathbf{W}\} = E\left\{\sum_{t=1}^{k} \Delta w_{ij}^t \mid \mathbf{W}^*\right\}$$

$$= \sum_{t=1}^{k} E\{\Delta w_{ij}^t \mid \mathbf{W}^*\}$$

$$= \sum_{t=1}^{k} \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}^*\}}{\partial w_{ij}^t}$$

$$= \alpha_{ij} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}},$$

where the last equality follows from Fact 4. ■

Theorem 2. For any episodic REINFORCE algorithm, $E\{\Delta\mathbf{W} \mid \mathbf{W}\}^{\mathrm{T}} \nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} \geq 0$. Furthermore, if $\alpha_{ij} > 0$ for all i and j, then equality holds if and only if $\nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} = 0$.

Proof.

$$E\{\Delta\mathbf{W} \mid \mathbf{W}\}^{\mathrm{T}} \nabla_{\mathbf{W}} E\{r \mid \mathbf{W}\} = \sum_{(i,j)\in I} E\{\Delta w_{ij} \mid \mathbf{W}\} \frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}}$$

$$= \sum_{(i,j)\in I} \alpha_{ij} \left(\frac{\partial E\{r \mid \mathbf{W}\}}{\partial w_{ij}}\right)^2,$$

by Lemma 3, and the result is immediate. ■

Note that the proof of Theorem 2 is identical to that for Theorem 1. This is because Theorem 1 uses Lemma 2 and Theorem 2 uses Lemma 3, and both lemmas have the same conclusion.

Appendix B

This appendix is devoted to the proof of the following result:

Proposition 1. Suppose that the probability mass or density function g has the form

$$g(y, \mu, \theta_2, \ldots, \theta_k) = \exp[Q(\mu, \theta_2, \ldots, \theta_k)y + D(\mu, \theta_2, \ldots, \theta_k) + S(y)]$$

for some functions Q, D, and S, where $\mu, \theta_2, \ldots, \theta_k$ are parameters such that μ is the mean of the distribution. Then

$$\frac{\partial \ln g}{\partial \mu} = \frac{y - \mu}{\sigma^2},$$

where σ^2 is the variance of the distribution.

Proof. Here we consider the case of a probability mass function only, but a corresponding argument can be given for a density function.

Let Y denote the support of g. In general,

$$\sum_{y \in Y} g \frac{\partial \ln g}{\partial \mu} = \sum_{y \in Y} \frac{\partial g}{\partial \mu} = \frac{\partial}{\partial \mu} \sum_{y \in Y} g = 0 \tag{15}$$

since $\Sigma_{y \in Y} g = 1$. Combining this with the fact that $\mu = \Sigma_{y \in Y} y g$, we also find that

$$\begin{aligned} \sum_{y \in Y} (y - \mu) g \frac{\partial \ln g}{\partial \mu} &= \sum_{y \in Y} yg \frac{\partial \ln g}{\partial \mu} - \mu \sum_{y \in Y} g \frac{\partial \ln g}{\partial \mu} \\ &= \sum_{y \in Y} y \frac{\partial g}{\partial \mu} \\ &= \frac{\partial}{\partial \mu} \sum_{y \in Y} yg \\ &= 1 \end{aligned} \tag{16}$$

Now introduce the shorthand notation $\alpha = \partial Q / \partial \mu$ and $\beta = \partial D / \partial \mu$. From the hypothesis of the proposition we have

$$\frac{\partial \ln g}{\partial \mu} = \frac{\partial Q}{\partial \mu} y + \frac{\partial D}{\partial \mu} = \alpha y + \beta,$$

so that

$$\sum_{y\in Y} g \frac{\partial \ln g}{\partial \mu} = \sum_{y\in Y} (\alpha y + \beta)\, g = \alpha \sum_{y\in Y} yg + \beta \sum_{y\in Y} g = \alpha\mu + \beta. \tag{17}$$

Also,

$$\begin{aligned} \sum_{y\in Y} (y - \mu) g \frac{\partial \ln g}{\partial \mu} &= \sum_{y\in Y} (y - \mu)(\alpha y + \beta) g \\ &= \sum_{y\in Y} (y - \mu)[\alpha(y - \mu) + \alpha\mu + \beta] g \\ &= \alpha \sum_{y\in Y} (y - \mu)^2 g + (\alpha y + \beta) \sum_{y\in Y} (y - \mu) g \\ &= \alpha\sigma^2 \end{aligned} \tag{18}$$

since $\Sigma_{y\in Y} (y - \mu) g = 0$.

Combining (15) with (17) and (16) with (18), we see that

$$\alpha\mu + \beta = 0$$

and

$$\alpha\sigma^2 = 1,$$

from which it follows that $\alpha = 1/\sigma^2$ and $\beta = -\mu/\sigma^2$. Therefore,

$$\frac{\partial \ln g(y, \mu, \theta_2, \ldots, \theta_k)}{\partial \mu} = \frac{1}{\sigma^2} y - \frac{\mu}{\sigma^2} = \frac{y - \mu}{\sigma^2}.$$ ■

References

Barto, A.G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, 4, 229–256.

Barto, A.G. & Anandan, P. (1985). Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15, 360–374.

Barto, A.G. & Anderson, C.W. (1985). Structural learning in connectionist systems. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, (pp. 43–53). Irvine, CA.

Barto, A.G., Sutton, R.S., & Anderson, C.W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13, 835–846.

Barto, A.G., Sutton, R.S., & Brouwer, P.S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40, 201–211.

Barto, A.G., & Jordan, M.I. (1987). Gradient following without back-propagation in layered networks. *Proceedings of the First Annual International Conference on Neural Networks*, Vol. II (pp. 629–636). San Diego, CA.

Barto, A.G., Sutton, R.S., & Watkins, C.J.C.H. (1990). Learning and sequential decision making. In: M. Gabriel & J.W. Moore (Eds.), *Learning and computational neuroscience: Foundations of adaptive networks*. Cambridge, MA: MIT Press.

Dayan, P. (1990). Reinforcement comparison. In D.S. Touretzky, J.L. Elman, T.J. Sejnowski, & G.E. Hinton (Eds.), *Proceedings of the 1990 Connectionist Models Summer School* (pp. 45–51). San Mateo, CA: Morgan Kaufmann.

Goodwin, G.C. & Sin, K.S. (1984). *Adaptive filtering prediction and control*. Englewood Cliffs, NJ: Prentice-Hall.

Gullapalli, V. (1990). A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural Networks*, 3, 671–692.

Hinton, G.E. & Sejnowski, T.J. (1986). Learning and relearning in Boltzmann machines. In: D.E. Rumelhart & J.L. McClelland, (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1: Foundations*. Cambridge, MA: MIT Press.

Jordan, M.I. & Rumelhart, D.E. (1990). *Forward models: supervised learning with a distal teacher*. (Occasional Paper - 40). Cambridge, MA: Massachusetts Institute of Technology, Center for Cognitive Science.

leCun, Y. (1985). Une procedure d'apprentissage pour resau a sequil assymetrique [A learning procedure for asymmetric threshold networks]. *Proceedings of Cognitiva, 85*, 599–604.

Munro, P. (1987). A dual back-propagation scheme for scalar reward learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 165–176). Seattle, WA.

Narendra, K.S. & Thathatchar, M.A.L. (1989). *Learning Automata: An introduction*. Englewood Cliffs, NJ: Prentice Hall.

Narendra, K.S. & Wheeler, R.M., Jr. (1983). An *N*-player sequential stochastic game with identical payoffs. *IEEE Transactions on Systems, Man, and Cybernetics*, 13, 1154–1158.

Nilsson, N.J. (1980). *Principles of artificial intelligence*. Palo Alto, CA: Tioga.

Parker, D.B. (1985). *Learning-logic*. (Technical Report TR-47). Cambridge, MA: Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science.

Rohatgi, V.K. (1976) *An introduction to probability theory and mathematical statistics*. New York: Wiley.

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: MIT Press.

Schmidhuber, J.H. & Huber, R. (1990). Learning to generate focus trajectories for attentive vision. (Technical Report FKI-128-90). Technische Universität München, Institut für Informatik.

Sutton, R.S. (1984). *Temporal credit assignment in reinforcement learning*. Ph.D. Dissertation, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA.

Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

Thathatchar, M.A.L. & Sastry, P.S. (1985). A new approach to the design of reinforcement schemes for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15, 168–175.

Wheeler, R.M., Jr. & Narendra K.S. (1986). Decentralized learning in finite Markov chains. *IEEE Transactions on Automatic Control*, 31, 519–526.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. Ph.D. Dissertation, Cambridge University, Cambridge, England.

Werbos, P.J. (1974). *Beyond regression: new tools for prediction and analysis in the behavioral sciences*. Ph.D. Dissertation, Harvard University, Cambridge, MA.

Williams, R.J. (1986). *Reinforcement learning in connectionist networks: A mathematical analysis*. (Technical Report 8605). San Diego: University of California, Institute for Cognitive Science.

Williams, R.J. (1987a). *Reinforcement-learning connectionist systems*. (Technical Report NU-CCS-87-3). Boston, MA: Northeastern University, College of Computer Science.

Williams, R.J. (1987b). A class of gradient-estimating algorithms for reinforcement learning in neural networks. *Proceedings of the First Annual International Conference on Neural Networks*, Vol. II (pp. 601–608). San Diego, CA.

Williams, R.J. (1988a). On the use of backpropagation in associative reinforcement learning. *Proceedings of the Second Annual International Conference on Neural Networks*, Vol. I (pp. 263–270). San Diego, CA.

Williams, R.J. (1988b). *Toward a theory of reinforcement-learning connectionist systems*. (Technical Report NU-CCS-88-3). Boston, MA: Northeastern University, College of Computer Science.

Williams, R.J. & Peng, J. (1991). Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3, 241–268.

Machine Learning, 8, 257-277 (1992)

Practical Issues in Temporal Difference Learning

GERALD TESAURO
IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 USA

Abstract. This paper examines whether temporal difference methods for training connectionist networks, such as Sutton's TD(λ) algorithm, can be successfully applied to complex real-world problems. A number of important practical issues are identified and discussed from a general theoretical perspective. These practical issues are then examined in the context of a case study in which TD(λ) is applied to learning the game of backgammon from the outcome of self-play. This is apparently the first application of this algorithm to a complex non-trivial task. It is found that, with zero knowledge built in, the network is able to learn from scratch to play the entire game at a fairly strong intermediate level of performance, which is clearly better than conventional commercial programs, and which in fact surpasses comparable networks trained on a massive human expert data set. This indicates that TD learning may work better in practice than one would expect based on current theory, and it suggests that further analysis of TD methods, as well as applications in other complex domains, may be worth investigating.

Keywords. Temporal difference learning, neural networks, connectionist methods, backgammon, games, feature discovery

1. Introduction

One of the most fascinating and challenging paradigms of traditional machine learning research is the *delayed reinforcement* learning paradigm. In the simplest form of this paradigm, the learning system passively observes a temporal sequence of input states that eventually leads to a final reinforcement or reward signal (usually a scalar). The learning system's task in this case is to predict expected reward given an observation of an input state or sequence of input states. The system may also be set up so that it can generate control signals that influence the sequence of states. In this case the learning task is usually to generate the optimal control signals that will lead to maximum reinforcement.

Delayed reinforcement learning is difficult for two reasons. First, there is no explicit teacher signal that indicates the correct output at each time step. Second, the temporal delay of the reward signal implies that the learning system must solve a temporal credit assignment problem, i.e., must apportion credit and blame to each of the states and actions that resulted in the final outcome of the sequence.

Despite these difficulties, delayed reinforcement learning has attracted considerable interest for many years in the machine learning community. The notion of a learning system interacting with an environment and learning to perform a task solely from the outcome of its experience in the environment is very intellectually appealing. It could also have numerous practical applications in areas such as manufacturing process control, navigation and path planning, and trading in financial markets.

One possible approach to temporal credit assignment is to base the apportionment of credit on the difference between temporally successive predictions. Algorithms using this approach have been termed "temporal difference" methods in Sutton (1988), and have been

studied for many years in a variety of contexts. Examples include Samuel's checkers program (Samuel, 1959) and Holland's bucket brigade algorithm (Holland, 1986). An incremental real-time algorithm called TD(λ) has been proposed in Sutton (1988) for adjusting the weights in a connectionist network. It has the following form:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \tag{1}$$

where P_t is the network's output upon observation of input pattern x_t at time t, w is the vector of weights that parameterizes the network, and $\nabla_w P_k$ is the gradient of network output with respect to weights. Equation 1 basically couples a temporal difference method for temporal credit assignment with a gradient-descent method for structural credit assignment. Many supervised learning procedures use gradient-descent methods to optimize network structures; for example, the back-propagation learning procedure (Rumelhart, et al., 1986) uses gradient-descent to optimize the weights in a feed-forward multilayer perceptron. Equation 1 provides a way to adapt such supervised learning procedures to solve temporal credit assignment problems. (An interesting open question is whether more complex supervised learning procedures, such as those that dynamically add nodes or connections during training, could be adapted to do temporal credit assignment.)

It can be shown that the case $\lambda = 1$ corresponds to an explicit supervised pairing of each input pattern x_t with the final reward signal z. Similarly, the case $\lambda = 0$ corresponds to an explicit pairing of x_t with the next prediction P_{t+1}. The parameter λ provides a smooth heuristic interpolation between these two limits.

Sutton provides a number of intuitive arguments why TD(λ) should be a more efficient learning procedure than explicit supervised pairing of input states with final reward. A rigorous proof is also given that TD(0) converges to the optimal predictions for a linear network and a linearly independent set of input patterns. This proof has recently been extended to arbitrary values of λ in Dayan (1992). However, no theoretical or empirical results are available for more complex tasks requiring multilayer networks, although a related algorithm called the Adaptive Heuristic Critic (Sutton, 1984) has been successfully applied to a relatively small-scale cart-pole balancing problem (Barto, Sutton & Anderson, 1983; Anderson, 1987).

The present paper seeks to determine whether temporal difference learning procedures such as TD(λ) can be applied to complex, real-world problem domains. The paper approaches this question from two perspectives. First, section 2 identifies a number of important practical issues and discusses them in the context of the current theoretical understanding of TD(λ). Some of the issues are familiar to connectionist researchers who have been studying real-world applications of supervised learning procedures. Other issues are novel and more complex than those that arise in supervised learning. Next, section 3 examines these issues in the context of a specific application: learning backgammon strategy from the outcome of self-play. This application was selected because of its complexity and stochastic nature, and because detailed comparisons can be made with the alternative approach of supervised learning from expert examples (Tesauro, 1990). We shall see that, despite a number of potentially serious theoretical and practical problems, the TD approach works

amazingly well. With zero built-in knowledge (apart from the rules), networks are able to learn to play a fairly strong intermediate-level game. The level of performance achieved not only exceeds conventional commercial programs, but perhaps more surprisingly, it also surpasses what can be achieved by supervised training on a massive data base of human expert examples. The final section discusses the implications of these results for more general practical applications, and suggests a number of directions for further research.

2. Practical issues in TD learning

2.1. Task-dependent considerations

Learning to predict and control simultaneously: A number of important practical issues in TD learning have to do with the exact nature of the task to be learned. For example, is the task purely a prediction task, or is it a combined prediction-control task? The latter is more representative of typical real-world problems, but is also presumably more difficult. The issue of simultaneously learning to predict and learning to control was not addressed in Sutton (1988), and may lie outside the scope of the TD(λ) algorithm. It might be necessary to train a separate controller network with a different algorithm while the predictor network is learning with the TD algorithm. Alternatively, one can imagine some tasks in which the output of the predictor could be used to select the control action, for example, by choosing the state with maximum expected reward. In either case it is not known whether the combined learning system would converge at all, and if so, whether it would converge to the optimal predictor/controller. It might be possible for the system to get stuck in a self-consistent but non-optimal predictor/controller.

Stationary vs. changing tasks: Another important issue is whether the task is stationary or changes over time. Also, even for fixed tasks, it is possible that the distribution of input patterns might change over time. In either case one would want the network to continually adapt in real time to changing conditions. However, there are presumably trade-offs between accuracy of learning and ability to respond to changing conditions. Such trade-offs have been extensively analyzed for the Widrow-Hoff LMS rule (Widrow, 1976), but have not been analyzed for TD learning.

Markovian vs. non-Markovian tasks: Another important issue is whether the transitions from state to state are Markovian, i.e., depend only on the current state, or whether they also depend on the history of previous states. The analysis in Sutton (1988) is only for Markovian processes, as it is pointed out that any non-Markovian process can be included within this framework by recoding the current state information so that it also contains all relevant information from previous states. However, in practice, this may make the input space so large and complex that the learning algorithm wouldn't work very well.

Multiple outcomes: The simplest types of reinforcement tasks are characterized by a binary reward signal (e.g., a success/failure signal), but more general and more complex tasks may have many different possible outcomes. The way in which these outcomes are represented in the network may be just as important as the way in which the inputs are represented. Moreover, some of the outcomes may have a much lower likelihood of occurring than other outcomes, and one might expect that such rarely occurring outcomes would

be harder to learn. In this case, one might need special techniques analogous to those used in pattern classification tasks when some of the classes have a much lower probability of occurring than other classes.

Noisy environment: A final issue is whether the environment is noisy or deterministic. Noise may appear, for example, in the rules which govern transitions from state to state, and in the final reward signal given at the terminal states. An important consideration which we examine in more detail below is the *volatility* of the stochastic environment, i.e., the step-to-step variance in expected reward. We shall see that learning is more difficult in highly volatile environments, and that a natural way to approach learning in such environments is with a look-ahead process akin to search or planning. Noise may also enter into the representation of input patterns seen by the network. This was not addressed by Sutton, and it is not known to what extent such noise degrades the learning performance.

2.2. Algorithmic considerations

Parameter tuning: As in other connectionist learning procedures, the TD(λ) algorithm has a number of adjustable parameters that have to be heuristically tuned for a given network and task. The main parameters are the learning rate α, and of course, λ itself. Ideally, one might want not just a fixed constant value of each parameter, but a schedule for varying the parameter value as a function of learning time. For example, when training a network on a stationary task, one probably needs a schedule for reducing the learning rate analogous to the $1/t$ schedules known in the stochastic approximation literature (Robbins & Monro, 1951). Also, a schedule for λ may be useful. Setting λ to a large value early in learning might help the network get off the ground quickly, while later in learning, when the predictions become more accurate, it might be better to use smaller values of λ.

Convergence: As stated previously, convergence of TD(λ) has only been proved for linear networks and linearly independent sets of input patterns. In the more general case, the algorithm may not converge even to a locally optimal solution, let alone to a globally optimal solution.

Scaling issues: Scaling considerations are often of critical importance in successful practical applications of network learning procedures. No results are available to indicate how the speed and quality of TD learning will scale with the temporal length of sequences to be learned, the dimensionality of the input space, or the dimensionality of the network (as measured, for example, by the number of weights or by the VC dimension (Vapnik & Chervonenkis, 1971)). Intuitively it seems likely that the required training time might increase very dramatically, possibly exponentially, with the sequence length. The training time might also scale poorly with the network or input space dimension, e.g., due to increased sensitivity to noise in the teacher signal. (In contrast, with perfect teacher information, we might expect the required number of training sequences to scale roughly linearly with the network's VC dimension (Blumer, et al., 1989)). Another potential problem is that the quality of solution found by gradient-descent learning relative to the globally optimal solution might get progressively worse with increasing network size.

Overtraining and overfitting: One potential advantage of the TD approach is that, unlike most applications of supervised learning, a fixed data set is not used. Instead, training takes

place on-line using patterns generated de novo. One might hope that in this situation, performance would always improve monotonically with increasing training time, i.e., overtraining would not occur. One might also hope that one could always improve the performance of the TD nets by adding more and more hidden units to the network, i.e., overfitting would not occur.

Both overtraining and overfitting may occur, however, if the error function minimized during training does not correspond to the performance function that the user cares about. For example, the performance that one cares about for a game-playing network is not how accurately it estimates the probability of winning in a given position, but rather its ability to select good moves. It may be the case that the network could produce fairly accurate predictions but not select very good moves. One would especially expect this to be true for games in which the best move is only slightly better than other alternatives. On the other hand, if the network has large errors in the absolute accuracy of its predictions, it could still be able to select good moves. This is because, as discussed in Christensen and Korf (1986), and Utgoff and Clouse (1991), a heuristic evaluation function need not exactly represent the true values of states for correct move selection. Instead, it only needs to have the correct sign for the slope between pairs of points in order to make correct state preferences. There may be many such functions, with widely varying degrees of prediction accuracy. A simple example illustrating this is shown in table 1. Overtraining and overfitting might also occur if the distribution of input training and testing patterns are different. For example, the game-playing network might be trained on its own play, but have to perform against other players. The differing styles of play would lead to different distributions of input patterns.

Incremental learning: A nice feature of TD(λ) is that the weights can be updated in a fully incremental fashion. It is not necessary to wait until the end of the sequence to compute the weight change at a given time step. However, this may not be strictly necessary in practical implementations. Modern workstations have enough memory to store the entire sequence of input and output patterns, even for fairly large problems, as long as the sequences terminate in a reasonable amount of time. If the sequence is so long that it cannot be stored, the algorithm may not be able to learn the task in any case. Thus one could imagine other algorithms that give improved performance at the price of sacrificing the fully incremental nature of TD(λ).

Table 1. An example showing the difference between a good predictor and a good move selector.

Move	True Prob.	Network #1	Network #2
A	0.846	0.842	0.719
B	0.840	0.843	0.621

The network must choose either move A or move B. The true probabilities of winning are 0.846 and 0.840 respectively. Network #1 makes highly accurate estimates of the probabilities of winning, but selects the wrong move. Network #2's estimates have large errors, but it picks the right move.

2.3. *Representational issues*

The way in which the input and output data are represented in multilayer connectionist networks has been found to be one of the most important factors in successful practical applications of supervised learning procedures. Such representational issues are likely to be equally important in practical applications of temporal difference learning. It is useful to distinguish between two basic kinds of representations: (a) *lookup table* representations, in which the network has enough adjustable parameters to explicitly store the correct output for every possible state in the input state space; and (b) *compact* representations, in which the number of adjustable parameters is much less than the number of states in the state space, and the network therefore has to capture the underlying regularity of the task.

In Sutton (1988), the TD(λ) algorithm is discussed only in terms of lookup table representations. However, the way in which TD learning works for lookup table representations is likely to be completely different from the way it works for compact representations. With lookup table representations, it is clear that the network has no way to estimate the predicted outcome of a particular state unless it has actually observed the state. Thus in order for the network to learn the entire function, it has to observe every possible state in the state space. In fact, Sutton's convergence theorem requires every possible state to be visited infinitely many times in order to guarantee convergence. This will clearly be intractable for real-world problems. Even if the state space is discrete, the number of possible states is likely to be so large that there is neither sufficient storage capacity to store the lookup table, not sufficient time to visit all possible states.

On the other hand, with compact representations, it might be possible to learn complex tasks in high-dimensional spaces reasonably well. After observing an infinitesimal fraction of all possible states, the network might be able to find a solution that generalizes acceptably for states not seen during training. Thus we can see that the ability of compact networks to generalize provides an ability to tackle otherwise intractable problems. However, there are also a number of limitations to the generalization capability of compact networks. For example, if the task is complex, a network with a limited number of hidden units might not have enough structural complexity to exactly represent the task. Also, the gradient-descent method of assigning structural credit within the network can only find local optima, not global optima. Such factors will limit the effectiveness of TD learning in complex domains.

2.4. *Volatility limit*

Let us examine equation 1 in more detail. Note that for any given input state x_t, there is a true expected outcome θ_t associated with that state, and P_t is the network's estimate of θ_t. If we had access to the values of θ_t we could use them to do back-propagation learning on the input-output pairs (x_t, θ_t). But in TD learning θ_t is not available. Instead, note that the next prediction P_{t+1} is used in a role analogous to θ_t in back-propagation. The network output P_t is being driven toward P_{t+1}, and no learning takes place when $P_t = P_{t+1}$. It should be intuitively clear that in TD learning, P_{t+1} is being used as a heuristic stochastic estimator of the true expected outcome θ_t. It should also be clear that the learning algorithm will only make progress when P_{t+1} is a more accurate stochastic estimator of θ_t than P_t is.

There are a number of reasons why P_{t+1} might not be a more accurate stochastic estimator of θ_t than P_t is. One possibility comes from the fact that the network has to generalize in any practical problem, as discussed previously. The network's generalizations for states near the end of the sequence may well be less accurate than for states far from the end.

Another reason why P_{t+1} may fail to be a more accurate estimator of θ_t is due to volatility in the environment. As learning progresses, P_t approaches θ_t, while P_{t+1} approaches θ_{t+1}. Now in highly volatile environments, θ_{t+1} may vary significantly from θ_t. It is true that the average value over all possible states that can evolve from x_t is given by $\langle \theta_{t+1} \rangle = \theta_t$, but we must remember that the network does not get to see infinitely many evolutions from x_t. In a typical complex problem, the network will only see x_t once, and thus only one state following x_t. If the value of x_{t+1} has little to do with the value of x_t due to high volatility, then P_{t+1} may be a poor estimator of θ_t. Furthermore in some problems it could be the case that volatility increases with time, so that states near the end of the sequence are more volatile than earlier states. This would provide a further limitation on the ability of P_{t+1} to estimate θ_t more accurately than P_t.

A natural way to try to overcome limitations due to volatility of the environment is to allow multiple evolutions from a given state, i.e., for each observed x_t, reset the state to x_t many times and let the stochastic process generate a distribution of successor states x_{t+1} with average expected reward $\langle P_{t+1} \rangle$. The error signal at each time step would then be proportional to $\langle P_{t+1} \rangle - P_t$. The multiple samples would provide an estimate of $\langle \theta_{t+1} \rangle$, which, as stated previously, should equal θ_t. If the temporal evolution at each step is governed by a stochastic process with a small number of possible outcomes (such as flipping a coin or rolling dice), one could explicitly compute an average over all possible outcomes.

3. A case study: TD learning of backgammon strategy

We have seen that current theory provides little indication of how TD(λ) will work in practice. In the absence of theoretical guidance, we now empirically examine the previously discussed practical issues within the context of a specific application: learning to play the game of backgammon from the outcome of self-play. Complex board games such as checkers, chess, Othello and backgammon have been widely studied as testing grounds for various machine learning procedures (Samuel, 1959; Samuel, 1967; Griffith, 1974; Quinlan, 1983; Mitchell, 1984; Frey, 1986; Christensen & Korf, 1986; Lee & Mahajan, 1988; Tesauro & Sejnowski, 1989; Tesauro, 1990). Several of these studies have employed temporal difference learning methods.

Unlike checkers, chess, and Othello, backgammon is a nondeterministic game in which the players take turns rolling dice and moving their pieces around the board as allowed by the dice roll. The first player to move all of his pieces around and off the board wins the game. The game is complicated because it is possible to "hit" opponent pieces and send them to the far end of the board, and to form blocking configurations that impede the forward movement of opponent pieces. These facts lead to a number of subtle and complex strategies and tactics at the highest levels of play (Magriel, 1976).

Backgammon offers a number of attractive features as a test vehicle for TD learning approaches. Due to the stochastic dice rolls, the evolution of board states during the course

of a backgammon game can be characterized as an absorbing Markov process, in which the initial state is always a unique starting configuration, and in which the subsequent transitions proceed randomly, depending only on current state information, until a well-defined terminal state is reached, characterized by one side having all its pieces off the board. This is precisely the type of situation for which TD(λ) was designed. Playing the game at expert level involves considerable complexity, but it has been demonstrated (Berliner, 1979; Tesauro & Sejnowski, 1989) that much of this complexity can be captured in a static evaluation function and does not require deep look-ahead searches. This means that a feed-forward neural network could learn a static evaluation function that would play well without search, and that the quality of learning can be directly assessed by measuring game performance. (In contrast, the performance of a search-based program depends on both the quality of the evaluation function and the power of the search procedure.) Finally, it is possible to make a detailed comparison of TD learning with the alternative approach of supervised learning from expert examples (Tesauro, 1990). This is important for general practical applications, because in order for TD learning to be successful in the real world, it not only has to work well on hard problems, but it also has to be competitive with other approaches such as supervised learning.

It seems reasonable that, by watching two fixed opponents play out a large number of games against each other, a network could learn by TD methods to predict the expected outcome of any given board position. In addition to estimating expected outcome, such a network could also be used for move selection by generating all legal moves in a given position and picking the move with the maximum expected outcome. A more interesting learning question, however, is whether the network could learn from the outcome of its own play. As the network learns, its control strategy changes, and thus the distribution of input patterns and final rewards would also change. This is the type of learning that will be examined in this section, even though it is not clear a priori that such a learning system would converge to a sensible solution.

3.1. Set-up of the learning system

The TD(λ) algorithm can be applied to backgammon in the following straightforward way: a network is set up to observe a sequence of board positions $x_1, x_2, \ldots, x_f$, resulting in a final reward signal z. In the simplest case the reward signal is $z = 1$ if White wins and $z = 0$ if Black wins. In this case the network's output P_t is an estimate of White's probability of winning from board position x_t. The sequence of board positions is generated by setting up an initial configuration, and making plays for both sides using the network's output as an evaluation function. In other words, the move selected at each time step is the move that maximizes P_t when White is to play and minimizes P_t when Black is to play.

A critical factor in the overall performance of the learning system is the representation scheme used to encode the input board description. It is well known in computer games research that significantly higher levels of performance can be achieved if the board state is described using "features" relevant to good play, as opposed to a "raw" board description. In the experiments reported here, however, the input encoding schemes only contained simple encodings of the raw board information (the number of White or Black men at each location) and did not utilize any additional pre-computed features.

Since the input encoding scheme contains no built-in knowledge about useful features, and since the network only observes its own play, we may say that this is a "knowledge-free" approach to learning backgammon. Such an approach is interesting because it is not clear that it can make any progress at all beyond its starting state, which for a network with random initial weights is a random move selector. The zero-knowledge approach also provides an important baseline for judging other approaches using various forms of built-in knowledge.

The approach described above is similar in spirit to Samuel's approach to learning checkers from self-play, but in several ways it is a more challenging learning task. One important difference is that Samuel's board description was in terms of a number of hand-crafted features, several of which were designed in consultations with human checkers experts. However, the networks studied here use only a raw board description and had no knowledge built into the input features. The evaluation function learned in Samuel's study was a linear function of the input variables, whereas multilayer networks learn more complex nonlinear functions. Also, the final reward signal in backgammon is noisy due to the dice rolls; this presumably makes the learning task more difficult than in noise-free games such as checkers. The branching ratios in backgammon are so large that look-ahead methods cannot be employed, whereas Samuel used search both in move selection and in calculation of the learning algorithm's error signal. Finally, Samuel found that it was necessary to give the learning system at least one fixed intermediate goal, material advantage, as well as the ultimate goal of the game. The proposed backgammon learning system has no such intermediate goals.

3.2. Learning disengaged bearoff strategy

Like many other games, the full complexity of backgammon is greatly simplified in certain special situations. For example, finding good moves in racing situations, in which hitting and blocking are not possible, is considerably easier than in fully engaged positions, in which hitting and blocking are possible. The first TD experiments we shall examine are designed to learn the case of disengaged bearoff positions, in which both sides have all of their men in the their home quadrant and can remove them from the board. This is an exceedingly simple situation because there is a simple yet strong heuristic for correct play: always select the move that takes the maximum number of pieces off the board. This principle is only violated in certain rare situations. If the principle does not uniquely determine a move, a secondary consideration is to distribute the remaining men as smoothly as possible; this usually determines the best move. (An interesting exception to this rule is discussed in Berliner (1977).)

Another advantage of studying this situation is that it can be solved essentially exactly by conventional algorithms (Berliner, 1977; Zadeh, 1977). For each of the approximately 54,000 possible bearoff configurations for a given side, one can recursively compute and store in a table the exact probability of removing all men on one roll, two rolls, three rolls, etc.. One can then use this information to compute the exact probability that either side will win. Thus, in addition to comparisons with supervised learning of expert examples, one can also compare the results of TD learning with the exact optimal move choices and probabilities of winning.

The networks trained on this task had 16 units to encode the board description: 6 units to encode the number of Black men at locations 1–6, 6 units to encode the number of White men at locations 19–24, 2 units to encode the number of White and Black men off the board, and 2 units to encode the side to move. The networks had a feed-forward structure with full connectivity from one layer to the next. Two architectures were examined: a single-layer architecture with direct input-output connections and no hidden units, and a multi-layer architecture containing a single hidden layer with varying numbers of hidden units. The single-layer architecture can only represent linearly separable functions (Minsky & Papert,1969), while the multilayer architecture, given sufficient hidden units, is capable of universal function approximation (Hornik, Stinchcombe & White, 1989). Both the hidden units and the output unit utilized a sigmoidal transfer function $y = 1/(1 + e^{-x})$.

The initial board configurations were generated randomly by distributing all 15 men for each side with uniform probability in the six home board locations. With this distribution of initial positions, the average sequence length is about 14 time steps with good play on both sides.

As with back-propagation, a certain amount of parameter tuning was required to get good results with the TD(λ) algorithm. It was found that a learning rate of $\alpha = 0.1$ usually gave good results. Lower learning rates did not improve the maximum performance (although they did reduce the level of stochastic fluctuations in performance), whereas significantly higher learning rates did degrade performance. Also, the value of λ appeared to have almost no effect on the maximum obtainable performance, although there was a speed advantage to using large values of λ. The results reported here used a value of λ set (somewhat arbitrarily) at 0.7. The initial random weight scale also did not appear to be important; in the results reported here, weights were initialized to uniform random values in the range $[-0.5, +0.5]$.

Two measures of performance were monitored to assess the results of learning. The absolute prediction error was monitored by comparing the network's outputs for the positions seen during training with the exact win probabilities computed from the lookup tables. (One should keep in mind that these exact probabilities were not used as part of the training signal.) The network's move selection ability was also monitored during training by measuring the fraction of time the network selected the theoretically optimal move in a fixed set of test positions. (There were 210 test positions, taken from a set of games in which the author played both sides. This test set was pruned from a larger set of over 300 positions by deleting all positions with no unique best move, according to the lookup tables. In most of the deleted positions, the probability of winning was either exactly 1 or exactly 0 regardless of which move was selected.)

A perhaps surprising discovery was that, apart from stochastic fluctuations controlled by the value of α, the absolute prediction error always decreased during training. On the other hand, the move-selection performance usually reached a maximum after several tens of thousands of games, and then decreased thereafter. (Due to the stochastic training procedure and the relatively flat nature of the learning curves, it is difficult to say precisely when the maximum performance was reached.) As stated previously, this "overtraining" could be due to the differing distributions of training and test positions, or it could also be due to the difference between the function minimized by the learning algorithm (prediction error) and the function that the user cares about (move-selection ability).

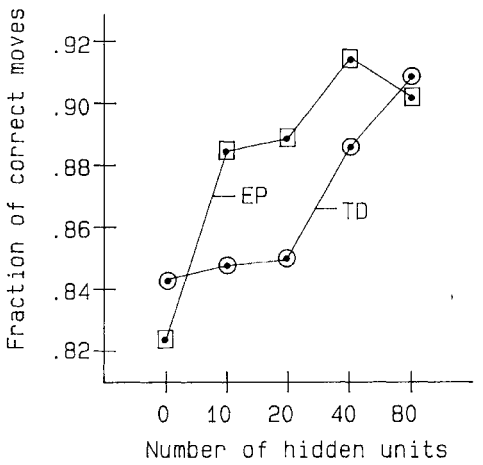


Figure 1. Plot of move selection performance vs. number of hidden units for networks trained on disengaged bearoff positions using TD learning from self-play (TD), and supervised training on human expert preferences (EP). Both learning systems used identical 16-unit coding schemes.

The level of move-selection performance obtained by the networks was also surprising in view of the average absolute prediction errors. Typically the networks are only able to estimate the probability of winning to within about 10% of the true probability. On the other hand, one usually needs to make discriminations at the 1% level or lower for accurate move selection. Thus based on this consideration alone, one would expect the networks to be rather poor move selectors. However, figure 1, which plots move-selection performance as a function of the number of hidden units for TD-trained networks, shows that performance is in fact quite good. Furthermore, the networks with hidden units clearly do better than networks without hidden units. This indicates that the networks have absorbed some of the nonlinear aspects of the problem, and are not just implementing a linear rule such as "maximize the number of pieces taken off the board." It is also reassuring that this nonlinear structure can be learned in the TD procedure, despite the lack of any theoretical guarantees.

Also plotted for comparison are the results of supervised training of identical networks with identical coding schemes on a data set of about 1700 training positions. In each training position, a human expert (the author) recorded a preference of best move. The training methodology is the "comparison paradigm" described in Tesauro (1989), in which the network is trained by back-propagation to score the expert's move higher than each of the other legal moves.

We can see that, as is usual with back-prop nets trained on a fixed data set, the ability to increase performance by adding more hidden units is eventually lost for sufficiently large nets: at this point, the networks start to overfit the training data. On the other hand, since the TD nets are not trained on a fixed data set, the performance can in principle always be improved by adding more hidden units. We do in fact find that, at least for the network

sizes studied here, performance increases monotonically with the number of hidden units, and that the TD nets eventually catch up to the performance level of the nets trained on expert preferences (denoted EP in figure 1).

One should note that the absolute prediction error of the TD nets generally gets worse as the end of the game approaches. This is not surprising, because states far from the end can be accurately represented by a simple pip count approximation, but this breaks down when there are only a few pieces left. Also for this particular task, states near the end of the game have higher volatility than states far from the end. As stated previously, these factors may provide important limits to the ability of TD learning to approach theoretically optimal performance.

3.3. Learning full-board racing strategy

The results of TD training of disengaged bearoff situations were encouraging enough to attempt the next logical extension: training of general racing situations covering the entire board. These situations require many more units to encode the board description, and they are usually more complicated because of the possibility of bonus wins called "gammons." These occur when one side removes all its pieces before the other side takes off any pieces, and the point value awarded is double the normal value. (There is also the possibility of winning a triple-value "backgammon," but this will not be considered here.) Thus in a general racing situation, if one has men in the outer quadrants, one has to choose between playing for a win and avoiding the loss of a gammon. Strategies in these two cases are usually different. For example, in gammon avoidance situations it is usually correct to bear men into the 6 point, whereas if there is no gammon risk, one should distribute one's men evenly on the 6, 5 and 4 points.

The possibility of winning a gammon means that a game can now end in one of four possible outcomes. A straightforward way to handle this is to use a four-component reward signal, with each component corresponding to one of the four possible outcomes. One would then train a network with four output units, which would learn to estimate for any input position the probabilities of the four separate outcomes. (Without additional constraints, however, there would be no guarantee that the network's estimated probabilities would sum to 1.) For move-making decisions, one could use the network's estimated probabilities to estimate expected payoff, and select the move that maximizes expected payoff. Specifically, if (p_1, p_2, p_3, p_4) are the network's estimates of the probabilities of (W wins, W gammons, B wins, B gammons), then the expected payoff to White is given by $p_1 + 2p_2 - p_3 - 2p_4$. Thus even in this more complex situation, one can still cast the problem as a prediction learning problem only, and can use the predictor network's outputs to generate control decisions without training a separate control network.

For TD training of general race situations, a 52-unit coding scheme was used: 24 units each to encode the number of White or Black men at locations 1–24, 2 units to encode White and Black men off, and 2 units to encode the player to move. A modified four-component reward signal was used in which the units representing regular wins were activated if a side had either a regular or a gammon win. This should be easier to learn for a network with sigmoidal units, since a unit that only represents regular wins has to learn a

non-monotonic function of the lead in the race. The initial board configurations were generated by randomly picking a divider location on the board, and randomly distributing all White men uniformly to one side and all Black men uniformly to the other side of the divider location. With this distribution of starting positions, the average number of time steps from start to finish was about 20, as compared to 14 in the bearoff experiments. As in the previous section, the algorithm parameters were set at $\alpha = 0.1$ and $\lambda = 0.7$.

In this more complex situation, algorithms are no longer available for computing the exact outcome probabilities. Hence the only performance measure used was the fraction of time the network agreed with a human expert (the author) on the best move in a fixed set of 248 full-board racing test positions. This test set measure is less reliable than in the previous section, because there may be significant human errors in the choice of best move. There may also be a substantial fraction of positions in which the outcome is uniquely determined regardless of which move is made.

Results of TD training are shown in figure 2, which plots maximum test set performance as a function of number of hidden units. Once again these results are compared with networks using an identical coding scheme trained on a data set of about 3100 human expert examples. Once again the TD networks reached peak performance after several tens of thousands of games. For this task, no clear evidence of overtraining was seen; thus with further training it might be possible to obtain higher levels of performance.

We can see that, despite the increased size of the input space, complexity of the task, and length of the sequence, the TD networks still were able to achieve a high level of performance. The test set performance is almost as good as the EP nets. In fact, since the test set measure is admittedly biased in favor of the expert-trained nets (which could pick

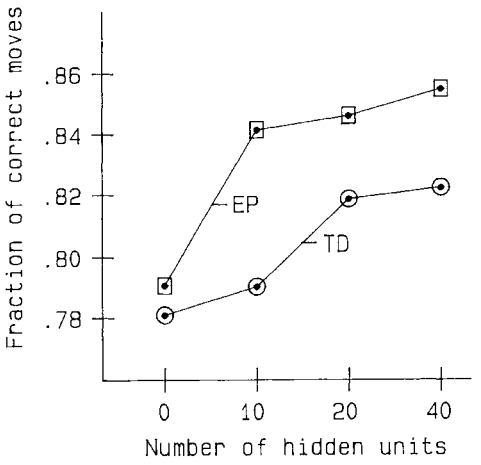


Figure 2. Plot of move selection performance vs. number of hidden units for networks trained on full-board racing positions using TD learning from self-play (TD), and supervised training on human expert preferences (EP). Both learning systems used identical 52-unit coding schemes.

up consistently wrong or arbitrary stylistic preferences of the expert), one may legitimately wonder whether the TD net really is inferior in an absolute sense, and if so, by how much. Also, further improvements in TD performance might be obtainable by adding more hidden units to the TD networks, by training longer, or by using a more representative distribution of starting positions.

Qualitatively, the TD nets appear to have discovered many important ingredients of general racing strategies. The main apparent defect is a tendency to waste pips when the network has both a small chance of winning and a small chance of losing a gammon. In this case a human would try to remove all doubt about saving the gammon, whereas the network tends to make plays that suggest trying to win.

3.4. Learning to play the entire game

We now consider extending TD learning to cover more general engaged situations, in which hitting and blocking are possible. These situations are much more complex than racing situations, and the sequences can be much longer. (In typical human play the average sequence length from start to finish is around 50–60 time steps, whereas for random move selectors, games can last several hundred or even thousands of time steps.) Also the optimal strategy for one side depends more critically on what strategy the opponent is using, whereas in racing situations the choice of best move is largely independent of the opponent's strategy. This means that a network adapting to its own play has a more serious possibility of getting stuck in self-consistent but non-optimal strategies.

The networks trained on this task used an expanded scheme to encode the local information. Rather than a single unit to encode the number of men of a given color at a given location, a truncated unary encoding with four units was used. The first three units encoded separately the cases of one man, two men, and three men, while the fourth unit encoded the number of men beyond 3. (In the development of Neurogammon, it was found that truncating at 5 or 6 units rather than 4 units gives better performance but of course takes longer to simulate.) This coding scheme thus used 96 units for each side to encode the information at locations 1–24, and an additional 6 units to encode the number of men on the bar, off the board, and the player to move, for a total of 198 input units.

The parameter settings in these experiments were once again $\alpha = 0.1$ and $\lambda = 0.7$. A few experiments with smaller and larger values of λ seemed to indicate that larger values would decrease the performance, while smaller values would give about the same performance.

Networks were trained on the entire game, starting from the opening position and going all the way to the end. This is an admittedly naive approach, and given the complexity of this task, one might wonder whether it would actually work. Alternatively, one can consider dividing the game into a number of phases (e.g., early engaged, middle engaged, late engaged, and race), and using as a heuristic reward for a given phase the network output for the next phase. Some preliminary experiments were performed with a two-phase approach in which the network was trained up to the point of disengagement, and the output of a previously trained racing net (described in the previous section) was used as a heuristic reward. In this way the network would see a shorter sequence of positions, and its evaluation function would be simpler than the evaluation function needed for the entire

game. On the other hand, there may be problems due to the fact that the racing network was trained with a somewhat arbitrary distribution of starting positions. Furthermore, any systematic biases in the racing network's judgement might be transferred to the engaged network. Empirically this two-phase approach seemed to offer some potential for improving the performance of smaller nets, but not for larger nets, and view of the above-mentioned theoretical concerns, this approach was eventually abandoned.

Training a single net on the entire game was not expected to yield any useful results other than a reference point for judging more sensible approaches. However, the rather surprising result was that a significant amount of learning actually took place. Performance on the 248-position racing test set reached about 65%. (This is substantially worse than the racing specialists described in the previous section.) Performance on a separate set of 500 full-contact test positions is plotted in figure 3. Again these figures are compared with results of supervised training on a human expert training set containing over 15,000 engaged positions. We can see that the TD nets reached levels of performance well beyond the initial random networks, showing that substantial learning took place. (The random initial networks only select the right move about 5% of the time.) The performance levels lag somewhat behind what can be achieved with expert preference training, but we must remember that this test set measure may not give the most accurate assessment of true game-playing strength.

A more accurate and objective measure of game-playing strength is actual game performance against an opponent. Both the TD nets and the EP nets have been tested in actual game play against Sun Microsystem's Gammontool program. Gammontool is representative of the level of performance that is typically achieved with conventional commercial programs,

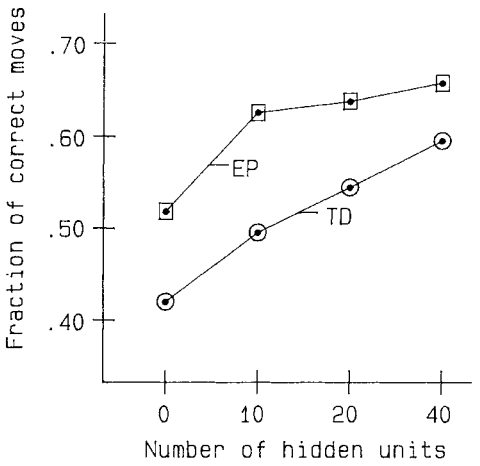


Figure 3. Plot of move selection performance in general engaged positions vs. number of hidden units for networks trained using TD learning from self-play (TD), and supervised training on human expert preferences (EP). Both learning systems used identical 198-unit coding schemes.

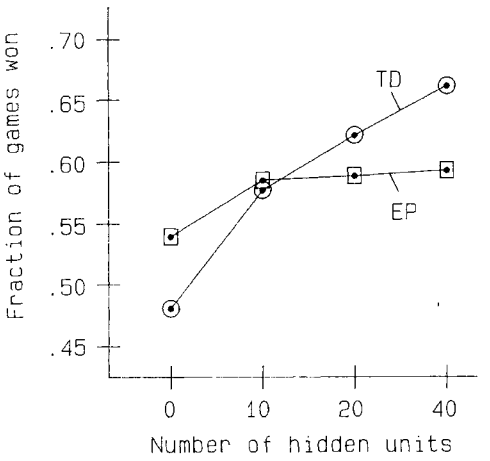


Figure 4. Plot of game performance against Gammontool vs. number of hidden units for networks trained using TD learning from self-play (TD), and supervised training on human expert preferences (EP). Each data point represents the result of a 10,000 game test, and should be accurate to within one percentage point.

and provides a decent benchmark for measuring game-playing strength. Human beginners can win about 40% of the time against it, decent intermediate-level humans would win about 60%, and human experts would win about 75%. (The random initial networks before training win only about 1%.) Since the EP nets are trained on engaged positions only, the testing procedure is to play out the game with the network until it becomes a race, and then use Gammontool's algorithm to move for both sides until the end. This also does not penalize the TD net for having learned rather poorly the racing phase of the game. Results are plotted in figure 4.

Given the complexity of the task, size of input space and length of typical sequences, it seems remarkable that the TD nets can learn on their own to play at a level substantially better than Gammontool. Perhaps even more remarkable is that the TD nets surpass the EP nets trained on a massive human expert data base: the best TD net won 66.2% against Gammontool, whereas the best EP net could only manage 59.4%. This was confirmed in a head-to-head test in which the best TD net played 10,000 games against the best EP net. The result was 55% to 45% in favor of the TD net. This confirms that the Gammontool benchmark gives a reasonably accurate measure of relative game-playing strength, and that the TD net really is better than the EP net. In fact, the TD net with no features appears to be as good as Neurogammon 1.0, backgammon champion of the 1989 Computer Olympiad, which does have features, and which wins 65% against Gammontool. A 10,000 game test of the best TD net against Neurogammon 1.0 yielded statistical equality: 50% for the TD net and 50% for Neurogammon.

The TD net's performance against these three opponents indicates that it has reached a significant level of playing ability. This violates a widely held belief in computer games and machine learning research that significant levels of performance in game-playing programs

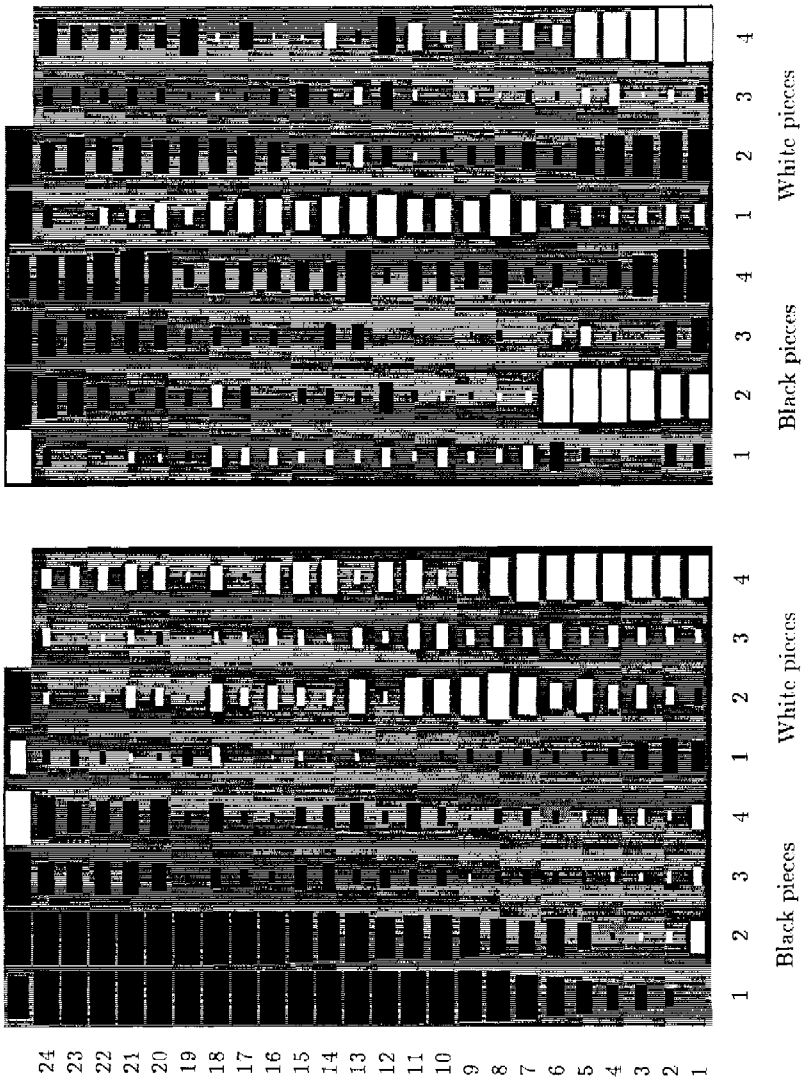
can only be achieved through the use of hand-crafted features in the evaluation function. Apparently the hidden units in the TD net have discovered useful features through self-play. When one looks at the pattern of weights learned by the TD net, one can see a great deal of spatially organized structure, and some of this structure can be interpreted as useful features by a knowledgeable backgammon player. Figure 5 shows the weights from the input layer to two of the hidden units in the best TD net. Both hidden units contribute positively to the estimation of Black's chances of winning and gammoning, and negatively to the estimation of White's chances of winning and gammoning. The first hidden unit appears to be a race-oriented feature detector, while the second hidden unit appears to be an attack-oriented feature detector.

The training times needed to reach the levels of performance shown in figure 4 were on the order of 50,000 training games for the networks with 0 and 10 hidden units, 100,000 games for the 20-hidden unit net, and 200,000 games for the 40-hidden unit net. Since the number of training games appears to scale roughly linearly with the number of weights in the network, and the CPU simulation time per game on a serial computer also scales linearly with the number of weights, the total CPU time thus scales quadratically with the number of weights: on an IBM RS/6000 workstation, the smallest network was trained in several hours, while the largest net required two weeks of simulation time.

The networks did not appear to overtrain, but it is not clear whether the performance increases very slowly with further training or stays flat. Since the networks in the previous sections also required several tens of thousands of training games, this suggests that the number of training sequences needed to train a TD network may not scale badly with the task complexity or average sequence length. Instead, the training time may just depend on the number of bits of information needed to specify a trained network, and on how many bits of information are received per game. Since the outcome of each game gives one bit of information (two bits including gammons), and since the networks have several thousand weights that probably must be specified to at least 4–8 bits of accuracy, this suggests a training time on the order of tens of thousands of games.

Some qualitative observations on the styles of play learned by the TD and EP nets are worth noting. The TD nets have developed a style emphasizing running and tactical play. For example, it prefers to immediately split its back men rather than bringing down builders or slotting home board points. It is good in running game situations and in tactical situations such as blot-hitting contests and blitz attacks. The EP nets, however, favor a more quiescent positional style of play emphasizing blocking rather than racing. This is more in line with human expert play (at least the particular human expert who made the training data), but it often leads to complex prime vs. prime and back-game situations that are hard for the network to evaluate properly. This suggests one possible advantage of the TD approach over the EP approach: by learning to imitate an expert teacher, the learning system may get itself into situations that it doesn't know how to handle. With the alternative approach of learning from experience, the learner may not reproduce the expert strategies, but at least it will learn to handle whatever situations are brought about by its own strategy.

It's also interesting that TD net ended up playing well in the early engaged phase of play, whereas its play in the late racing phase is rather poor. This is contrary to the intuitive notion that in temporal credit assignment learning, states far from the end of the sequence will be harder to learn than states near the end. Apparently the inductive bias due to the


24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 1 2 3 4
Black pieces
White pieces
1 2 3 4 1 2 3 4
Black pieces
White pieces

representation scheme and network architecture is more important than temporal distance to the final outcome. This may partially explain why training times were not dramatically worse in the full-game situation than in the simplified endgame situations.

4. Conclusions

We have seen that, from the theoretical point of view, there may be a number of important limitations to the effectiveness of TD learning in large-scale complex domains. The algorithm may not converge even for prediction only tasks, let alone for combined prediction/control tasks. Even if the algorithm does converge, it may get stuck in poor locally optimal solutions. Finally, even if the algorithm is able to find good solutions, the scaling of required training time with problem size or sequence length may be so poor that the learning task would be effectively intractable.

In view of these potential difficulties, there are a number of very encouraging conclusions that can be drawn from the backgammon application. Empirically the algorithm always converges to at least a local minimum. The quality of solution found by the network is usually fairly good, and generally improves with increasing numbers of hidden units. Furthermore, the scaling of training time with the length of input sequences, and with the size and complexity of the task, does not appear to be a serious problem. Finally, it is encouraging that the network was able to learn to make good control decisions as well as learn to estimate expected outcome. In fact, the network's ability to select good moves is much better than we have a right to expect, because its absolute accuracy in estimating expected outcome is usually at the 10% level, whereas the difference in expected outcome between optimal and non-optimal moves is usually at the level of 1% or less. This suggests that much of the network's absolute prediction error is systematic error that applies equally to all moves generated from the same initial position, and thus cancels out in move-making decisions.

The most encouraging finding, however, is a clear demonstration that in the full-game situation, the TD learning network with zero built-in knowledge can achieve a higher level of overall game performance than an identical network trained on a massive data base of human expert examples. The level of performance achieved is in fact equal to a program that convincingly won the backgammon championship in a major tournament for computer programs. This level of performance is so far beyond what one would have expected beforehand that it seems worthwhile to devote some further effort to understanding exactly how

Figure 5. A diagram illustrating the weights from the input units to two of the 40 hidden units in the best TD net. Black squares represent negative weights and white squares represent positive weights; the size of the square indicates the magnitude of the weights. The rows represent from bottom to top the spatial locations 1–24. The top row represents: (W barmen, B barmen, W men off, B men off, W turn, B turn). The columns represent the number of Black and White men as indicated. The first hidden unit has two noteworthy features: a linearly increasing pattern of negative weights for Black blots and Black points, and a negative weighting of White men off and a positive weighting of Black men off. These features are recognizable as contributing to an estimate of Black's probability of winning based on his lead in the race. The second hidden unit has the following noteworthy features: strong positive weights for Black home board points, strong positive weights for White men on bar, positive weights for White blots, and negative weights for White points in Black's home board. The experienced backgammon player recognizes that these factors all contribute to the probability of a successful Black attacking strategy.

this is possible. It may also be worthwhile trying to apply the combination of TD with back-propagation, or TD with other supervised learning algorithms, to other difficult temporal credit assignment problems.

Future prospects for the backgammon application look very promising. It certainly seems possible that further improvements in performance can be obtained merely by adding more hidden units to the network and training for longer training times, although the quadratic scaling of CPU time with the number of weights may limit how far this can be carried in practice. Better results might also be obtainable by optimizing the parameter settings, or by modifications of the TD training procedure. For example, the next prediction could be replaced by an average over all possible dice rolls; this could reduce limitations due to volatility. Also, partitioning the game into a number of temporal phases and training separate specialist networks on each phase may make it easier for each specialist network to learn the evaluation function appropriate for that particular phase. In actual game play, the outputs of the specialists could be combined using smoothly-varying application coefficients, as suggested in Berliner (1979). Finally, an improved representation scheme, which uses features known to be useful in backgammon evaluation functions, and which is better able to represent the value of near-end states, might give substantially better results. Such improved representation schemes are currently under investigation. As this article goes to press, a TD net containing all of Neurogammon's features is learning from self-play. The network has reached 71% against Gammontool and continues to improve slowly with further training. It appears to be the strongest program ever seen by this author.

Beyond this specific application, however, the larger and more important issue is whether learning from experience can be useful and practical for more general complex problems. Certainly the quality of results obtained in this study suggests that the approach may work well in practice, and may work better than we have a right to expect theoretically. There may also be some intrinsic advantages to this approach over supervised training on a fixed set of labeled exemplars. At the very least, for tasks in which the exemplars are hand-labeled by humans, it eliminates the laborious and time-consuming process of labeling the data. Furthermore, the learning system would not be fundamentally limited by the quantity of labeled data, or by any intrinsic errors in whatever process is used to label the data. Also, the learning system might be able to avoid getting itself into situations that it doesn't know how to handle, as can happen when one is learning by imitating an expert. Finally, preserving the intrinsic temporal nature of the task, and informing the system of the consequences of its actions, may convey important information about the task that is not necessarily contained in the labeled exemplars. More theoretical and empirical work will be needed to establish the relative advantages and disadvantages of the two approaches. A result of this may be the development of novel learning paradigms combining supervised learning with learning from outcome; in combination it might be possible to surpass what either approach could achieve individually. Preliminary work supporting this hypothesis is reported in Utgoff and Clouse (1991).

Acknowledgments

The author thanks Scott Kirkpatrick and Rich Sutton for helpful comments on an earlier version of the manuscript.

References

Anderson, C.W. (1987). Strategy learning with multilayer connectionist representations. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 103–114).

Barto, A.G., Sutton, R.S., & Anderson, C.W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics, 13*, 835–846.

Berliner, H. (1977). Experiences in evaluation with BKG—a program that plays backgammon. *Proceedings of IJCAI* (pp. 428–433).

Berliner, H. (1979). On the construction of evaluation functions for large domains. *Proceedings of IJCAI* (pp. 53–55).

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *JACM, 36*, 929–965.

Christensen, J. & Korf, R. (1986). A unified theory of heuristic evaluation functions and its application to learning. *Proceeding of AAAI-86* (pp. 148–152).

Dayan, P. (1992). The convergence of TD(λ). *Machine Learning, 8*, 341–362.

Frey, P.W. (1986). Algorithmic strategies for improving the performance of game playing programs. In: D. Farmer, et al. (Eds.), *Evolution, games and learning*. Amsterdam: North Holland.

Griffith, A.K. (1974). A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence, 5*, 137–148.

Holland, J.H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In: R.S. Michalski, J.G. Carbonell & T.M. Mitchell, (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks, 2*, 359–366.

Lee, K.-F. & Majahan, S. (1988). A pattern classification approach to evaluation function learning. *Artificial Intelligence, 36*, 1–25.

Magriel, P. (1976). *Backgammon*. New York: Times Books.

Minsky, M.L. & Papert, S.A. (1969). *Perceptrons*. Cambridge, MA: MIT Press. (Republished as an expanded edition in 1988).

Mitchell, D.H. (1984). Using features to evaluate positions in experts' and novices' Othello games. Master's Thesis, Northwestern Univ., Evanston, IL.

Quinlan, J.R. (1983). Learning efficient classification procedures and their application to chess end games. In: R.S. Michalski, J.G. Carbonell & T.M. Mitchell (Eds.), *Machine learning*. Palo Alto, CA: Tioga.

Robbins, H. & Monro, S. (1951). A stochastic approximation method. *Annals of Mathematical Statistics, 22*, 400–407.

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning internal representations by error propagation. In: D. Rumelhart & J. McClelland, (Eds.), *Parallel distributed processing*. Vol. 1. Cambridge, MA: MIT Press.

Samuel, A.(1959). Some studies in machine learning using the game of checkers. *IBM J. of Research and Development, 3*, 210–229.

Samuel, A.(1967). Some studies in machine learning using the game of checkers, II—recent progress. *IBM J. of Research and Development, 11*, 601–617.

Sutton, R.S. (1984). Temporal credit assignment in reinforcement learning. Doctoral Dissertation, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst.

Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning, 3*, 9–44.

Tesauro, G. & Sejnowski, T.J. (1989). A parallel network that learns to play backgammon. *Artificial Intelligence, 39*, 357–390.

Tesauro, G. (1989). Connectionist learning of expert preferences by comparison training. In D.Touretzky (Ed.), *Advances in neural information processing, 1*, 99–106.

Tesauro, G. (1990). Neurogammon: a neural network backgammon program. *IJCNN Proceedings*, III, 33–39.

Utgoff, P.E. & Clouse, J.A. (1991). Two kinds of training information for evaluation function training. To appear in: *Proceedings of AAAI-91*.

Vapnik, V.N. & Chervonenkis (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory Prob. Appl., 16*, 264–280.

Widrow, B., et al. (1976). Stationary and nonstationary learning characteristics of the LMS adaptive filter. *Proceedings of the IEEE, 64*, 1151–1162.

Zadeh, N. & Kobliska, G. (1977). On optimal doubling in backgammon. *Management Science, 23*, 853–858.

Machine Learning, 8, 279–292 (1992)

Technical Note
Q-Learning

CHRISTOPHER J.C.H. WATKINS
25b Framfield Road, Highbury, London N5 1UU, England

PETER DAYAN
Centre for Cognitive Science, University of Edinburgh, 2 Buccleuch Place, Edinburgh EH8 9EH, Scotland

Abstract. Q-learning (Watkins, 1989) is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states.

This paper presents and proves in detail a convergence theorem for Q-learning based on that outlined in Watkins (1989). We show that Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely. We also sketch extensions to the cases of non-discounted, but absorbing, Markov environments, and where many Q values can be changed each iteration, rather than just one.

Keywords. Q-learning, reinforcement learning, temporal differences, asynchronous dynamic programming

1. Introduction

Q-learning (Watkins, 1989) is a form of model-free reinforcement learning. It can also be viewed as a method of asynchronous dynamic programming (DP). It provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains.

Learning proceeds similarly to Sutton's (1984; 1988) method of temporal differences (TD): an agent tries an action at a particular state, and evaluates its consequences in terms of the immediate reward or penalty it receives *and* its estimate of the value of the state to which it is taken. By trying all actions in all states repeatedly, it learns which are best overall, judged by long-term discounted reward. Q-learning is a primitive (Watkins, 1989) form of learning, but, as such, it can operate as the basis of far more sophisticated devices. Examples of its use include Barto and Singh (1990), Sutton (1990), Chapman and Kaelbling (1991), Mahadevan and Connell (1991), and Lin (1992), who developed it independently. There are also various industrial applications.

This paper presents the proof outlined by Watkins (1989) that Q-learning converges. Section 2 describes the problem, the method, and the notation, section 3 gives an overview of the proof, and section 4 discusses two extensions. Formal details are left as far as possible to the appendix. Watkins (1989) should be consulted for a more extensive discussion of Q-learning, including its relationship with dynamic programming and TD. See also Werbos (1977).

2. The task for $\mathcal{Q}$-learning

Consider a computational agent moving around some discrete, finite world, choosing one from a finite collection of actions at every time step. The world constitutes a controlled Markov process with the agent as a controller. At step n, the agent is equipped to register the state x_n ($\in X$) of the world, an can choose its action a_n ($\in \mathcal{A}$)[1] accordingly. The agent receives a probabilistic reward r_n, whose mean value $\mathcal{R}_{x_n}(a_n)$ depends only on the state and action, and the state of the world changes probabilistically to y_n according to the law:

$$\text{Prob}\,[y_n = y | x_n, a_n] = P_{x_n y}[a_n].$$

The task facing the agent is that of determining an optimal policy, one that maximizes total discounted expected reward. By discounted reward, we mean that rewards received s steps hence are worth less than rewards received now, by a factor of γ^s ($0 < \gamma < 1$). Under a policy π, the value of state x is

$$V^\pi(x) \equiv \mathcal{R}_x(\pi(x)) + \gamma \sum_y P_{xy}[\pi(x)] V^\pi(y$$

because the agent expects to receive $\mathcal{R}_x(\pi(x))$ immediately for performing the action π recommends, and then moves to a state that is 'worth' $V^\pi(y)$ to it, with probability $P_{xy}[\pi(x)]$. The theory of DP (Bellman & Dreyfus, 1962; Ross, 1983) assures us that there is at least one optimal stationary policy π^* which is such that

$$V^*(x) \equiv V^{\pi^*}(x) = \max_a \left\{ \mathcal{R}_x(a) + \gamma \sum_y P_{xy}[a] V^{\pi^*}(y) \right\}$$

is as well as an agent can do from state x. Although this might look circular, it is actually well defined, and DP provides a number of methods for calculating V^* and one π^*, assuming that $\mathcal{R}_x(a)$ and $P_{xy}[a]$ are known. The task facing a $\mathcal{Q}$ learner is that of determining a π^* without initially knowing these values. There are traditional methods (e.g., Sato, Abe & Takeda, 1988) for learning $\mathcal{R}_x(a)$ and $P_{xy}[a]$ while concurrently performing DP, but any assumption of certainty equivalence, i.e., calculating actions as if the current model were accurate, costs dearly in the early stages of learning (Barto & Singh, 1990). Watkins (1989) classes $\mathcal{Q}$-learning as incremental dynamic programming, because of the step-by-step manner in which it determines the optimal policy.

For a policy π, define $\mathcal{Q}$ values (or action-values) as:

$$\mathcal{Q}^\pi(x, a) = \mathcal{R}_x(a) + \gamma \sum_y P_{xy}[\pi(x)] V^\pi(y).$$

In other words, the $\mathcal{Q}$ value is the expected discounted reward for executing action a at state x and following policy π thereafter. The object in $\mathcal{Q}$-learning is to estimate the $\mathcal{Q}$

values for an optimal policy. For convenience, define these as $Q^*(x, a) \equiv Q^{\pi^*}(x, a)$, $\forall x, a$. It is straightforward to show that $V^*(x) = \max_a Q^*(x, a)$ and that if a^* is an action at which the maximum is attained, then an optimal policy can be formed as $\pi^*(x) \equiv a^*$. Herein lies the utility of the Q values—if an agent can learn them, it can easily decide what it is optimal to do. Although there may be more than one optimal policy or a^*, the Q^* values are unique.

In Q-learning, the agent's experience consists of a sequence of distinct stages or *episodes*. In the n^{th} episode, the agent:

- observes its current state x_n,
- selects and performs an action a_n,
- observes the subsequent state y_n,
- receives an immediate payoff r_n, and
- adjusts its Q_{n-1} values using a learning factor α_n, according to:

$$Q_n(x, a) = \begin{cases} (1 - \alpha_n) Q_{n-1}(x, a) + \alpha_n [r_n + \gamma V_{n-1}(y_n)] & \text{if } x = x_n \text{ and } a = a_n, \\ Q_{n-1}(x, a) & \text{otherwise,} \end{cases} \tag{1}$$

where

$$V_{n-1}(y) \equiv \max_b \{Q_{n-1}(y, b)\} \tag{2}$$

is the best the agent thinks it can do from state y. Of course, in the early stages of learning, the Q values may not accurately reflect the policy they implicitly define (the maximizing actions in equation 2). The initial Q values, $Q_0(x, a)$, for all states and actions are assumed given.

Note that this description assumes a look-up table representation for the $Q_n(x, a)$. Watkins (1989) shows that Q-learning may not converge correctly for other representations.

The most important condition implicit in the convergence theorem given below is that the sequence of episodes that forms the basis of learning must include an infinite number of episodes for each starting state and action. This may be considered a strong condition on the way states and actions are selected—however, under the stochastic conditions of the theorem, no method could be guaranteed to find an optimal policy under weaker conditions. Note, however, that the episodes need not form a continuous sequence—that is the y of one episode need not be the x of the next episode.

The following theorem defines a set of conditions under which $Q_n(x, a) \to Q^*(x, a)$ as $n \to \infty$. Define $n^i(x, a)$ as the index of the i^{th} time that action a is tried in state x.

Theorem

Given bounded rewards $|r_n| \leq \mathcal{R}$, learning rates $0 \leq \alpha_n < 1$, and

$$\sum_{i=1}^{\infty} \alpha_{n^i(x,a)} = \infty, \quad \sum_{i=1}^{\infty} [\alpha_{n^i(x,a)}]^2 < \infty, \quad \forall x, a, \tag{3}$$

then $\mathcal{Q}_n(x, a) \to \mathcal{Q}^*(x, a)$ as $n \to \infty$, $\forall x, a$, with probability 1.

3. The convergence proof

The key to the convergence proof is an artificial controlled Markov process called the *action-replay process* ARP, which is constructed from the episode sequence and the learning rate sequence α_n.

A formal description of the ARP is given in the appendix, but the easiest way to think of it is in terms of a card game. Imagine each episode $\langle x_t, a_t, y_t, r_t, \alpha_t \rangle$ written on a card. All the cards together form an infinite deck, with the first episode-card next-to-bottom and stretching infinitely upwards, in order. The bottom card (numbered 0) has written on it the agent's initial values $\mathcal{Q}_0(x, a)$ for all pairs of x and a. A state of the ARP, $\langle x, n \rangle$, consists of a card number (or *level*) n, together with a state x from the real process. The actions permitted in the ARP are the same as those permitted in the real process.

The next state of the ARP, given current state $\langle x, n \rangle$ and action a, is determined as follows. First, all the cards for episodes later than n are eliminated, leaving just a finite deck. Cards are then removed one at a time from top of this deck and examined until one is found whose starting state and action match x and a, say at episode t. Then a biased coin is flipped, with probability α_t of coming out heads, and $1 - \alpha_t$ of tails. If the coin turns up heads, the episode recorded on this card is replayed, a process described below; if the coin turns up tails, this card too is thrown away and the search continues for another card matching x and a. If the bottom card is reached, the game stops in a special, absorbing, state, and just provides the reward written on this card for x, a, namely $\mathcal{Q}_0(x, a)$.

Replaying the episode on card t consists of emitting the reward, r_t, written on the card, and then moving to the next state $\langle y_t, t - 1 \rangle$ in the ARP, where y_t is the state to which the real process went on that episode. Card t itself is thrown away. The next state transition of the ARP will be taken based on just the remaining deck.

The above completely specifies how state transitions and rewards are determined in the ARP. Define $P^{\text{ARP}}_{\langle x,n \rangle, \langle y,m \rangle}[a]$ and $\mathcal{R}^{(n)}_x(a)$ as the transition-probability matrices and expected rewards of the ARP. Also define:

$$P^{(n)}_{xy}[a] = \sum_{m=1}^{n-1} P^{\text{ARP}}_{\langle x,n \rangle, \langle y,m \rangle}[a]$$

as the probabilities that, for each x, n and a, executing action a at state $\langle x,n \rangle$ in the ARP leads to state y of the real process at some lower level in the deck.

As defined above, the ARP is as much a controlled Markov process as is the real process. One can therefore consider sequences of states and controls, and also optimal discounted $\mathcal{Q}^*$ values for the ARP.[2] Note that during such a sequence, episode cards are only removed from the deck, and are never replaced. Therefore, after a finite number of actions, the bottom card will always be reached.

3.1. Lemmas

Two lemmas form the heart of the proof. One shows that, effectively by construction, the optimal $\mathcal{Q}$ value for ARP state $\langle x, n \rangle$ and action a is just $\mathcal{Q}_n(x, a)$. The next shows that for almost all possible decks, $P_{xy}^{(n)}[a]$ converge to $P_{xy}[a]$ and $\mathcal{R}_x^{(n)}(a)$ converge to $\mathcal{R}_x(a)$ as $n \to \infty$. Informal statements of the lemmas and outlines of their proofs are given below; consult the appendix for the formal statements.

Lemma A

$\mathcal{Q}_n(x, a)$ are the optimal action values for ARP states $\langle x, n \rangle$ and ARP actions a.

The ARP was directly constructed to have this property. The proof proceeds by backwards induction, following the ARP down through the stack of past episodes.

Lemma B

Lemma B concerns the convergence of the ARP to the real process. The first two steps are preparatory; the next two specify the form of the convergence and provide foundations for proving that it occurs.

B.1

Consider a discounted, bounded-reward, finite Markov process. From any starting state x, the difference between the value of that state under the finite sequence of s actions and its value under that same sequence followed by any other actions tends to 0 as $s \to \infty$.

This follows from the presence of the discount factor which weighs the $(s + 1)^{\text{th}}$ state by $\gamma^s \to 0$ as $s \to \infty$.

B.2

Given any level l, there exists another yet higher level, h, such that the probability can be made arbitrarily small of straying below l after taking s actions in the ARP, starting from above h.

The probability, starting at level h of the ARP of straying below any fixed level l tends to 0 as $h \to \infty$. Therefore there is some sufficiently high level for which s actions can be safely accommodated, with an arbitrarily high probability of leaving the ARP above l.

B.3

With probability 1, the probabilities $P_{xy}^{(n)}[a]$ and expected rewards $\mathcal{R}_x^{(n)}(a)$ in the ARP converge and tend to the transition matrices and expected rewards in the real process as the level n increases to infinity. This, together with B.2, makes it appropriate to consider $P_{xy}^{(n)}[a]$ rather than the ARP transition matrices $P^{\text{ARP}}_{\langle x,n\rangle,\langle y,m\rangle}[a]$, i.e., essentially ignoring the level at which the ARP enters state y.

The ARP effectively estimates the mean rewards and transitions of the real process over all the episodes. Since its raw data are unbiased, the conditions on the sums and sums of squares of the learning rates $\alpha_{n^i(x,a)}$ ensure the convergence with probability one.

B.4

Consider executing a series of s actions in the ARP and in the real process. If the probabilities $P_{xy}^{(n)}[a]$ and expected rewards $\mathcal{R}_x^{(n)}(a)$ at appropriate levels of the ARP for each of the actions, are close to $P_{xy}[a]$ and $\mathcal{R}_x(a)$, $\forall a, x, y$, respectively, then the value of the series of actions in the ARP will be close to its value in the real process.

The discrepancy in the action values over a finite number s of actions between the values of two approximately equal Markov processes grows at most quadratically with s. So, if the transition probabilities and rewards are close, then the values of the actions must be close too.

3.2. The theorem

Putting these together, the ARP tends towards the real process, and so its optimal $\mathcal{Q}$ values do too. But $\mathcal{Q}_n(a, x)$ are the optimal $\mathcal{Q}$ values for the n^{th} level of the ARP (by Lemma A), and so tend to $\mathcal{Q}^*(x, a)$.

Assume, without loss of generality, that $\mathcal{Q}_0(x, a) < \mathcal{R}/(1 - \gamma)$ and that $\mathcal{R} \geq 1$.

Given $\epsilon > 0$, choose s such that

$$\gamma^s \frac{\mathcal{R}}{1 - \gamma} < \frac{\epsilon}{6}.$$

By B.3, with probability 1, it is possible to choose l sufficiently large such that for $n > l$, and $\forall a, x, y$,

$$|P_{xy}^{(n)}[a] - P_{xy}| < \frac{\epsilon}{3s(s+1)\mathcal{R}}, \text{ and } |\mathcal{R}_x^{(n)}(a) - \mathcal{R}_x(a)| < \frac{\epsilon}{3s(s+1)},$$

By B.2, choose h sufficiently large such that for $n > h$, the probability, after taking s actions, of ending up at a level lower than l is less than $\min\{(\epsilon(1 - \gamma)/6s\mathcal{R}), (\epsilon/3s(s+1)\mathcal{R})\}$. This means that

$$|P'^{(n)}_{xy}[a] - P_{xy}| < \frac{2\epsilon}{3s(s+1)\mathcal{R}}, \text{ and } |\mathcal{R}'^{(n)}_{x}(a) - \mathcal{R}_x(a)| < \frac{2\epsilon}{3s(s+1)},$$

where the primes on $P'^{(n)}$ and $\mathcal{R}'^{(n)}$ indicate that these are conditional on the level in the ARP after the s^{th} step being greater than l.

Then, for $n > h$, by B.4, compare the value $\bar{\mathcal{Q}}_{\text{ARP}}(\langle x, n\rangle, a_1, \ldots, a_s)$ of taking actions $a_1, \ldots, a_s$ at state x in the ARP, with $\bar{\mathcal{Q}}(x, a_1, \ldots, a_s)$ of taking them in the real process:[3]

$$|\bar{\mathcal{Q}}_{\text{ARP}}(\langle x, n\rangle, a_1, \ldots, a_s) - \bar{\mathcal{Q}}(x, a_1, \ldots, a_s)| < \frac{\epsilon(1-\gamma)}{6s\mathcal{R}} \frac{2s\mathcal{R}}{1-\gamma} + \frac{2\epsilon}{3s(s+1)} \frac{s(s+1)}{2} = \frac{2\epsilon}{3}. \quad (4)$$

Where, in equation 4, the first term counts the cost of conditions for B.2 not holding, as the cost of straying below l is bounded by $2s\mathcal{R}/(1-\gamma)$. The second term is the cost, from B.4, of the incorrect rewards and transition probabilities.

However, by B.1, the effect of taking only s actions makes a difference of less than $\epsilon/6$ for both the ARP and the real process. Also since equation 4 applies to any set of actions, it applies perforce to a set of actions optimal for *either* the ARP or the real process. Therefore

$$|\mathcal{Q}^*_{\text{ARP}}(\langle x, n\rangle, a) - \mathcal{Q}^*(x, a)| < \epsilon.$$

So, with probability 1, $\mathcal{Q}_n(x, a) \to \mathcal{Q}^*(x, a)$ as $n \to \infty$ as required.

4. Discussions and conclusions

For the sake of clarity, the theorem proved above was somewhat restricted. Two particular extensions to the version of Q-learning described above have been used in practice. One is the non-discounted case ($\gamma = 1$), but for a Markov process with absorbing goal states, and the other is to the case where many of the Q values are updated in each iteration rather than just one (Barto, Bradtke & Singh, 1991). The convergence result holds for both of these, and this section sketches the modifications to the proof that are necessary.

A process with absorbing goal states has one or more states which are bound in the end to trap the agent. This ultimate certainty of being trapped plays the rôle that $\gamma < 1$ played in the earlier proof, in ensuring that the value of state x under any policy π, $V^\pi(x)$, is bounded, and that lemma B.1 holds, i.e., that the difference between considering infinite and finite (s) numbers of actions tends to 0 as $s \to \infty$.

Since the process would always get trapped were it allowed to run, for every state x there is some number of actions $u(x)$ such that no matter what they are, there is a probability $p(x) > 0$ of having reached one of the goal states after executing those actions. Take

$u^* = \max_x \{u(x)\}$, and $p^* = \min_x \{p(x)\} > 0$ (since there is only a finite number of states). Then a crude upper bound for $V^\pi(x)$ is

$$|V^\pi(x)| \leq u^*\mathcal{R} + (1 - p^*)u^*\mathcal{R} + (1 - p^*)^2u^*\mathcal{R} + \ldots$$

$$= \frac{u^*\mathcal{R}}{p^*}$$

since in each u^* steps the agent earns a reward of less than $u^*\mathcal{R}$, and has probability less than $(1 - p^*)$ of not having been trapped. Similarly, the effect of measuring the reward after only ϕu^* steps is less than $(1 - p^*)^\phi u^*\mathcal{R} \to 0$ as $\phi \to \infty$, and so an equivalent of lemma B.1 does hold.

Changing more than one $\mathcal{Q}$ value on each iteration requires a minor modification to the action replay process ARP such that an action can be taken at any level at which it was executed in the real process—i.e., more than one action can be taken at each level. As long as the stochastic convergence conditions in equation 3 are still satisfied, the proof requires no non-trivial modification. The $\mathcal{Q}_n(x, a)$ values are still optimal for the modified ARP, and this still tends to the real process in the original manner. Intuitively, the proof relies on the ARP estimating rewards and transition functions based on many episodes, and this is just speeded up by changing more than one $\mathcal{Q}$ value per iteration.

Although the paper has so far presented an apparent dichotomy between $\mathcal{Q}$-learning and methods based on certainty equivalence, such as Sato, Abe and Takeda (1988), in fact there is more of a continuum. If the agent can remember the details of its learning episodes, then, after altering the learning rates, it can use each of them more than once (which is equivalent to putting cards that were thrown away, back in, lower down on the ARP stack). This biases the $\mathcal{Q}$-learning process towards the particular sample of the rewards and transitions that it has experienced. In the limit of re-presenting 'old' cards infinitely often, this reuse amounts to the certainty equivalence step of calculating the optimal actions for the observed sample of the Markovian environment rather than the actual environment itself.

The theorem above only proves the convergence of a restricted version of Watkins' (1989) comprehensive $\mathcal{Q}$-learning algorithm, since it does not permit updates based on the rewards from more than one iteration. This addition was pioneered by Sutton (1984; 1988) in his TD(λ) algorithm, in which a reward from a step taken r iterations previously is weighted by λ^r, where $\lambda < 1$. Unfortunately, the theorem does not extend trivially to this case, and alternative proof methods such as those in Kushner and Clark (1978) may be required.

This paper has presented the proof outlined by Watkins (1989) that $\mathcal{Q}$-learning converges with probability one under reasonable conditions on the learning rates and the Markovian environment. Such a guarantee has previously eluded most methods of reinforcement learning.

Acknowledgments

We are very grateful to Andy Barto, Graeme Mitchison, Steve Nowlan, Satinder Singh, Rich Sutton and three anonymous reviewers for their valuable comments on multifarious aspects of $\mathcal{Q}$-learning and this paper. Such clarity as it possesses owes to Rich Sutton's

tireless efforts. Support was from Philips Research Laboratories and SERC. PD's current address is CNL, The Salk Institute, PO Box 85800, San Diego, CA 92186-5800, USA.

Notes

1. In general, the set of available actions may differ from state to state. Here we assume it does not, to simplify the notation. The theorem we present can straightfowardly be extended to the general case.
2. The discount factor for the ARP will be taken to be γ, the same as for the real process.
3. The bars over the $\mathcal{Q}$ indicate that the sum is over only a finite number of actions, with 0 terminal reward.

Appendix

The action-replay process

The definition of the ARP is contingent on a particular sequence of episodes observed in the real process. The state space of the ARP is $\{\langle x, n\rangle\}$, for x a state of the real process and $n \geq 1$, together with one, special, absorbing state, and the action space is $\{a\}$ for a an action from the real process.

The stochastic reward and state transition consequent on performing action a at state $\langle x, n\rangle$ is given as follows. For convenience, define $n^i \equiv n^i(x, a)$, as the index of the i^{th} time action a was tried at state x. Define

$$i_* = \begin{cases} \text{argmax}_i \{n^i < n\} & \text{if } x, a \text{ has been executed before episode } n \\ 0 & \text{otherwise} \end{cases}$$

such that n^{i_*} is the last time before episode n that x, a was exeucted in the real process. If $i_* = 0$, the reward is set as $\mathcal{Q}_0(x, a)$, and the ARP absorbs. Otherwise, let

$$i_e = \begin{cases} i_* & \text{with probablity } \alpha_{n^{i_*}} \\ i_* - 1 & \text{with probability } (1 - \alpha_{n^{i_*}})\alpha_{n^{i_*-1}} \\ i_* - 2 & \text{with probability } (1 - \alpha_{n^{i_*}})(1 - \alpha_{n^{i_*-1}})\alpha_{n^{i_*-2}}, \\ & \vdots \\ 0 & \text{with probability } \prod_{i=1}^{i_*} (1 - \alpha_{n^i}) \end{cases}$$

be the index of the episode that is *replayed* or taken, chosen probabilistically from the collection of existing samples from the real process. If $i_e = 0$, then the reward is set at $\mathcal{Q}_0(x, a)$ and the ARP absorbs, as above, Otherwise, taking i_e provides reward $r_{n^{i_e}}$, and causes a state transition to $\langle y_{n^{i_e}}, n^{i_e} - 1\rangle$ which is at level $n^{i_e} - 1$. This last point is crucial, taking an action in the ARP always causes a state transition to a lower level—so it ultimately terminates. The discount factor in the ARP is γ, the same as in the real process.

Lemma A: $\mathcal{Q}_n$ are optimal for the ARP

$\mathcal{Q}_n(x, a)$ are the optimal action values for ARP states $\langle x, n\rangle$ and ARP actions a. That is

$$\mathcal{Q}_n(x, a) = \mathcal{Q}^*_{\mathsf{ARP}}(\langle x, n\rangle, a), \ \forall a, x, \text{ and } n \geq 0.$$

Proof

By induction. From the construction of the ARP, $\mathcal{Q}_0(x, a)$ is the optimal—indeed the only possible—action value of $\langle x, 0\rangle$, a. Therefore,

$$\mathcal{Q}_0(x, a) = \mathcal{Q}^*_{\mathsf{ARP}}(\langle x, 0\rangle, a).$$

Hence the theorem holds for $n = 0$.

Suppose that the values of $\mathcal{Q}_{n-1}$, as produced by the $\mathcal{Q}$-learning rule, are the optimal action values for the ARP at level $n - 1$, that is

$$\mathcal{Q}_{n-1}(x, a) = \mathcal{Q}^*_{\mathsf{ARP}}(\langle x, n - 1\rangle, a), \ \forall a, x.$$

This implies that the $V_{n-1}(x)$ are the optimal values V^* for the ARP at the $n - 1^{\text{th}}$ level, that is

$$V^*(\langle x, n - 1\rangle) = V_{n-1}(x) \equiv \max_a \mathcal{Q}_{n-1}(x, a).$$

Now consider the cases in trying to perform action a in $\langle x, n\rangle$. If $x, a \neq x_n, a_n$, then this is the same as performing a in $\langle x, n - 1\rangle$, and $\mathcal{Q}_n(x, a) = \mathcal{Q}_{n-1}(x, a)$. Therefore,

$$\mathcal{Q}_n(x, a) = \mathcal{Q}_{n-1}(x, a) = \mathcal{Q}^*_{\mathsf{ARP}}(\langle x, n - 1\rangle, a) = \mathcal{Q}^*_{\mathsf{ARP}}(\langle x, n\rangle, a)$$

Otherwise, performing a_n in $\langle x_n, n\rangle$

- with probability $1 - \alpha_n$ is exactly the same as performing a_n in $\langle x_n, n - 1\rangle$, or
- with probability α_n yields immediate reward r_n and new state $\langle y_n, n - 1\rangle$.

Therefore the optimal action value in the ARP of $\langle x_n, n\rangle$, a_n is

$$\begin{aligned}\mathcal{Q}^*_{\mathsf{ARP}}(\langle x_n, n\rangle, a_n) &= (1 - \alpha_n)\mathcal{Q}^*_{\mathsf{ARP}}(\langle x_n, n - 1\rangle, a_n) + \alpha_n(r_n + \gamma V^*(\langle y_n, n - 1\rangle)) \\ &= (1 - \alpha_n)\mathcal{Q}_{n-1}(x_n, a_n) + \alpha_n(r_n + \gamma V_{n-1}(y_n)) \\ &= \mathcal{Q}_n(x_n, a_n)\end{aligned}$$

from the induction hypothesis and the $\mathcal{Q}_n$ interation formula in equation 1.

Hence, $\mathcal{Q}_n(x, a) = \mathcal{Q}^*_{\mathsf{ARP}}(\langle x, n\rangle, a), \ \forall a, x$, as required.

Lemma B

B.1 Discounting infinite sequences

Consider a discounted, bounded-reward, finite Markov process with transition matrix $P_{xy}[a]$. From any starting state x, the difference between the value of that state under any set of s actions and under those same s actions followed by any arbitrary policy tends to 0 as $s \to \infty$.

Proof

Ignoring the value of the $s + 1^{\text{th}}$ state incurs a penalty of

$$\delta \equiv \gamma^s \sum_{y_{s+1}} P_{y_s y_{s+1}}[a_s] V^\pi(y_{s+1})$$

But if all rewards are bounded by $\mathcal{R}$, $|V^\pi(x)| < \mathcal{R}/(1 - \gamma)$, and so

$$|\delta| < \gamma^s \frac{\mathcal{R}}{1 - \gamma} \to 0 \text{ as } s \to \infty.$$

B.2 The probability of straying below level l is executing s *actions can be make arbitrarily small*

Given any level l, there exists another yet higher level, h, such that the probability can be made arbitrarily small of straying below l after taking s actions in the ARP, starting from above h.

Proof

Define i_h as the largest i such that $n^i(x, a) \leq n$, and i_l as the smallest such that $n^i(x, a) \geq l$. Then, defining $\alpha_{n^0} = 1$, the probability of straying below l starting from $\langle x, n \rangle$, $n > l$ executing action a is:

$$\left[\prod_{i=i_l}^{i_h} (1 - \alpha_{n^i}) \right] \sum_{j=0}^{i_l - 1} \left\{ \alpha_{n^j} \prod_{k=j+1}^{i_l - 1} (1 - \alpha_{n^k}) \right\} \leq \prod_{i=i_l}^{i_h} (1 - \alpha_{n^i}),$$

where, as before, $n^i \equiv n^i(x, a)$. But $\Pi_{i=i_l}^{i_h}(1 - \alpha_{n^i}) < \exp(-\Sigma_{i=i_l}^{i_h} \alpha_{n^i}) \to 0$ as n and hence $i_h \to \infty$. Furthermore, since the state and action spaces are finite, given η, there exists some level n_1 such that starting above there from any (x, a) leads to a level above l with probability at least $1 - \eta$. This argument iterates for the second action with n_1 as the new lower limit. η can be chosen appropriately to set the overall probability of straying below l less than any arbitrary $\epsilon > 0$.

B.3 Rewards and transition probabilities converge with probabability 1

With probability 1, the probabilities $P_{xy}^{(n)}[a]$ and expected rewards $\mathfrak{R}_x^{(n)}(a)$ in the ARP converge and tend to the transition matrices and expected rewards in the real process as the level n increases to infinity.

Proof

A standard theorem in stochastic convergence (e.g., theorem 2.3.1 of Kushner & Clark, 1978) states that if X_n are updated according to

$$X_{n+1} = X_n + \beta_n(\xi_n - X_n)$$

where $0 \leq \beta_n < 1$, $\Sigma_{i=1}^{\infty} \beta_n = \infty$, $\Sigma_{i=1}^{\infty} \beta_n^2 < \infty$, and ξ_n are bounded random variables with mean Ξ, then

$$X_n \to \Xi, \text{ as } n \to \infty, \text{ with probability 1.}$$

If $\mathfrak{R}_{\langle x,n\rangle}(a)$ is the expected immediate reward for performing action a from state x at level n in the ARP, then $\mathfrak{R}_{\langle x,n\rangle}(a)$ satisfies

$$\mathfrak{R}_{\langle x,n^{i+1}\rangle}(a) = \mathfrak{R}_{\langle x,n^i\rangle}(a) + \alpha_{n^{i+1}}(r_{n^{i+1}} - \mathfrak{R}_{\langle x,n^i\rangle}(a))$$

where the $\mathfrak{R}$ and the α satisfy the conditions of the theorem with $\Xi = \mathfrak{R}_x(a)$, and remembering that n^i is the i^{th} occasion on which action a was tried at state x. Therefore $\mathfrak{R}_{\langle x,n\rangle}(a) \to \mathfrak{R}_x(a)$ as $n \to \infty$, with probbility one. Also, since there is only a finite number of states and actions, the convergence is uniform.

Similarly, define

$$\chi_n(y) = \begin{cases} 1 & \text{if } y_n = y \\ 0 & \text{otherwise} \end{cases}$$

as a (random variable) indicator function of the n^{th} transition, mean value $P_{xy}(a)$. Then, with $P_{xy}^{(n)}[a]$ as the probability of ending up at state y based on a transition from state x using action a at level n in the ARP,

$$P_{xy}^{(n^{i+1})}[a] = P_{xy}^{(n^i)}[a] + \alpha_{n^{i+1}}(\chi_{n^{i+1}} - P_{xy}^{(n^i)}[a]),$$

and so, by the theorem, $P_{xy}^{(n)}[a] \to P_{xy}[a]$ (the transition matrix in the real process) as $n \to \infty$, with probability one.

Since, in addition, all observations from the real process are independent, and, by B.2, the probability of straying below a fixed level k can be made arbitrarily small, the transition probabilities and expected rewards for a single step *conditional* on ending up at a level greater than k also converge to $P_{xy}[a]$ and $\mathfrak{R}_x(a)$ as $n \to \infty$.

B.4 Close rewards and transitions imply close values

Let $P^i_{xy}[a]$, for $i = 1 \ldots s$ be the transition matrices of s Markov chains, and $\mathcal{R}^i_x(a)$ be the reward functions. Consider the s-step chain formed from the concatenation of these—i.e., starting from state x_1, move to state x_2 according to $P^1_{x_1x_2}[a_1]$, then state x_3 according to $P^2_{x_2x_3}[a_2]$, and so on, with commensurate rewards. Given $\eta > 0$, if $P^i[a]$ are within $\eta/\mathcal{R}$ of $P_{xy}[a]$, $\forall a, x, y$, and $\mathcal{R}^1_x(a) \ldots \mathcal{R}^s_x(a)$ are within η of $\mathcal{R}_x(a)$, $\forall a, x$, then the value of the s actions in the concatenated chain is within $\eta s(s + 1)/2$ of their value in the real process.

Proof

Define:

$$\bar{\mathcal{Q}}(x, a_1, a_2) = \mathcal{R}_x(a_1) + \gamma \sum_y P_{xy}[a_1]\mathcal{R}_y(a_2)$$

as the expected reward in the real process for executing two actions, a_1 and a_2 at state x, and

$$\bar{\mathcal{Q}}'(x, a_1, a_2) = \mathcal{R}^1_x(a_1) + \gamma \sum_y P^1_{xy}[a_1]\mathcal{R}^2_y(a_2)$$

as the equivalent in the concatenated chain for exactly the same actions.

Then, since $|\mathcal{R}^i_x(a) - \mathcal{R}_x(a)| < \eta$ and $P^i_{xy}[a] - P_{xy}[a]| < \eta/\mathcal{R}$, $\forall a, i, x, y$,

$$\begin{aligned}|\bar{\mathcal{Q}}'(x, a_1, a_2) - \bar{\mathcal{Q}}(x, a_1, a_2)| \leq\ & |\mathcal{R}^1_x(a_1) - \mathcal{R}_x(a_1)| + \\ & \gamma \left| \sum_y P^2_{xy}[a_2](\mathcal{R}^2_y(a_2) - \mathcal{R}_x(a_2)) \right| + \\ & \gamma \left| \sum_y (P^2_{xy}[a_2] - P_{xy}[a_2])\mathcal{R}_x(a_2) \right| \\ <\ & 3\eta\end{aligned}$$

Similarly, for s actions,

$$|\bar{\mathcal{Q}}'(x, a_1, \ldots, a_s) - \bar{\mathcal{Q}}(x, a_1, \ldots, a_s)| < \frac{s(s+1)}{2}\eta.$$

This applies to the ARP if the rewards and transition matrices at the successively lower levels are sufficiently close to those in the real process—the main body of the theorem quantifies the cost of this condition failing.

References

Barto, A.G., Bradtke, S.J. & Singh, S.P. (1991). *Real-time learning and control using asynchronous dynamic programming.* (COINS technical report 91-57). Amherst: University of Massachusetts.

Barto, A.G. & Singh, S.P. (1990). On the computational economics of reinforcement learning. In D.S. Touretzky, J. Elman, T.J. Sejnowski & G.E. Hinton, (Eds.), *Proceedings of the 1990 Connectionist Models Summer School.* San Mateo, CA: Morgan Kaufmann.

Bellman, R.E. & Dreyfus, S.E. (1962). *Applied dynamic programming.* RAND Corporation.

Chapman, D. & Kaelbling, L.P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *Proceedings of the 1991 International Joint Conference on Artificial Intelligence* (pp. 726-731).

Kushner, H. & Clark, D. (1978). *Stochastic approximation methods for constrained and unconstrained systems.* Berlin, Germany: Springer-Verlag.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning, 8.*

Mahadevan & Connell (1991). Automatic programming of behavior-based robots using reinforcement learning. *Proceedings of the 1991 National Conference on AI* (pp. 768-773).

Ross, S. (1983). *Introduction to stochastic dynamic programming.* New York, Academic Press.

Sato, M., Abe, K. & Takeda, H. (1988). Learning control of finite Markov chains with explicit trade-off between estimation and control. *IEEE Transactions on Systems, Man and Cybernetics, 18,* pp. 677-684.

Sutton, R.S. (1984). *Temporal credit assignment in reinforcement learning.* PhD Thesis, University of Massachusetts, Amherst, MA.

Sutton, R.S. (1988). Learning to predict by the methods of temporal difference. *Machine Learning, 3,* pp. 9-44.

Sutton. R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards.* PhD Thesis, University of Cambridge, England.

Werbos, P.J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook, 22,* pp. 25-38.

Machine Learning, 8, 293–321 (1992)

Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching

LONG-JI LIN ljl@cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

Abstract. To date, reinforcement learning has mostly been studied solving simple learning tasks. Reinforcement learning methods that have been studied so far typically converge slowly. The purpose of this work is thus two-fold: 1) to investigate the utility of reinforcement learning in solving much more complicated learning tasks than previously studied, and 2) to investigate methods that will speed up reinforcement learning.

This paper compares eight reinforcement learning frameworks: *adaptive heuristic critic* (*AHC*) learning due to Sutton, *Q-learning* due to Watkins, and three extensions to both basic methods for speeding up learning. The three extensions are experience replay, learning action models for planning, and teaching. The frameworks were investigated using connectionism as an approach to generalization. To evaluate the performance of different frameworks, a dynamic environment was used as a testbed. The environment is moderately complex and nondeterministic. This paper describes these frameworks and algorithms in detail and presents empirical evaluation of the frameworks.

Keywords. Reinforcement learning, planning, teaching, connectionist networks

1. Introduction

Reinforcement learning is an interesting learning technique. It requires only a scalar reinforcement signal as performance feedback from the environment. Reinforcement learning often involves two difficult subproblems. The first is known as the *temporal credit assignment problem* (Sutton, 1984). Suppose a learning agent performs a sequence of actions and finally obtains certain outcomes. It must figure out how to assign credit or blame to each individual situation (or situation-action pair) to adjust its decision making and improve its performance. The second subproblem is the *generalization problem* (also known as the *structural credit assignment problem*). When the problem space is too large to explore completely, a learning agent must have the ability to guess about new situations based on experience with similar situations. In the course of learning, both subproblems must be solved.

The most popular and best-understood approach to the temporal credit assignment problem is *temporal difference* (TD) methods (Sutton, 1988). TD methods, which have a solid mathematical foundation, are closely related to dynamic programming (Barto et al., 1991). Two forms of TD-based reinforcement learning have been proposed: the *adaptive heuristic critic* (AHC) learning architecture (Sutton, 1984; Barto et al., 1990) and *Q-learning* (Watkins, 1989). While both methods have been applied to solve some simple learning problems (e.g., (Sutton, 1984; Anderson, 1987; Watkins, 1989; Kaelbling, 1990; Sutton, 1990)), they were found to converge slowly.

Several approaches to the generalization problem have been studied for reinforcement learning. Anderson (1987) and Lin (1991a, 1991b, 1991c) have successfully combined TD

methods with the connectionist error backpropagation algorithm (Rumelhart et al., 1986). Watkins (1989) used a CMAC algorithm, Grefenstette et al. (1990) a genetic algorithm method, Mahadevan and Connell (1991) a statistical clustering method, Chapman and Kaelbling (1991) a method similar to decision-tree algorithms, and Moore (1991) a method which is basically look-up tables but which uses variable state resolution.

The goal of this work is to study connectionist reinforcement learning in solving non-trivial learning problems and to study methods for speeding up reinforcement learning. Although this study took the connectionist approach, the algorithms and results presented here appear to be relevant to other generalization methods as well.

This work studied eight frameworks for connectionist reinforcement learning: connectionist AHC- and Q-learning, and their extensions. A number of studies (Anderson, 1987; Chapman & Kaelbling, 1991; Mahadevan & Connell, 1991) have found that AHC- and Q-learning often converge slowly. For speedup, three extensions to both basic methods were investigated in this work: experience replay, learning action models for planning, and teaching.

The different frameworks were evaluated using a nondeterministic dynamic environment as a testbed. The environment consists of four kinds of objects: the agent, stationary food and obstacles, and moving enemies. The task of the agent is to survive in the environment, which is by no means trivial for a knowledge-poor agent, since the agent has to process a large number of input signals, has several actions to choose from, and has multiple goals.

The rest of this paper is organized as follows. Section 2 presents the learning frameworks and algorithms. Section 3 describes the dynamic environment and the survival task. Sections 4 and 5 present the implementation and performance of the learning agents. Section 6 assesses the different learning frameworks by comparing the agents' performance. Section 7 discusses the limitations. Finally, Section 8 concludes the paper by summarizing the lessons learned from this study.

2. Reinforcement learning frameworks

In the reinforcement learning paradigm, a learning agent continually receives sensory inputs from the environment, selects and performs actions to affect the environment, and after each action, receives from the environment a scalar signal called *reinforcement*. The reinforcement can be positive (reward), negative (punishment), or 0. The objective of learning is to construct an optimal *action selection policy* (or simply policy) that maximizes the agent's performance. A natural measure of performance is the *discounted cumulative reinforcement* (or for short, *utility*) (Barto et al., 1990):

$$V_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1}$$

where V_t is the discounted cumulative reinforcement starting from time t throughout the future, r_t is the reinforcement received after the transition from time t to $t + 1$, and $0 \leq \gamma \leq 1$ is a *discount factor*, which adjusts the importance of long-term consequences of actions.

This section describes eight frameworks for connectionist reinforcement learning. They are summarized as follows:

- AHCON: connectionist AHC-learning
- AHCON-R: AHCON plus experience replay
- AHCON-M: AHCON plus using action models
- AHCON-T: AHCON plus experience replay plus teaching
- QCON: connectionist Q-learning
- QCON-R: Q-CON plus experience replay
- QCON-M: QCON plus using action models
- QCON-T: QCON plus experience replay plus teaching

The basic idea behind these frameworks is to learn an *evaluation function* (see *eval* and *util* below) to predict the discounted cumulative reinforcement to be received in the future. The evaluation function is represented using connectionist networks and learned using a combination of temporal difference (TD) methods (Sutton, 1988) and the error backpropagation algorithm (Rumelhart et al., 1986). In essence, TD methods compute the error (called the *TD error*) between *temporally successive predictions*, and the backpropagation algorithm minimizes the error by modifying the weights of the networks.

Before starting to discuss the learning frameworks, two terms are defined for later use:

- *eval*(x): the expected discounted cumulative reinforcement that will be received starting from world state x, or simply the utility of state x;
- *util*(x, a): the expected discounted cumulative reinforcement that will be received after the execution of action a in response to world state x; or simply the utility of the state-action pair (x, a).

In a deterministic world, *util*(x, a) is equal to the immediate reinforcement r plus the utility of the next state y, discounted by γ:

$$util(x, \mathrm{a}) = r + \gamma \cdot eval(y) \tag{2}$$

Equation 2 can be generalized to a nondeterministic world by taking into consideration the probabilities of multiple outcomes (Watkins, 1989). Note that it is only meaningful to estimate both functions, *eval* and *util*, relative to some policy, since different policies will result in different reinforcements received. Unless explicitly mentioned, both functions are assumed to be relative to the current policy.

2.1. AHC-learning: Framework AHCON

Framework AHCON (Figure 1) is a connectionist implementation of the adaptive heuristic critic learning architecture (Barto et al., 1990). It consists of three components: an *evaluation network*, a *policy network*, and a *stochastic action selector*. In essence, the framework decomposes reinforcement learning into two subtasks. The first subtask is to construct,

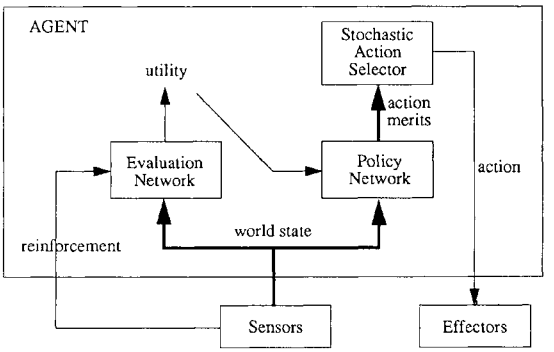


Figure 1. Framework AHCON. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

using TD methods, an evaluation network that accurately models the function $eval(x)$. The policy network takes a world state[1] as inputs and assigns to each action a value (called *action merit*) indicating the relative merit of performing that action in response to that world state. Thus, the second subtask is to adjust the policy network so that it will assign higher merits to actions that result in higher utilities (as measured by the evaluation network).

Ideally, the outputs of the policy network will approach the extremes, for example, 1 for the best action(s) and -1 for the others. Given a policy network and a state, the agent's policy is to choose the action that has the highest merit. In order to learn an optimal policy effectively, the agent has to take actions that appear suboptimal once in a while so that the relative merits of actions can be assessed. A stochastic action selector, which favors actions having high merits, can be used for this purpose.

In the course of learning, both the evaluation and policy networks are adjusted incrementally. Figure 2 summarizes the learning algorithm, in which the simplest TD method, TD(0), is used.[2] To use TD methods to learn an evaluation function, the first step is to write down a recursive definition of the desired function. By Definition 1, the utility of a state x is the immediate payoff r plus the utility of the next state y, discounted by γ.[3] Therefore, the desired function must satisfy Equation 3:

$$eval(x) = r + \gamma \cdot eval(y) \tag{3}$$

During learning, the equation may not hold true. The difference between the two sides of the equation (called TD error) is reduced by adjusting the weights of the evaluation network using the backpropagation algorithm (Step 5).

The policy network is also adjusted (Step 6) according to the same TD error. The idea is as follows: If $e' > e$, meaning that action a is found to be better than previously expected, the policy is modified to increase the merit of the action. On the other hand, if $e' < e$, the policy is modified to decrease the merit. The policy network is not updated with respect to actions other than a, since from a single experience we know nothing about the merits of the other actions.

1. $x \leftarrow$ current state; $e \leftarrow eval(x)$;
2. $a \leftarrow select(policy(x), T)$;
3. Perform action a; $(y, r) \leftarrow$ new state and reinforcement;
4. $e' \leftarrow r + \gamma \cdot eval(y)$;
5. Adjust evaluation network by backpropagating TD error $(e' - e)$ through it with input x;
6. Adjust policy network by backpropagating error Δ through it with input x, where $\Delta_i = \begin{cases} e' - e & \text{if } i = a \\ 0 & \text{otherwise} \end{cases}$
7. go to 1.

Figure 2. The algorithm for Framework AHCON.

During learning, the stochastic action selector (i.e., the *select* function in the algorithm) chooses actions randomly according to a probability distribution determined by action merits. In this work, the probability of choosing action a_i is computed as follows:

$$Prob(a_i) = e^{m_i/T} / \sum_k e^{m_k/T} \tag{4}$$

where m_i is the merit of action a_i, and the *temperature* T adjusts the randomness of action selection.

2.2. *Q-learning: Framework QCON*

Framework QCON (Figure 3) is a connectionist implementation of *Q-learning* (Watkins, 1989). QCON learns a network (called the *utility network*) that accurately models the function *util*(x, a). Given a utility network and a state x, the agent's policy is to choose the

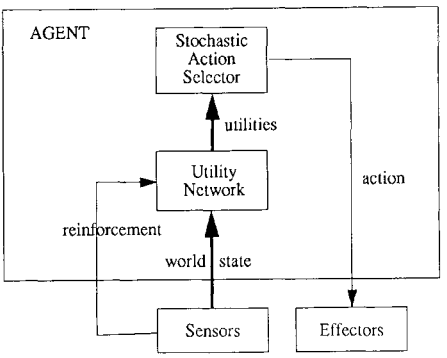


Figure 3. Framework QCON. The bold lines indicate a vector of signals and the thinner lines indicate a scalar signal.

1. $x \leftarrow$ current state; for each action i, $U_i \leftarrow util(x, i)$;
2. $a \leftarrow select(U, T)$;
3. Perform action a; $(y, r) \leftarrow$ new state and reinforcement;
4. $u' \leftarrow r + \gamma \cdot Max\{util(y, k) \mid k \in actions\}$;
5. Adjust utility network by backpropagating error ΔU through it with input x, where $\Delta U_i = \begin{cases} u' - U_i & \text{if } i = a \\ 0 & \text{otherwise} \end{cases}$
6. go to 1.

Figure 4. The algorithm for Framework QCON.

action a for which $util(x, a)$ is maximal. Therefore, the utility network is not only an evaluation function but also used as a policy.

The learning algorithm is depicted in Figure 4 and briefly described below. Generally speaking, the utility of an action a in response to a state x is equal to the immediate payoff r plus the best utility that can be obtained from the next state y, discounted by γ. Therefore, the desired *util* function must satisfy Equation 5:

$$util(x, a) = r + \gamma \cdot Max\{util(y, k) \mid k \in actions\} \tag{5}$$

During learning, the difference between the two sides of the equation is minimized using the backpropagation algorithm (Step 5). Note that the network is not modified with respect to actions other than a, since from a single experience we know nothing about the utilities of the other actions.

The utility network described above has multiple outputs (one for each action). It could be replaced by multiple networks (one for each action); each with a single output. The former implementation might be less desirable, because whenever the single utility network is modified with respect to an action, no matter whether it is desired or not, the network is also modified with respect to the other actions as a result of shared hidden units between actions. A similar argument can apply to the policy network of Framework AHCON. (This study used multiple networks with single outputs.)

Again, a stochastic action selector is needed to explore the consequences of different actions. The same action selector used by Framework AHCON can be employed for this purpose.

2.3. Experience replay: Frameworks AHCON-R and QCON-R

Reinforcement learning is a trial-and-error method, and thus may result in undesirable damages to the learning agent in a hostile environment. The faster the agent learns, the less damage the agent is likely to suffer. Generally speaking, temporal difference learning is a slow process for temporal credit assignment, especially when credits must be propagated through a long action sequence. The rest of this section describes three techniques to speed up the credit propagation process and/or to shorten the trial and error process.

The basic AHC- and Q-learning algorithms described above are inefficient in that experiences obtained by trial-and-error are utilized to adjust the networks only once and then thrown away. This is wasteful, since some experiences may be rare and some (such as those involving damages) costly to obtain. Experiences should be reused in an effective way.

In this paper, an *experience* is a quadruple, (x, a, y, r), meaning that the execution of an action a in a state x results in a new state y and reinforcement r. A *lesson* is a temporal sequence of experiences starting from an initial state to a final state, where the goal may or may not be achieved.

The most straightforward way of reusing experiences is what I call *experience replay*. By experience replay, the learning agent simply remembers its past experiences and repeatedly presents the experiences to its learning algorithm as if the agent experienced again and again what it had experienced before. The result of doing this is that the process of credit/blame propagation is sped up and therefore the networks usually converge more quickly. However, it is important to note that a condition for experience replay to be useful is that the laws that govern the environment of the learning agent should not change over time (or at least not change rapidly), simply because if the laws have changed, past experiences may become irrelevant or even harmful.

Experience replay can be more effective in propagating credit/blame if a sequence of experiences is replayed in temporally **backward** order. It can be even more effective if TD(λ) methods are used with the *recency factor*, λ, greater than zero. With $\lambda > 0$, the TD error for adjusting network weights is determined by the discrepancy between not only two but multiple consecutive predictions. Lin (1991c) presents a detailed algorithm and study of using backward replay and nonzero λ in a mobile robot domain. The results of that paper are encouraging. In this paper, however, only backward replay and $\lambda = 0$ are used for simplicity reason.

The algorithm for AHCON-R is simply the one for AHCON plus repeated presentation of past experiences to the algorithm, except that *experiences involving non-policy actions (according to the current policy) are not presented*. By presenting an experience to the algorithm, I mean to bind the variables, (x, a, y, r), used in the algorithm with the experience. The reason for the exception is the following: AHCON estimates $eval(x)$ (relative to the current policy) by sampling the current policy (i.e., executing some policy actions). Replaying past experiences, which is equivalent to sampling policies in the past, will disturb the sampling of the current policy, if the past policies are different from the current one. Consider an agent whose current policy chooses a very good action a in state x. If the agent changes to choose a very bad action b in the same state, the utility of state x, $eval(x)$, will drop dramatically. Similarly, the utility of x will be underestimated if the bad action b are replayed a few times. For experience replay to be useful, the agent should only replay the experiences involving actions that still follow the current policy.

Just like AHCON-R but for a different reason, QCON-R should also replay only actions which follow the current policy. Consider the previous example. Even if the agent changes from choosing good action a to choosing bad action b in state x, the values of both $util(x, a)$ and $util(x, b)$ will not change. As a matter of fact, if the *util* function is implemented as look-up tables, Watkins (1989) has shown that Q-learning is guaranteed to learn an optimal policy as long as all state-action pairs are tried an infinite number of times (along with a few weak conditions). In other words, it is not harmful for the tabular version of Q-learning

to replay bad actions. But this does not hold true for connectionist Q-learning, because whenever the backpropagation algorithm modifies a utility network with respect to one input state, it also affects the network with respect to many or all of the possible input states. If a utility network is trained on bad experiences many times consecutively, the network might come to underestimate the real utilities of some state-action pairs.

In short, AHCON-R and QCON-R should replay only policy actions. However, since an agent may use a stochastic policy as suggested previously, all actions have more or less possibility of being chosen by any policy. It is difficult to say whether an action is a current policy action or not. In this work, an action is considered as non-policy action, if its probability (see Equation 4) of being chosen (according to the *current* policy) is lower than P_l.[4] For AHCON-R, $P_l = 0.2$ was found to give roughly the best performance for the learning task to be described later in this paper. For QCON-R P_l, = 0.1, 0.01 and 0.001 were tried and found to give similar performance. ($P_l = 0$ indeed gave bad performance.) In other words, AHCON-R is quite sensitive to replaying non-policy actions, while QCON-R is much less sensitive. The difference in sensitivity can be explained by the fact that replaying non-policy actions, in the case of using look-up tables, is bad for AHC-learning but is fine for Q-learning.

For Frameworks AHCON-R and QCON-R to be efficient in terms of time and memory space, only "recent" experiences need to be stored and replayed. As a matter of fact, it is not only unnecessary but also harmful to replay an experience as many times as possible, because the networks might be over-trained and become too specific to that experience, which usually harms generalization. Consider an agent living in a nondeterministic world. Whenever the agent is in state x and performs action a, 80% of the time it will receive a great penalty, while 20% of the time it will receive no penalty. If the agent just experienced the situation with no penalty and is trained on this experience many times, the agent will come to believe that it is harmless to perform a in x, which is certainly wrong. Section 4.7 describes a partial solution to prevent over-training.

2.4. Using action models: Frameworks AHCON-M and QCON-M

Another way of reusing past experiences, which was investigated in the Dyna architecture (Sutton, 1990), is to use experiences to build an *action model*[5] and use it for planning and learning. An action model is a function from a state and an action, (x, a), to the next state and the immediate reinforcement, (y, r). For stochastic worlds, each action may have multiple outcomes. To be accurate, the action model may also need to model the probability distribution of outcomes, which is an issue this paper does not address.

An action model is intended to mimic the behaviors of the environment. Using an accurate action model (if it is available), the agent can experience the consequences of actions without participating in the real world. As a result, the agent will learn faster (if projecting is faster than acting) and more importantly, fewer mistakes will be committed in the real world. Frameworks AHCON-M and QCON-M to be discussed below are based on this idea.

2.4.1. *Framework AHCON-M*

Framework AHCON-M is very similar to Framework AHCON, but differs in that AHCON-M also learns an action model and uses it to do what Sutton called *relaxation planning* (Sutton, 1990). Relaxation planning, which is closely related to dynamic programming, is an incremental planning process that consists of a series of shallow (usually one-step look-ahead) searches and ultimately produces the same results as a conventional deep search. In Sutton's Dyna architecture, his learning algorithm (similar to the one for AHCON) is applied to real-world situations faced by the agent and also hypothetical situations randomly generated. In the former case the next state is obtained by executing an action, while in the latter case the next state is obtained by applying an action model. His approach has two inefficiencies. First of all, since hypothetical situations are randomly generated, the agent may spend too much effort planning what to do about hypothetical situations that will never happen in the real world at all. Second, it is not clear how his approach decides when relaxation planning is no longer necessary and to stop doing it.

The relaxation planning algorithm (see Figure 5) proposed here addresses both inefficiencies by projecting all actions from states actually visited, not from states chosen at random. Since all actions to a state are examined at the same time, the relative merits of actions can be more directly and effectively assessed than the kind of *policy iteration* (Howard, 1960) used in AHCON and the Dyna architecture. Compared with the Dyna architecture, the disadvantage of this algorithm, however, is that the number of hypothetical experiences that can be generated is limited by the number of states visited.

In a deterministic world, the utility of a state x is equal to the immediate payoff r obtained from executing the best action a plus the discounted utility of the new state y:

$$eval(x) = Max\{r + \gamma \cdot eval(y) \mid a \in actions\} \quad \text{(Note: } y \text{ is a function of } a\text{)} \quad (6)$$

Thus, if a correct action model is available, the utility of a state can be accurately estimated by looking ahead one step. During learning, the evaluation network is adjusted to minimize the difference between the two sides of Equation 6 (Step 6).

The policy network is updated in the following manner: First we compute the average utility of state x, μ, under the assumption that the current policy and stochastic action selector will be used throughout the future (Step 5). *Prob* is the probability distribution function for action selection; it is Equation 4 in this work. Next, the policy network is modified to increase/decrease the merits of actions which are above/below the average (Step 7).

The algorithm for Framework AHCOM-M is similar to that for AHCON, except that relaxation planning may take place either before Step 2 or after Step 3 (in Figure 2)—in the latter case the agent will be more reactive, while in the former case the agent may make better action choices because it can benefit directly from the one-step look-ahead. To be efficient, relaxation planning is performed selectively (Steps 2 & 3 in Figure 5). The idea is this: For situations where the policy is very decisive about the best action, relaxation planning is not needed. If the policy cannot be very sure about which is the best action, relaxation planning is performed. In this way, at the beginning of learning, all actions are equally good and relaxation planning is performed frequently. As learning proceeds, relaxation planning is performed less and less often. (In this study, promising actions are those whose probability of being chosen is greater than 2%.)

1. $x \leftarrow$ current state; $e \leftarrow eval(x)$;
2. Select promising actions S according to $policy(x)$;
3. If there is only one action in S, go to 8;
4. For $a \in S$ do
   4.a. Simulate action a; $(y, r) \leftarrow$ predicted new state and reinforcement;
   4.b. $E_a \leftarrow r + \gamma \cdot eval(y)$;
5. $\mu \leftarrow \sum_{a \in S} Prob(a) \cdot E_a$; $max \leftarrow Max\{E_a \mid a \in S\}$;
6. Adjust evaluation network by backpropagating error $(max - e)$ through it with input x;
7. Adjust policy network by backpropagating error Δ through it with input x, where $\Delta_a = \begin{cases} E_a - \mu & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$;
8. exit.

Figure 5. Relaxation planning algorithm for AHCON-M. *Prob* is the probability function for stochastic action selection.

2.4.2 *Framework QCON-M*

Framework QCON-M is Framework QCON plus using action models. The algorithm is similar to that in Figure 4. The main difference is that before Step 2 or after Step 3, the agent can perform relaxation planning by looking one step ahead (see Figure 6). For the same reason that it is harmful for QCON-R to replay non-policy actions (Section 2.3), experiencing bad actions with a model is also harmful.[6] Therefore, relaxation planning must be performed selectively. For situations where the best action is obvious, no look-ahead is needed. If there are several promising actions, then these actions are tried with the model. QCON-M is similar to Sutton's Dyna-Q architecture except that only the currently visited state is used to start hypothetical experiences. (Again, in this study, promising actions are those whose probability of being chosen is greater than 2%.)

1. $x \leftarrow$ current state; for each action i, $U_i \leftarrow util(x, i)$;
2. Select promising actions S according to U;
3. If there is only one action in S, go to 6;
4. For $a \in S$ do
   4.a. Simulate action a; $(y, r) \leftarrow$ predicted new state and reinforcement;
   4.b. $U'_a \leftarrow r + \gamma \cdot Max\{util(y, k) \mid k \in actions\}$;
5. Adjust utility network by backpropagating error ΔU through it with input x, where $\Delta U_a = \begin{cases} U'_a - U_a & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$;
6. exit.

Figure 6. Relaxation planning algorithm for QCON-M.

2.5. Teaching: Frameworks AHCON-T and QCON-T

Reinforcement learning is a trial-and-error process. The success of learning relies on the agent's luck in achieving the goal by chance in the first place. If the probability of achieving the goal by chance is arbitrarily small, the time needed to learn will be arbitrarily long (Whitehead, 1991b). This learning barrier will prevent agents from shortening learning time dramatically on their own. One way (probably the best way) to overcome the barrier is to learn expertise directly from external experts.

Teaching plays a critical role in human learning. Very often teaching can shorten our learning time, and even turn intractable learning tasks into tractable. If learning can be viewed as a search problem (Mitchell, 1982), teaching, in some sense, can be viewed as external guidance for this search. Teaching is useful for two reasons: First, teaching can direct the learner to first explore the promising part of the search space which contains the goal states. This is important when the search space is large and thorough search is infeasible. Second, teaching can help the learner avoid being stuck in local maxima. A real example of this is presented in Section 6.4.

While learning from experts is an efficient way to learn, experts may not be available, or may be available only part of the time. So, after having overcome the most formidable learning barrier with the help of experts, the learners must be able to work out the rest on their own. As will be shown below, teaching can be easily and gracefully integrated into both AHC- and Q-style of reinforcement learning. The frameworks to be discussed below are able to learn from teachers as well as through reinforcement learning.

Frameworks AHCON-T and QCON-T, which are AHCON-R and QCON-R plus teaching, use exactly the experience replay algorithms for AHCON-R and QCON-R, respectively. Teaching is conducted in the following manner: First, the teacher shows the learning agent how an instance of the target task can be achieved from some initial state. The sequence of the shown actions as well as the state transitions and received reinforcements are recorded as a *taught lesson*. Several taught lessons can be collected and repeatedly replayed the same way *experienced* (i.e., self-generated) *lessons* are replayed. Like experienced lessons, taught lessons should be replayed selectively; in other words, only policy actions are replayed. But if the taught actions are known to be optimal, all of them can be replayed all the time. In this paper, the term *lesson* means both taught and experienced lessons.

It is unnecessary that the teacher demonstrate only optimal solutions in order for the agent to learn an optimal policy. In fact, the agent can learn from both positive and negative examples. This property makes this approach to teaching different from the *supervised learning* approaches (e.g., (Mozer, 1986; Pomerleau, 1989)). In the supervised learning paradigm, a learning agent tries to mimic a teacher by building a mapping from situations to the demonstrated actions and generalizing the mapping. The consequent drawbacks are that (1) the teacher is required to create many many training instances to cover most of the situations to be encountered by the agent, (2) when a new situation is encountered and the agent does not have a good strategy for it, a teacher must be available to give the solution, and (3) the teacher must be an expert, or else the agent cannot learn the optimal strategy. The third drawback is often neglected by researchers, but it is important when humans want to build robots using supervised learning techniques. Since humans and robots have different sensors and hence see different things, an optimal action from a human's

point of view may not be optimal for robots if we take into account the fact that robots may not be able to sense all the information that humans use to make decisions. Human teachers must teach in terms of what robots can sense.

Obviously, Frameworks AHCON-T and QCON-T do not have the first and second drawbacks. They also do not have the third drawback, because they do not learn by rote. Instead, they determine the real utilities of the shown actions—If a shown action results in a state which the agent **now** believes to be good/bad, then the agent increments/decrements the likelihood of performing that action given the same situation in the future. On the other hand, typically learners are going to benefit from experts more than from naive teachers.

3. A dynamic environment

To date, reinforcement learning has been studied mostly for solving simple learning tasks, such as pole-balancing, route-finding, etc. (Some researchers, for instance, Mahadevan and Connell (1991) and Chapman and Kaelbling (1991), have begun to study more complicated problems, however.) It is therefore unclear whether reinforcement learning can scale up to deal with more realistic problems. One of the goals of this work is thus to study the utility of various frameworks in solving nontrivial learning problems. To this end, I devised a simulated dynamic environment to be used as a testbed for evaluating the performance of various frameworks.

The dynamic environment is a 25 × 25 cell world. A sample environment is shown in Figure 7.[7] There are four kinds of objects in the environment: the agent ("I"), food ("$"), enemies ("E"), and obstacles ("O"). The perimeter of the world is considered to be occupied by obstacles. The bottom of the figure is an energy indicator ("H"). At the start, the agent and four enemies are placed in their initial positions as shown in Figure

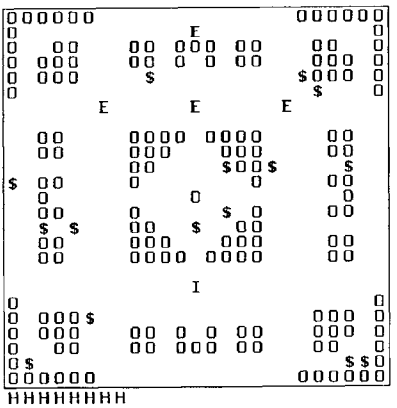


Figure 7. A dynamic environment involving an agent (I), enemies (E), food ($), and obstacles (O). The H's indicate the agent's energy level.

7, and fifteen pieces of food are randomly placed on unoccupied cells. On each move, the agent has four actions to choose from; it can walk to one of the four adjacent cells. If the agent attempts to walk into obstacles, it will remain at the same position.

After the agent moves, each of the enemies is allowed to stay or move to an adjacent cell that is not occupied by obstacles. To allow the agent to escape from chasing enemies, the enemy speed is limited to 80% of the full speed of the agent. The enemies move randomly, but tend to move towards the agent; the tendency becomes stronger as the agent gets closer. Appendix A gives the algorithm for choosing enemy actions.

The agent has two goals: to get as much food as possible and to avoid being caught by enemies. Note that the two goals conflict in some situations, and the agent must learn to arbitrate between them. A play ends when the agent gets all of the food or dies. The agent dies when it either collides with an enemy or runs out of energy. At the start of a new play, the agent is given 40 units of energy. Each piece of food provides the agent 15 units of additional energy, and each move costs the agent 1 unit. These parameter values were empirically chosen so that survival in the environment would not be too easy or too difficult.

To make the learning task more interesting and realistic, the agent is allowed to see only a local area surrounding it. From the agent's point of view, the world is nondeterministic, not only because the enemies behave randomly, but also because the world is only partially observable (thus, what will be seen after a move is not completely predictable). Although a human player can avoid the enemies and get all of the food most of the time, survival in this environment is not trivial. To survive, the agent must learn to 1) approach food, 2) escape from enemies, 3) avoid obstacles, 4) identify and stay away from certain dangerous places, (e.g., corridors), where it can be easily seized, and 5) seek food when food is out of sight.

4. The learning agents

This section presents the implementation of four AHC-agents (i.e., Agents AHCON, AHCON-R, AHCON-M, and AHCON-T) and four Q-agents (i.e., Agents QCON, QCON-R, QCON-M, and QCON-T), which learn to survive in the environment. The agents are named after the learning frameworks on which they are based. In order to compare their performance, the agents use exactly the same reinforcement signals and sensory inputs as described below.

All of the connectionist networks are trained using a symmetrical version of the error backpropagation algorithm; the squashing function is a variation of the sigmoid function: $f(x) = 1/(1 + e^{-x}) - 0.5$. (Note: Once the agents die, they get re-incarnated to try again, and in the meanwhile their learned networks are preserved.)

4.1. The reinforcement signals

After each move, the learning agent receives from the environment one of the following reinforcement signals:

- −1.0 if the agent dies,
- 0.4 if the agent gets food,
- 0.0 otherwise.

Negative reinforcement is considered bad, and positive is good. The food reward is smaller than the penalty for being dead, since it is more important to stay alive. The food reward, 0.4, was chosen empirically; I tried a few different values for Agent AHCON, and 0.4 gave roughly the best performance. Theoretically speaking, the agent should learn to get food even without food reward, because the agent is required to get food to stay alive. But the sooner good action decisions are rewarded, the sooner the agent should learn.

4.2. Input representation

As described before, AHC-agents have evaluation and policy networks, while Q-agents have utility networks. This subsection describes the input representation for the networks, while another subsection describes the output representation.

The evaluation, policy and utility networks are all structurally similar; each of them is a three-layer, feed-forward network, consisting of 145 input units and 1 output unit. (The number of hidden units is a parameter to be tuned for performance.) The networks are fully connected except that there are no connections between the input and output units. The input units of the networks can be divided into five groups: enemy map, food map, obstacle map, energy level, and some history information. Each of the maps shows a certain kind of object in a local region surrounding the agent and can be thought of as being obtained by an array of sensors fixed on the agent. The sensor array moves as the agent moves. If the local action representation (see Section 4.3) is used, the array also rotates as the agent rotates. Figure 8 shows the configuration of the food, enemy and obstacle sensor arrays.

Each food sensor may be activated by several nearby food objects and each food object may activate several nearby food sensors. The food sensor array is composed of three different

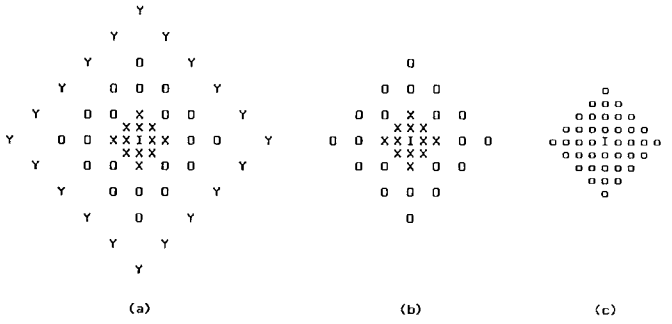


Figure 8. The (a) food, (b) enemy, and (c) obstacle sensor arrays.

types of sensors, types "X", "O", and "Y". Different food sensor types have different resolution and different receptive fields—"X", "O", and "Y" sensors can be activated by food at any of the five, nine, and thirteen nearby cells, respectively. This technique of encoding spatial positions of objects is known as *coarse coding* (Hinton et al., 1986). Through the use of multiple resolution and coarse coding techniques, the food positions are effectively coded without loss of critical information.

The enemy sensor array has a similar layout of the food sensor array, except that it consists of only "X" and "O" types of sensors. The obstacle sensors are organized differently; there is only one type of obstacle sensor, and each obstacle sensor is only activated by an obstacle at the corresponding cell. The coarse coding technique is not used to encode obstacle positions, because it only works effectively when the features to be encoded are sparse (Hinton et al., 1986), and this is not the case for obstacles.

The agent's energy level is, again, coarse-coded using sixteen input units. Each of the sixteen units represents a specific energy level and is activated when the agent's energy level is close to that specific level. Finally, four units are used to encode the agent's previous action choice and one unit to indicate whether or not the previous action resulted in a collision with an obstacle. These five units convey a kind of history information, which allows the agent to learn heuristics such as "moving back to previous positions is generally bad," "when no interesting objects (e.g., food) are around, keep moving in the same direction until you see something interesting," etc.

4.3. Action representation

As mentioned before, the agent has four actions. This work experimented with two different representations of agent actions:

- **Global representation:** The four actions are to move to the north, south, west and east, regardless of the agent's current orientation in the environment.
- **Local representation:** The four actions are to move to the front, back, right and left relative to the agent's current orientation.

If the local representation is used, the agent begins with a random orientation for each new play, and its orientation is changed to the move direction after each step. For instance, if the agent takes four consecutive "right" moves and no collision occurs, it will end up being at the same cell with the same orientation as before.

4.4. Output representation

The single output of the evaluation network represents the estimated utility of the given input state. Corresponding to the action representations, there are two different output representations for policy and utility networks.

Global representation: With this action representation, the agent uses only one policy network, whose single output represents the merit of moving to the "north." The merits of moving in other directions can be computed using the same network by rotating the state inputs (including food map, enemy map, obstacle map, and the 4 bits encoding the agent's previous action) by 90, 180, and 270 degrees. Similarly, the agent uses only one utility network, whose single output represents the utility of moving to the "north" in response to the input state. Again, the utilities of moving in other directions can be computed by rotating the state inputs appropriately. By taking advantage of the action symmetry, the learning task is simplified, because whatever is learned for a situation is automatically carried over to situations which are more or less symmetric to it.

Local representation: This representation does not take advantage of action symmetry. Each AHC-agent uses four policy networks and each Q-agent four utility networks. Each network has a single output.

The outputs of all the networks are between -1 and $+1$. By Definition 1, the *eval* and *util* functions may be greater than 1 when the agent is close to many pieces of food. In such (rare) cases, they are truncated to 1 before being used to compute TD errors. Also, the output units of the evaluation and utility networks use mainly the linear part of the sigmoid function. An alternative is to use a truly linear activation function for the output units.

4.5. The action model

Agents AHCON-M and QCON-M learn an action model. The action model is intended to model the input-output behavior of the dynamic environment. More specifically, given a world state and an action to be performed, the model is to predict what reinforcement signal will be received, where the food, enemies and obstacles will appear, and what the agent's energy level will be. In this nondeterministic environment, each action can have many possible outcomes. There are two alternatives to modeling the environment: the action model can generate either a list of outcomes associated with probabilities of happening or only the most likely outcome. The second alternative is adopted, because of its simplicity.

Since the food and obstacles do not move, their positions were found to be quite easy to predict using connectionist networks. So, to shorten simulation time without significantly simplifying the model-learning task, Agents AHCON-M and QCON-M were only required to learn *reinforcement networks* for predicting the immediate reinforcement signal and *enemy networks* for predicting the enemy positions. The reinforcement and enemy networks are all two-layer networks (i.e., without hidden layers). The reinforcement networks use as inputs all of the 145 input bits mentioned before, while the enemy networks use only the enemy and obstacle maps. The single output of each enemy network is trained to predict whether a particular enemy sensor will be turned on or off after the agent moves.

The reinforcement and enemy networks are learned on-line just like the other networks. Learning these networks is a kind of supervised learning, and the encountered experiences (i.e., x, a, y, r) can be saved in a queue (of limited length) for repeated presentation to the networks. Because the enemies are nondeterministic and the agent does not know the exact enemy positions due to coarse coding, the prediction of enemy positions was found to be often incorrect even after the networks were well-trained. Since it is also interesting

to see what performance the agent will have if a perfect model can be learned quickly, AHCON-M and QCON-M are also provided with a "perfect" model, which is just the environment simulator. Section 6.3 presents a performance comparison between using a perfect model and using a learned, potentially incorrect one.

4.6. Active exploration

An important problem often faced by autonomous learning agents is the tradeoff between acting to gain information and acting to gain rewards. Kaelbling (1990) proposed an *interval estimate* (IE) method for control of this tradeoff. However, it is unclear whether the method can apply to connectionist reinforcement learning. Thrun and Möller (1992) proposed to have the agent learn a *competence map*, which estimates the errors of what the agent learned about the world. The map should display small errors for well-explored world states and large errors for rarely-explored states. Active exploration is achieved by driving the agent into regions of large errors.

As mentioned before, this work uses a stochastic action selector as a crude strategy to allow active exploration. It was found that the stochastic selector alone did not give a good compromise between gaining information and gaining rewards. A complementary strategy is also used: the agent dead-reckons its trajectory and increases the temperature T whenever it finds itself stuck in a small area without getting food. A similar strategy is also used for some robot learning tasks (Lin, 1991c). For both domains, this strategy was found quite effective in improving learning speed and quality.

4.7 Prevention of over-training

As mentioned in Section 2.3, undesired over-training may occur if the same experiences are replayed too many times, no matter whether the experiences come from a teacher or from trial-and-error. To reduce the possibility of over-training and also to save computation, the following heuristic strategies are employed:

- After each complete play, the agent replays n lessons chosen randomly from the most recent 100 experienced lessons, with recent lessons exponentially more likely to be chosen. n is a decreasing number between 12 and 4. Appendix B gives the algorithm for choosing experiences stochastically.
- After each complete play, the agent (if it is AHCON-T or QCON-T) also stochastically chooses taught lessons for replay. Each of the taught lessons is chosen with a decreasing probability between 0.5 and 0.1.

Recall that only policy actions are replayed. Policy actions are those whose probability of being chosen is greater than $P_l = 0.2$ for AHCON-type agents or $P_l = 0.01$ for Q-type agents. (Since the taught lessons I gave to the agents were nearly optimal, P_l was in fact set to 0 during replaying taught lessons.)

5. Experimental results

This section presents the performance of various learning agents. In the first study, the learning agents used the global action representation and took advantage of action symmetry. In the second study, the agents used the local action representation and did not exploit action symmetry.

5.1. Experimental design

Each study consisted of 7 experiments. (Each experiment took a Sparc Station two days to complete.) For each experiment, 300 training environments and 50 test environments were randomly generated. The agents were allowed to learn only from playing the 300 training environments. Each time an agent played 20 training environments, it was tested on the 50 test environments with learning turned off and the temperature T set to zero. The average number of food pieces obtained by the agent in the test environments was then plotted versus the number of training environments that the agent had played so far. The learning curves presented below show the mean performance over all the experiments.

It is instructive to mention the difference between the objective of the learning algorithms and that I set up for the agents. The former is to maximize the discounted cumulative reinforcement, while the latter is to have as much food as possible. Achieving the former objective does not necessarily end up achieving the latter. However, given the reinforcement function defined in Section 4.1, it is unlikely that both objectives are quite different.

For both studies, two lessons were generated for Agents AHCON-T and QCON-T to use. To generate the lessons, I pretended to be the agent trying to survive in a manually-chosen environment, which involved a few instructive situations. I got all the food in each of the lessons.

Each learning agent has several parameters that can be tuned for performance:

- γ: the discount factor (fixed to be 0.9);
- T: the temperature for the stochastic action selector;
- H_e, H_p and H_u: the number of hidden units of the evaluation, policy and utility networks;
- η_e, η_p and η_u: the learning rate of the backpropagation algorithm for the evaluation, policy and utility networks;
- the momentum factor of the backpropagation algorithm (fixed to be 0.9 for all networks);
- the range of the random initial weights of networks (fixed to be 0.1).

5.2. Study 1: Using global action representation

In this study, the agents used the global action representation and exploited action symmetry. Table 1 shows the parameter settings used to generate the mean learning curves shown in Figure 9.[8] Those parameter values were empirically chosen to give roughly the best performance for Agents AHCON and QCON. Little search was done for the other agents. AHC-agents used a temperature much lower than that used by Q-agents, because action

Table 1. Parameter values used for Study 1.

Agent	H	η	T
AHCON	$H_e = 30$ $H_p = 30$	$\eta_e = 0.2$ $\eta_p = 0.4$	$1/T = 2 \rightarrow 10$
Other AHC-agents	$H_e = 30$ $H_p = 30$	$\eta_e = 0.1$ $\eta_p = 0.2$	$1/T = 2 \rightarrow 10$
QCON	$H_u = 30$	$\eta_u = 0.3$	$1/T = 20 \rightarrow 60$
Other Q-agents	$H_u = 30$	$\eta_u = 0.15$	$1/T = 20 \rightarrow 60$

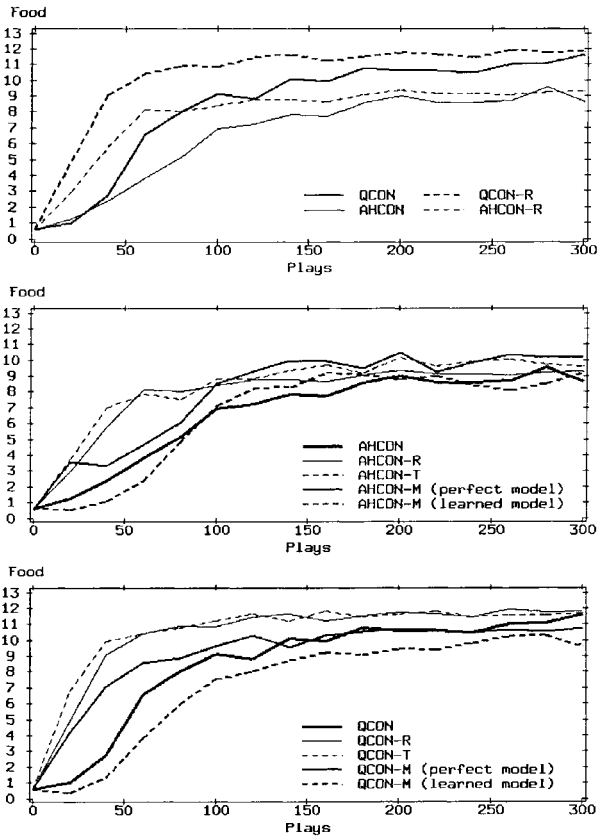


Figure 9. Learning curves of the agents using the global action representation.

merits are supposed to approach 1 for the best action(s) and −1 for the others, while utilities of state-action pairs are usually small numbers between 1 and −1. Cooling temperatures were used, although fixed low temperatures seemed to work as well. The learning agents with learning-speedup techniques used smaller learning rates than those used by the basic learning agents, because the former trained their networks more often.

5.3. Study 2: Using local action representation

In this study, the agents used the local action representation and did not exploit action symmetry. The agents used the same parameter settings as in Study 1, except that all the learning rates were doubled. Figure 10 shows the learning curves of the agents.

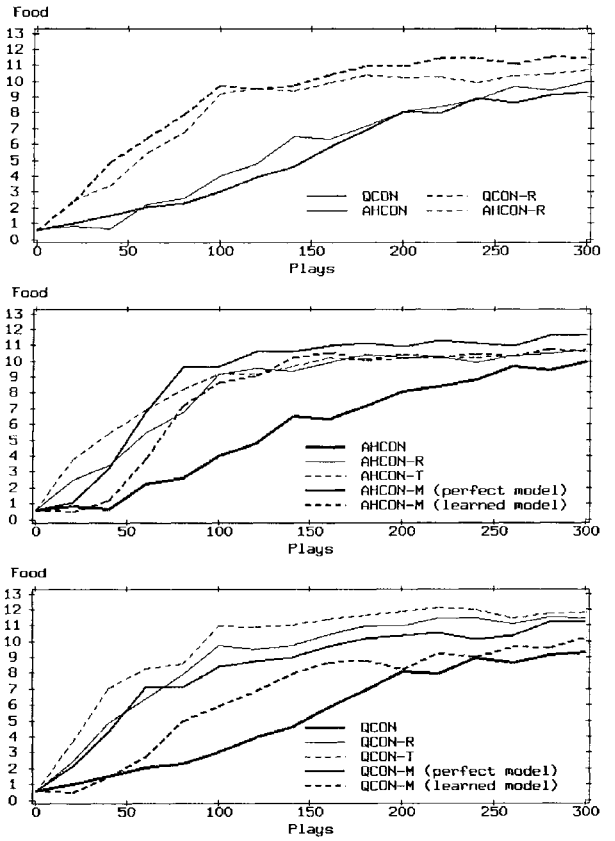


Figure 10. Learning curves of the agents using the local action representation.

5.4. More observation

Among all the learning agents, QCON-T was roughly the best one. To see the absolute performance of the best agent after 300 trials, QCON-T was tested on 8000 randomly generated environments. The following table shows how often the agent got all the food, got killed, or ran out of energy.

got all food	got killed	ran out of energy
39.9%	31.9%	28.2%

In average the agent got 12.1 pieces of food in each play. The table below shows how often the agent got no food, 1 piece of food, more pieces, or all of the 15 pieces.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
%	0.1	0.3	0.8	1.8	2.2	2.9	4.0	4.1	3.8	3.7	3.4	4.1	5.4	8.2	15.2	39.9

As mentioned previously, the learning curves in Figures 9 and 10 show the mean performance over 7 experiments. The standard deviations of these curves, point by point, are roughly 2 food pieces at the beginning of learning and 1 food piece after the asymptotic performance is reached. Generally speaking, the agents who had better performance had smaller performance deviations between experiments. But an agent's performance difference between experiments did not imply much about the repeatability of experimental results, because each experiment used a different set of training and test environments. What really matters is the relative performance of agents, which was found consistent most of the time through each entire experiment.

6. Discussion

This section compares the learning agents by analyzing the experimental results in Section 5.

6.1. AHC-agents vs. Q-agents

In the first study, the asymptotic performance of Q-agents was significantly better than that of AHC-agents. A number of previous studies (Lin, 1991a; Sutton, 1990) also found the superiority of Q-learning over AHC-learning. But Study 2 did not confirm this superiority; in fact both types of learning were found to be similarly effective in terms of both asymptotic performance and learning speeds. It is still an open question whether Q-learning works better than AHC-learning in general.

Note also that the two different action representations did not make a significant difference for Q-agents. But for AHC-agents, the local action representation apparently worked better than the global action representation, as far as asymptotic performance is concerned. This result reveals that different action representations, as well as different input representations, may make a big difference in the performance of a learning system.

6.2. Effects of experience replay

Compared with AHCON (or QCON), the learning speed of AHCON-R (or QCON-R) was significantly improved. The difference in their asymptotic performance, however, was not found to be significantly different after 300 trials in Study 1, and seemed to get less significant in Study 2 if learning continued. This concludes that experience replay is indeed an effective way to speed up the credit assignment process. Experience replay is also very easy to implement. The cost of using it is mainly the extra memory needed for storing experiences. To prevent over-training and to be efficient, a learning agent also needs a good strategy for determining what experiences to remember and how often to replay them.

6.3. Effects of using action models

The advantage of using action models for relaxation planning was inconclusive. On the one hand, compared with AHCON (or QCON), the learning speed of AHCON-M (or QCON-M) was significantly improved when the agent did not need to learn an action model on its own (i.e., a perfect model was provided). On the other hand, when the agent needed to learn a model itself, whether using a model was going to help depended on the relative difficulty of learning the model and learning the survival task directly. In the second study, learning the model was found to be faster than learning the task. Therefore once a sufficiently good model had been learned, Agents AHCON-M and QCON-M began to benefit from using it. On the other hand, in the first study AHCON and QCON already could learn the task quite rapidly. Therefore a potentially incorrect model turned out to be only misleading, at least in the beginning. As a matter of fact, QCON-M with a learned model performed worse than QCON in Study 1, which can be explained by the fact that the agent was not able to learn a perfect model for the nondeterministic world, even after hundreds of trials.

As speculated in Section 2.4, relaxation planning should be less and less needed as learning proceeds. This speculation was confirmed in both studies. Figure 11 shows the average number of hypothetical actions that were taken on each step. (The curves show the mean

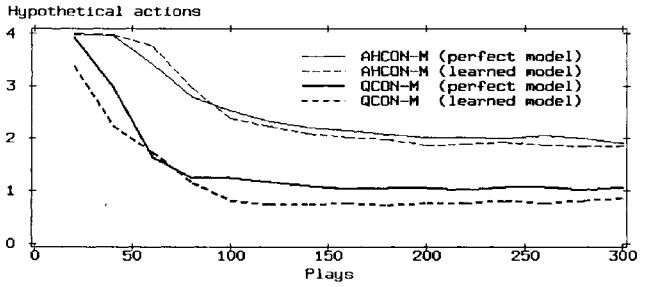


Figure 11. Average number of hypothetical actions taken on each step (from Study 2).

results from the 7 experiments of Study 2.) The maximum is 4, since there are 4 actions to choose from on any step. As we can see from the figure, the amount of planning dropped very quickly as learning approached the asymptote. But it did not drop close to zero at the end, because in this task there are many situations where several actions are in fact almost equally good and therefore they all end up being tried in simulation. For tasks where only a small portion of actions are relevant most of the time, the saving should be more significant.

6.4. Effects of teaching

Recall that AHCON-T (or QCON-T) is AHCON-R (or QCON-R) plus replaying taught lessons. In the first study, the performance differences between AHCON-T and AHCON-R and between QCON-T and QCON-R were negligible over the entire experiment. Since the task in Study 1 was not very difficult and experience replay alone had done a good job, there was not much improvement we could expect from using teaching. But in the second study, there were noticeable differences between the learning speeds of AHCON-R and AHCON-T and between those of QCON-R and QCON-T. The differences between their asymptotic performance, however, were insignificant.

This result reveals that the advantage of teaching should become more significant as the learning task gets more difficult. Indeed, Lin (1991c) reported similar results from a learning robot. One of the robot's tasks is to dock on a battery charger, using sonar sensors for detecting obstacles and light intensity sensors for locating the charger, which has a light on top of it. To dock, the robot must position itself in front of the charger within small errors and then drive to collide with the charger in order to get a tight connection. Without a teacher, the robot failed to learn the task, because at the early stage of learning, the robot quickly learned to avoid obstacles, which in turn prevented it from colliding with the charger (the learning was stuck in a local maximum). After being shown a lesson (i.e., how to dock from one sample position), the robot became able to learn the task. After being shown four lessons, the robot learned the task rapidly.

6.5. Experience replay vs. using action models

Comparing the learning curves of Agents AHCON-M and AHCON-R and those of QCON-M and QCON-R, there was clear, consistent superiority of experience replay over using action models, when the agents had to learn a model themselves. (But there was no clear, consistent superiority of one over the other, when the agents were provided with a perfect action model.) This result can be simply explained by the fact that it took some time for Agents AHCON-M and QCON-M to learn a sufficiently good model before the agents could start taking advantage of it. Before a good model was learned, the model could be misleading and impede learning. (As a matter of fact, AHCON-M and QCON-M did not perform relaxation planning for the first 10 trials, avoiding using a completely senseless model.)

Why couldn't relaxation planning with a perfect action model outperform experience replay? First of all, the relaxation planning algorithms used here may not be the most effective

way of using action models. For example, the relaxation planning algorithms used here performed just one-step look-ahead. Also, the number of hypothetical experiences that AHCON-M and QCON-M could generate was limited by the number of states actually visited. (On the other hand, if the agents are allowed to generate hypothetical experiences from any randomly chosen states, they may spend too much time planning what to do about hypothetical situations that will never happen at all.)

Secondly, *experience replay is also a kind of relaxation planning*! Experience replay uses an "action model," but does not need to learn one; an action model is simply obtained by sampling past experiences. In essence, the collection of past experiences is a model. It represents not only explicitly the environment's input-output patterns but also implicitly the probability distributions of multiple outcomes of actions.

Since experience replay is so effective and also easy to implement, is it of no use to learn an action model? The answer is unclear. If a model is learned *merely* for doing relaxation planning, perhaps it is *not* worthwhile to learn a model, since experience replay does the same thing as relaxation planning. One may argue that: 1) a model that generalizes can provide the learning algorithms with induced or interpolated experiences, which are not seen before, and 2) the extra experiences will result in better learning of the evaluation functions. But if the evaluation functions also generalize (as it should be the case for non-toy tasks), it is unclear whether these extra experiences can actually do any extra good.

On the other hand, having an action model can be useful. In this work I only investigated how to use action models for learning the evaluation functions. There are other ways of using a model to improve performance that I did not study here. For example, we can use an evaluation function, an action model, and a look-ahead planning technique to help find the best actions (Whitehead & Ballard, 1989; Thrun et al., 1991). In a complex domain where an optimal policy may be hardly obtainable, by looking ahead a few steps (much as computer chess does), the non-optimality of a policy can be compensated, if an accurate action model is available. How to use action models effectively is an interesting issue and needs further study.

6.6. Why not perfect performance?

Why did all the agents fail to reach the perfect performance (i.e., get all the food)? There are at least two reasons: 1) The local information used by the agents may be insufficient to determine the optimal actions, and 2) the perfect policy may be too complex to be represented by the connectionist networks used here. As a matter of fact, I also played against the simulator myself. Being allowed to see only objects in the local region as the learning agents do, I got all the food pieces most (but not all) of the time. I found that I often employed two techniques to play it so successfully: look-ahead planning (although very crude) and remembering food positions so that I could come back to get the food after I lost the sight of food because of chasing enemies. The learning agents did not use either of the two techniques.

7. Limitations

While it seems that all of these learning frameworks are promising, they have several common limitations.

Representation dependent. As in any other learning system, a good input representation is critical to successful learning. Because of the use of coarse coding and concentric multiple resolution maps, the number of input units needed for the task was dramatically reduced. Without using these techniques, the agents could not have been so successful. Choosing a good action representation is also crucial sometimes. For example, the local action representation gave AHC-agents better performance than the global action representation.

Discrete time and discrete actions. So far TD methods have been used almost exclusively in domains with discrete time and actions. While it seems possible to extend the idea of temporal difference to handle continuous time and actions, it is not clear how easily this extension can be done. But note that these frameworks can work for continuous states, since connectionist networks take real as well as binary numbers as inputs.

Unwise use of sensing. The learning agent is required to sense the entire environment at each step to determine what world state it is in. In the real world, there are too many things to look at. Most of them may be irrelevant. In practice, an agent often cannot afford to sense everything all the time. It is an important issue to decide how to use sensors efficiently while still knowing what the current state is (Whitehead & Ballard, 1991a; Tan, 1991).

History insensitive. Unlike humans who usually make decisions based on information which is currently sensed and information which is in the past and cannot be sensed, the learning agent is only reactive to what is perceived at the current moment, but insensitive to what was perceived in the past. One possibility of becoming history-sensitive is to use *time-delay networks* (Lang, 1989) or *recurrent networks* (Williams & Zipser, 1988).

Perceptual aliasing. The *perceptual aliasing problem* occurs when the agent's internal state representation is insufficient to discriminate different external world states. As a consequence of perceptual aliasing, the learning agent cannot learn a correct evaluation function and therefore cannot act optimally. Consider a packing task which involves three steps: open a box, put an object in the box, and close the box. An agent driven only by its current visual percepts cannot accomplish this task, because facing a closed box, the agent does not know whether an object is already in the box or not. There are two ways to resolve this problem. The first solution is to actively choose perceptual actions, such as measuring the weight of the box, to resolve the ambiguity. This kind of solutions have been investigated by Whitehead and Ballard (1991a) for a block-stacking problem and by Tan (1991) for a route-finding problem. The second solution is to use history information, such as whether the box was ever opened or not, to help determine the current state of the world. This solution seems more general and more powerful than the first one. To deal with complex real world problems, perhaps both kinds of solutions are needed.

No hierarchical control. In theory, TD methods are able to accurately propagate credit through a long sequence of actions. However, this can be true only when the evaluation function can be modeled to any arbitrary accuracy, for example, using look-up tables. In practice, more compact function approximators that allow generalization, such as connectionist networks and decision trees, should be used. By using such approximators, TD methods can hardly propagate credit accurately through an action sequence longer than,

say, 20. Furthermore, the longer the sequence is, the more time is needed for credit propagation. A potential solution to this problem is to use *hierarchical control*. By hierarchical control, a top-level task is decomposed into subtasks, each subtask is learned separately using reinforcement learning, and finally a top-level policy is learned to control the invocation of the subtasks (Lin, 1991c; Mahadevan & Connell, 1991).

8. Conclusion

This paper studied connectionist AHC- and Q-learning for a moderately complex task—survival in a dynamic environment. The results are encouraging and suggest that reinforcement learning is a promising approach to building autonomous learning systems. Some previous studies (Lin, 1991a; Sutton, 1990) found the superiority of Q-learning over AHC-learning. This work confirmed this finding in one case and found comparable performance of both methods in the other. More studies will be required before it is possible to formulate the conditions for which AHC- or Q-learning will be more effective than the other.

Reinforcement learning algorithms often converge slowly. This paper investigated three extensions for speedup: experience replay, learning action models for relaxation planning, and teaching. Relaxation planning, which is a kind of incremental dynamic programming (Watkins, 1989), caches the results of repeated shallow searches in an evaluation function, using an action model for looking ahead. Learning an action model for planning is a well-known idea. But whether using a model is going to speed up learning depends on the relative difficulty of learning the model and learning to solve the task directly. In this study, learning a good model for the nondeterministic, dynamic world turned out to be difficult, and this has limited the utility of using a model in some cases.

By sampling past experiences, experience replay performs a process much like relaxation planning, but does not need to learn an action model. The key point is to replay only policy actions. This study found that experience replay was quite effective in speeding up the credit assignment process. As a matter of fact, experience replay was found to work better than relaxation planning with a learned model; after all, learning a model takes time. So, perhaps it is *not* worthwhile to learn a model, if a model is learned *merely* for doing relaxation planning. But an action model is still a good thing to have, because a model together with an evaluation function can be utilized, for instance, to perform conventional look-ahead search for optimal actions when an optimal policy is not available.

This paper also described an approach to integrating teaching with reinforcement learning. To learn to solve a problem by reinforcement learning, the learning agent must achieve the goal (by trial-and-error) at least once. Teaching can be an effective technique to shorten the trial-and-error process by simply providing some success examples to the learner. Teaching can also help the learner avoid being stuck in local maxima (for instance, Section 6.4). Although teaching was not found to be critical to learning the survival task in this work, I expect teaching to be more and more important as tasks get more complicated and rewards get less likely to obtain by luck (Lin, 1991c).

Acknowledgments

I would like to express my gratitude to Tom Mitchell and Rich Sutton for many fruitful discussions on this work. Thanks also to Jeffrey Schlimmer, Nils Nilsson, the reviewers, and Sebastian Thrun for their valuable comments. This work was supported partly by NASA under Contract NAGW-1175 and partly by Fujitsu Laboratories Ltd.

Notes

1. Normally the agent's internal description of the world is obtained from a vector of sensory readings, which may or may not be pre-processed. Here it is assumed that the agent's input adequately represents the external world and is sufficient to determine the optimal actions. Reinforcement learning often works more or less, even when this assumption is relaxed.
2. See (Lin, 1991c); Watkins, 1989; Dayan, 1992) for using TD(λ), $\lambda > 0$.
3. For simplicity, I will assume the world is deterministic throughout the discussion of the learning algorithms, although it is straightforward to extend the discussion to handle nondeterministic worlds (Barto et al., 1990; Watkins, 1989). The reinforcement learning algorithms presented in the paper can work for deterministic and nondeterministic worlds.
4. Another way to reduce the negative effects of AHCON-R replaying non-policy actions is to treat an action being a policy action as a matter of probability. In other words, the computed TD error, which is used to adjust the networks, is weighted by the probability of choosing the replayed action. This method, however, did not work as effectively as the thresholding method (i.e., using P_l).
5. Instead of using the term *world model* as Sutton did, I decided to use the term *action model* to refer to a model for predicting the effects of actions. I will use the term *world model* to mean an agent's internal representation of the world. For example, knowledge about the next-state function is referred to as action model, while a cognitive map about a familiar environment is referred to as world model.
6. It seems that AHCON-M does not have this problem, since at each planning step, only the most promising action is used to determine the amount of change to $eval(x)$.
7. It may appear that the learning task is too simplified by making the obstacles symmetric. That is not all true, because there are also other objects in the world—enemies and food pieces are positioned randomly and with no symmetrical pattern. The number of different input patterns that the agent is likely to come across is estimated to be greater than 2^{50}.
8. My previous studies (Lin, 1991a; Lin, 1991b) reported similar results. There are in fact a few differences between the experimental designs of this study and the previous studies. For example, the action model was different, some of the parameter values were changed, and some implementation details of the learning algorithms were also different. See (Lin, 1991a; Lin 1991b) for more details about the differences.

References

Anderson, C.W. (1987). Strategy learning with multilayer connectionist representations. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 103–114).

Barto, A.G., Sutton, R.S., & Watkins, C.J.C.H. (1990). Learning and sequential decision making. In: M. Gabriel & J.W. Moore (Eds.), *Learning and computational neuroscience*. MIT Press.

Barto, A.G., Bradtke, S.J., & Singh, S.P. (1991). *Real-time learning and control using asynchronous dynamic programming*. (Technical Report 91-57). University of Massachusetts, Computer Science Department.

Chapman, D. & Kaelbling, L.P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *Proceedings of IJCAI-91*.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning*, *8*, 341–362.

Grefenstette, J.J., Ramsey, C.L., & Schultz, A.C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning*, *5*, 355–382.

Hinton, G.E., McClelland, J.L., & Rumelhart, D.E. (1986). Distributed representations. *Parallel distributed processing: Explorations in the microstructure of cognition, Vol. 1*, Bradford Books/MIT Press.

Howard, R.A. (1960). *Dynamic programming and Markov processes*. Wiley, New York.

Kaelbling, L.P. (1990). *Learning in embedded systems*. Ph.D. Thesis, Department of Computer Science, Stanford University.

Lang, K.J. (1989). *A time-delay neural network architecture for speech recognition*. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University.

Lin, Long-Ji. (1991a). Self-improving reactive agents: Case studies of reinforcement learning frameworks. *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats* (pp. 297–305). Also Technical Report CMU-CS-90-109, Carnegie Mellon University.

Lin, Long-Ji. (1991b). Self-improvement based on reinforcement learning, planning and teaching. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 323–327).

Lin, Long-Ji. (1991c). Programming robots using reinforcement learning and teaching. *Proceedings of AAAI-91* (pp. 781–786).

Mahadevan, S. & Connell, J. (1991). Scaling reinforcement learning to robotics by exploiting the subsumption architecture. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 328–332).

Mitchell, T.M. (1982). Generalization as search. *Artificial Intelligence*, 18, 203–226.

Moore, A.W. (1991). Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 333–337).

Mozer, M.C. (1986). *RAMBOT: A connectionist expert system that learns by example*. (Institute for Cognitive Science Report 8610). University of California at San Diego.

Pomerleau, D.A. (1989). *ALVINN: An autonomous land vehicle in a neural network* (Technical Report CMU-CS-89-107). Carnegie Mellon University.

Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning internal representations by error propagation. *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1*. Bradford Books/MIT Press.

Sutton, R.S. (1984). *Temporal credit assignment in reinforcement learning*. Ph.D. Thesis, Dept. of Computer and Information Science, University of Massachusetts.

Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

Sutton, R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Workshop on Machine Learning* (pp. 216–224).

Tan, Ming. (1991). Learning a cost-sensitive internal representation for reinforcement learning. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 358–362).

Thrun, S.B., Möller, K., & Linden, A. (1991). Planning with an adaptive world model. In D.S. Touretzky (Ed.), *Advances in neural information processing systems 3*, Morgan Kaufmann.

Thrun, S.B. & Möller, K. (1992). Active exploration in dynamic environments. To appear in D.S. Touretzky (Ed.), *Advances in neural information processing systems 4*, Morgan Kaufmann.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. Ph.D. Thesis, King's College, Cambridge.

Williams, R.J. & Zipser, D. (1988). *A learning algorithm for continually running fully recurrent neural networks* (Institute for Cognitive Science Report 8805). University of California at San Diego.

Whitehead, S.D. & Ballard, D.H. (1989). A role for anticipation in reactive systems that learn. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 354–357).

Whitehead, S.D. & Ballard, D.H. (1991a). Learning to perceive and act by trial and error. *Machine Learning*, 7, 45–83.

Whitehead, S.D. (1991b). Complexity and cooperation in Q-learning. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 363–367).

Appendix A: The algorithm for choosing enemy actions

Each enemy in the dynamic environment behaves randomly. On each step, 20% of the time the enemy will not move, and 80% of the time it will choose one of the four actions, A_0 (east), A_1 (south), A_2 (west), and A_3 (north), according to the following probability distribution:

$$prob(A_i) = P_i/(P_0 + P_1 + P_2 + P_3)$$

where

$$P_i = \begin{cases} 0 & \text{if } A_i \text{ will result in collison with obstacles} \\ exp(0.33 \cdot W(angle) \cdot T(dist)) & \text{otherwise} \end{cases}$$

$angle$ = angle between the direction of action A_i and the direction from the enemy to the agent

$dist$ = distance between the enemy and the agent

$$W(angle) = (180 - |angle|)/180$$

$$T(dist) = \begin{cases} 15 - dist & \text{if } dist \leq 4 \\ 9 - dist/2 & \text{if } dist \leq 15 \\ 1 & \text{otherwise} \end{cases}$$

Appendix B: The algorithm for choosing lessons for replay

The agents which replay experiences keep only the most recent 100 lessons in memory. Lessons are randomly chosen for replay; recent lessons are exponentially more likely to be chosen. The algorithm for choosing a lesson from memory is shown below.

Input: a sequence of lessons, $L_0, L_1, \ldots, L_{n-1}$, where L_{n-1} is the latest.
Output: an integer k, $0 \leq k < n$
Algorithm:
1. $w \leftarrow Min(3, 1 + 0.02 \cdot n)$
2. $r \leftarrow$ a random number between 0 and 1
3. $k \leftarrow n \cdot log(1 + r \cdot (e^w - 1))/w$

Machine Learning, 8, 323-339 (1992)

Transfer of Learning by Composing Solutions of Elemental Sequential Tasks

SATINDER PAL SINGH SATINDER@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

Abstract. Although building sophisticated learning agents that operate in complex environments will require learning to perform multiple tasks, most applications of reinforcement learning have focused on single tasks. In this paper I consider a class of sequential decision tasks (SDTs), called composite sequential decision tasks, formed by temporally concatenating a number of elemental sequential decision tasks. Elemental SDTs cannot be decomposed into simpler SDTs. I consider a learning agent that has to learn to solve a set of elemental and composite SDTs. I assume that the structure of the composite tasks is unknown to the learning agent. The straightforward application of reinforcement learning to multiple tasks requires learning the tasks separately, which can waste computational resources, both memory and time. I present a new learning algorithm and a modular architecture that learns the decomposition of composite SDTs, and achieves transfer of learning by sharing the solutions of elemental SDTs across multiple composite SDTs. The solution of a composite SDT is constructed by computationally inexpensive modifications of the solutions of its constituent elemental SDTs. I provide a proof of one aspect of the learning algorithm.

Keywords. Reinforcement learning, compositional learning, modular architecture, transfer of learning

1. Introduction

Although building sophisticated autonomous learning systems that operate in complex environments will require learning to perform multiple tasks, most applications of control architectures based on reinforcement learning have focused on single tasks (e.g., Barto et al., 1991; Sutton, 1990; Whitehead & Ballard, 1990). One way to extend reinforcement learning to multiple tasks is to use different modules to learn the solutions of the different tasks. If the tasks are related, learning them separately will waste computational resources, both memory and time. In addition, reinforcement learning methods do not scale efficiently to large tasks, and thus the "learn each task independently" method is only feasible for simple tasks. Discovering faster learning algorithms for single tasks will certainly help, but algorithms that allow transfer of learning across tasks must also play a crucial role in control systems that have to learn to perform multiple tasks. While achieving transfer of learning across an arbitrary set of tasks may be difficult, or even impossible, there are useful classes of tasks where such transfer is achievable. In this paper I focus on one class of such tasks.

In this paper I consider a learning agent that interacts with a dynamic external environment and faces multiple sequential decision tasks. Each task requires the agent to execute a sequence of actions to control the environment,[1] either to bring it to a desired state or to traverse a desired state trajectory over time. In addition to the environment dynamics that define state transitions, such tasks are defined by a payoff function that specifies the

immediate evaluation of each action taken by the agent. The agent's goal is to learn a closed loop control policy,[2] i.e., a function assigning actions to states, that extremizes some measure of the cumulative payoffs received over time. Sequential decision tasks are difficult to solve because the long-term consequences of taking an action are seldom reflected in the immediate payoff, making it difficult to assign credit to actions. Thus, decisions cannot be viewed in isolation because the agent must account for both the short-term and the long-term consequences of its actions. The framework of sequential decision-making is quite general and can be used to pose a large number of tasks from diverse fields (e.g., Bertsekas, 1987).

Much of everyday human activity involves sequential tasks that have compositional structure, i.e., complex tasks built up in a systematic way from simpler tasks. As an example, consider how many of our daily activities involve the subtasks of lifting an object, opening a door, sitting down, walking, etc. Compositionally-structured tasks allow the possibility of sharing knowledge across the many tasks that have common subtasks. To formulate the problem abstractly, consider an agent that has to learn to solve many different elemental and composite tasks. In general, there may be n elemental tasks labeled $T_1, T_2, \ldots, T_n$. *Elemental* tasks cannot be decomposed into simpler subtasks. *Composite* tasks, labeled $C_1, C_2, \ldots, C_m$, are produced by temporally concatenating a number of elemental tasks. For example, $C_j = [T(j, 1)T(j, 2) \ldots T(j, k)]$, is composite task j made up of k elemental tasks that have to be performed in the order listed. $T(j, i) \in \{T_1, T_2, \ldots, T_n\}$, for $1 \leq i \leq k$, is the ith elemental task in the list for task C_j. The sequence of elemental tasks in a composite task will be referred to as the *decomposition* of the composite task. I assume that the decomposition of a composite task is unknown to the leaning agent.

Compositional learning involves solving a composite task by learning to compose the solutions of the elemental tasks in its decomposition. It is to be emphasized that given the short-term, evaluative nature of the payoff from the environment (often the agent gets informative payoff only at the completion of the composite task), the task of discovering the decomposition of a composite task is formidable. In this paper I propose a compositional learning scheme in which the separate modules learn to solve the elemental tasks, and a task-sensitive gating module solves composite tasks by learning to compose the appropriate elemental modules over time (also see Singh, 1992a). I present the results of three simulations using a set of simple navigational tasks that illustrate the advantage of compositional learning, and I prove one aspect of the learning algorithm in Appendix A.

2. Markovian decision tasks and Q-learning

One often-studied class of sequential decision tasks consists of Markovian decision tasks (MDTs) in which the agent seeks to control a finite-state, discrete-time, stochastic dynamical system. Let S be the finite set of states and A be the finite set of actions available to the agent.[3] At each time step t, the agent observes the system's current state $x_t \in S$ and executes action $a_t \in A$. As a result, the agent receives a payoff with expected value $R(x_t, a_t) \in \mathbb{R}$ and the system makes a transition to state $x_{t+1} \in S$ with probability $P_{x_t x_{t+1}}(a_t)$. The environment satisfies the Markovian property, i.e., the current state and future actions determine the expected future sequence of states and payoffs independently of the state trajectory

prior to the current state. The transition probabilities for all states and actions constitute the environment dynamics. The payoff function that assigns expected payoffs to state-action pairs is defined independently of the environment dynamics.

The agent's *goal* is to determine a closed loop control policy that maximizes some cumulative measure of the payoffs received over time. The number of time steps over which the cumulative payoff is determined is called the *horizon* of the MDT. One measure, called the expected *return* of a policy for a given initial environment state x_0, is the expected value of the sum over an infinite horizon of the discounted payoffs that the agent would receive if it were to use the given policy to select its actions. The expected return is equal to $\mathcal{E}[\Sigma_{t=0}^{\infty} \gamma^t R(x_t, a_t)]$, where $\mathcal{E}$ indicates expected value. The discount factor γ, where $0 \leq \gamma \leq 1$, allows the payoffs distant in time to be weighted less than more immediate payoffs. Any control policy that achieves the agent's goal (and there may be more than one) is called an optimal policy. The optimal policies for the class of infinite horizon MDTs defined above are stationary (Ross, 1983), i.e., are not functions of time, and therefore, I restrict attention to stationary policies. Clearly, both the environment dynamics and the payoff function play a role in determining the set of optimal policies. Changing the payoff function while keeping the same environment dynamics can change the set of policies that are optimal. That this is not always true is apparent from the fact that there are a finite number of stationary policies but an infinite number of payoff functions.

If the environment dynamics and the payoff function are known, computational procedures based on dynamic programming can be used to find an optimal policy (Bertsekas, 1987). In particular, DP methods based on the value iteration algorithm (e.g., Ross, 1983) compute an optimal *value function* that assigns to states their scalar expected returns under an optimal policy. Given the optimal value function, an optimal policy can be determined by selecting for each state an action that if executed would maximize the expected value of the sum of the immediate payoff and the expected return for the next state. In the absence of a model of the environment dynamics, reinforcement learning methods that approximate dynamic programming, such as temporal difference (Sutton, 1988; Barto et al., 1983) and Q-learning methods (Watkins, 1989; Barto et al., 1990) can be used to directly estimate an optimal policy without building an explicit model of the dynamics of the environment.

Following Watkins, I define the Q-value, $Q(x, a)$, for $x \in S$ and $a \in A$, as the expected return on taking action a in state x, under the condition that an optimal policy is followed thereafter. Given the Q-values, a greedy policy that in each state selects an action with the highest associated Q-value, is optimal. Q-learning is an asynchronous version of value iteration that maintains an estimate, $\hat{Q}$, of the Q-values for all states and actions. Q-learning is often used on-line, i.e., actual experience interacting with the environment is used to incrementally improve $\hat{Q}$ over time. On executing action a in state x at time t, the resulting payoff and next state are used to update the estimate at time t, $\hat{Q}_t(x, a)$ as follows:[4]

$$\hat{Q}_{t+1}(x, a) = (1.0 - \alpha_t)\hat{Q}_t(x, a) + \alpha_t(R(x, a) + \gamma \max_{a' \in A} \hat{Q}_t(y, a')), \tag{1}$$

where y is the state at time $t + 1$, and α_t is the value of a positive learning rate parameter at time t. Watkins and Dayan (1992) prove that under certain conditions on the sequence

$\{\alpha_t\}$, if every state-action pair is updated infinitely often using Equation 1, $\hat{Q}$ converges to the true Q-values asymptotically.

Computing a greedy policy with respect to $\hat{Q}$ requires evaluating $\hat{Q}(x, a)$ for all $x \in S$ and $a \in A$. A common practice that eliminates this computation is to cache the control policy in a separate controller module.[5] To ensure asymptotic convergence most control methods based on Q-learning incorporate some exploration strategy (Kaelbling, 1990; Barto & Singh, 1990; Sutton, 1990) that sometimes takes non-greedy actions.

3. Elemental and composite Markovian decision tasks

Several different elemental tasks can be defined in the same environment. In this paper I restrict attention to elemental tasks requiring the agent to bring the environment to a desired final state. Each elemental task has its own final state. All elemental tasks are MDTs that share the same state set S, action set A, and the same environment dynamics. The payoff function, however, can be different across the elemental tasks. Furthermore, I assume that the payoff function for each elemental task T_i, $1 \leq i \leq n$, $R_i(x, a) = \Sigma_{y \in S} P_{xy}(a) r_i(y) - c(x, a)$, where $r_i(y)$ is the positive reward associated with the state y resulting from executing action a in state x for task T_i, and $c(x, a)$ is the positive cost of executing action a in state x. I assume that $r_i(x) = 0$ if x is not the desired final state for T_i. Note that the reward function r_i is task-dependent while the cost function c is task-independent. Thus, the elemental tasks share the same cost function but have their own reward functions.

A composite task is defined as an ordered sequence of elemental MDTs. It is clear that a composite task is not itself an MDT because the payoff is a function of both the state and the current elemental task, instead of the state alone. To make composite tasks qualify as MDTs, the set S can be extended as follows:[6] imagine a device that for each elemental task detects when the desired final state of that elemental task is visited for the first time and then remembers this fact. This device can be considered part of the learning system or equivalently as part of a *new* environment for the composite task.

Formally, the new state set for a composite task, S', is formed by augmenting the elements of set S by n bits, one for each elemental task. For each $x' \in S'$ the *projected state* $x \in S$ is defined as the state obtained by removing the augmenting bits from x'. The environment dynamics and cost function, c, for a composite task is defined by assigning to each $x' \in S'$ and $a \in A$ the transition probabilities and cost assigned to the projected state $x \in S$ and $a \in A$. The reward function for composite task C_j, r_j^c, is defined as follows: $r_j^c(x') > 0$ if the projected state x is the final state of some elemental task in the decomposition of C_j, say task T_i, and if the augmenting bits of x' corresponding to elemental tasks before and including subtask T_i in the decomposition of C_j are one, and if the rest of the augmenting bits are zero; $r_j^c(x') = 0$ everywhere else.

Composite tasks as defined above are MDTs, and the results of Watkins and Dayan (1992) extend to them. Thus, Q-learning can be used to learn the Q-values for a composite task. The learning agent has to learn to solve a number of elemental and composite tasks in its environment. At any given time, the task the agent faces is determined by a device that can be considered to be part of the environment or to be a part of the agent. If the device is considered to be part of the environment, it can be thought of as providing a task

command to the agent. On the other hand, if the device is part of the agent, it provides a context or internal state for the agent. The two views are formally equivalent; the crucial property is that they determine the payoff function but do not affect the dynamics of the environment.

If the task command is considered part of the state description, the entire set of tasks faced by the agent becomes one unstructured Markovian decision task. While an optimal policy for the unstructured MDT can be found by using Q-learning, just as for an elemental MDT, the structure inherent in the set of compositionally structured tasks allows a more efficient solution. In the next section I show that the Q-values for a composite task have a special relationship to the Q-values of the elemental tasks in its decomposition. I then present a learning architecture that uses that relationship to advantage.

4. Compositional Q-learning

Compositional Q-learning (CQ-learning) is a method for constructing the Q-values of a composite task from the Q-values of the elemental tasks in its decomposition. Let $Q^{T_i}(x, a)$ be the Q-value of (x, a), $x \in S$ and $a \in A$, for elemental task T_i, and let $Q^{C_j}_{T_i}(x', a)$ be the Q-value of (x', a), for $x' \in S'$ and $a \in A$, for task T_i when performed as part of the composite task $C_j = [T(j, 1) \ldots T(j, k)]$. Let $T(j, l) = T_i$. Note that the superscript on Q refers to the task and the subscript refers to the elemental task currently being performed. The absence of a subscript implies that the task is elemental.

Consider a set of undiscounted ($\gamma = 1$) MDTs that have compositional structure and satisfy the following conditions:

(A1) Each elemental task has a single desired final state.

(A2) For all elemental and composite tasks the expected value of the undiscounted return for an optimal policy is bounded both from above and below for all states.

(A3) The cost associated with each state-action pair is independent of the task being accomplished.

(A4) For each elemental task T_i the reward function r_I is zero for all states except the desired final state for that task. For each composite task C_j, the reward function r^c_j is zero for all states except the final states of the elemental tasks in its decomposition (see Section 3).

Then, for any elemental task T_i and for all composite tasks C_j containing elemental task T_i, the following holds:

$$Q^{C_j}_{T_i}(x', a) = Q^{T_i}(x, a) + K(C_j, l), \tag{2}$$

for all $x' \in S'$ and $a \in A$, where $x \in S$ is the projected state, and $K(C_j, l)$ is a function of the composite task C_j and l, where T_i is the lth task in the decomposition of C_j. Note that $K(C_j, l)$ is independent of the state and the action. Thus, given solutions of the elemental tasks, learning the solution of a composite task with n elemental tasks requires learning only the values of the function K for the n different subtasks. A proof of Equation 2 is given in Appendix A.

Equation 2 is based on the assumption that the decomposition of the composite tasks is known. In the next Section, I present a modular architecture and learning algorithm that simultaneously discovers the decomposition of a composite task and implements Equation 2.

5. CQ-L: The CQ-learning architecture

Jacobs et al. (1991) developed a modular gating architecture that does task decomposition. The gating architecture consists of several expert modules and a special gating module that has an output for each expert module. When presented with training patterns (input-output pairs) from multiple tasks, the expert modules compete with each other to learn the training patterns, and this competition is mediated by the gating module. The best interpretation of this architecture is as an "associative Gaussian mixture model" (see Jacobs & Jordan, 1991; Jacobs, et al., 1991) of the distribution of training patterns. Learning is achieved by gradient descent in the log likelihood of generating the desired target patterns.

The gating architecture has been used to learn only multiple non-sequential tasks within the supervised learning paradigm (Duda & Hart, 1973). I extend the gating architecture to a CQ-Learning architecture (Figure 1), called CQ-L, that can learn multiple compositionally-structured sequential tasks even when training information required for supervised learning is not available. CQ-L combines CQ-learning and the gating architecture to achieve transfer of learning by "sharing" the solutions of elemental tasks across multiple composite tasks. Only a high level description of CQ-L is provided in this section; the details are given in Appendix B.

In CQ-L the expert modules of the gating architecture are replaced by Q-learning modules that learn to approximate the Q-values for the elemental tasks. The Q-modules receive as input both the state ($\in S$) and the actions ($\in A$). The gating and bias modules (see Figure 1) receive as input the augmenting bits and the task command used to encode the current task being performed by the architecture. The stochastic switch in Figure 1 selects one Q-module at each time step. CQ-L's output is the output of the selected Q-module added to the output of the bias module.

At time step t, let the output of the Q-module i be $q_i(t)$, and let the ith output of the gating module after normalization be $g_i(t)$. $g_i(t)$ is the *prior* probability that the Q-module i is selected at time step t by the stochastic switch. At each time step, the following steps are performed. For the current task and current state, x_t, the outputs of the gating module are determined and used to select a single Q-module. I use q_t to refer to the output of the Q-module selected at time t. $q_t(x_t, a)$ is evaluated for all $a \in A$. One action is selected probabilistically using the Gibbs distribution, i.e., $\forall a \in A$, $P(a \mid x_t) = e^{\beta q_t(x_t,a)}/\Sigma_{a' \in A} e^{\beta q_t(x_t,a')}$, where the parameter β controls the probability of selecting a non-greedy action. The chosen action, a_t, is then executed, and $q_i(x_t, a_t)$ is evaluated for all i.

The estimated Q-value for the current task and state action pair, $\hat{Q}(x_t, a_t)$ is set equal to $q_t(x_t, a_t) + K(t)$, where $K(t)$ is the output of the bias module at time t. The bias module implements the function K of Equation 2. The estimate of the error in $\hat{Q}(x_t, a_t)$, which becomes available at time $t + 1$, is

$$e(t) = R(x_t, a_t) + \hat{Q}(x_{t+1}, a_{t+1}) - \hat{Q}(x_t, a_t), \tag{3}$$

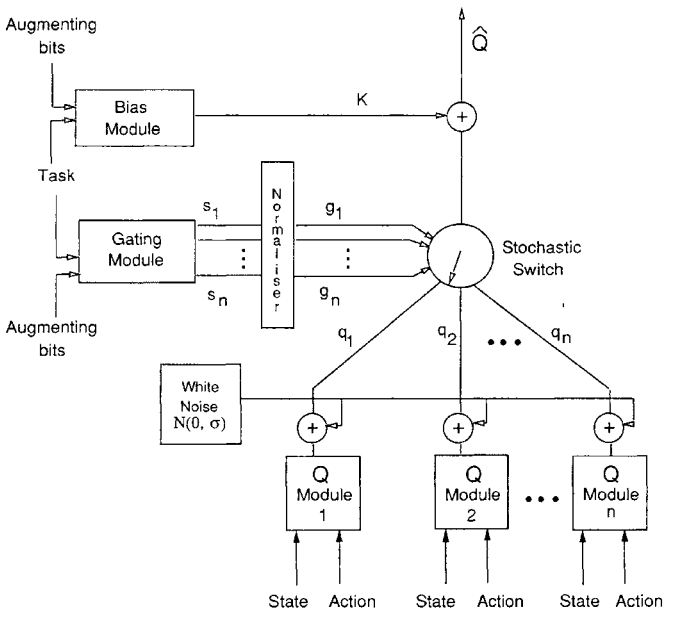


Figure 1. CQ-L: The CQ-Learning Architecture. This figure is adapted from Jacobs, et al. (1991). The Q-modules learn the Q-values for elemental tasks. The gating module has an output for each Q-module and determines the probability of selecting a particular Q-module. The bias module learns the function K (see Equation 2).

where $R(x_t, a_t)$ was the expected payoff at time t.[7] Alternatively, the estimate of the desired output of CQ-L at time t can be defined as $D(t) = e(t) + \hat{Q}(t)$. At each time step the learning rules described in Appendix B are used to adjust *all* the Q-modules, the gating module, and the bias module.

Each Q-module is adjusted to reduce the error between its output and $D(t) - K(t)$ in proportion to the probability of that Q-module having produced the estimated desired output. Hence the Q-module whose output would have produced the least error is adjusted the most. The gating module is adjusted so that the *a priori* probability of selecting each Q-module becomes equal to the *a posteriori* probability of selecting that Q-module, given $D(t)$. Over time, different Q-modules start winning the competition for different elemental tasks, and the gating module learns to select the appropriate Q-module for each task. For composite tasks, while performing subtask T_i, the Q-module that has best learned task T_i will have smaller expected error than any other Q-module. The bias module is also adjusted to reduce the error $e(t)$. The parameter β is increased over time so that eventually only the greedy actions are selected.

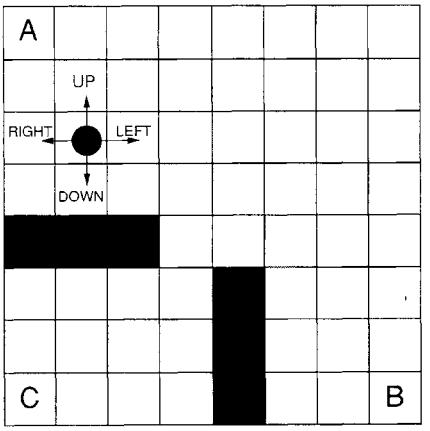


Figure 2. The Grid Room. The room is an 8 × 8 grid with three desired final locations designated *A*, *B* and *C*. The filled squares represent obstacles. The robot is shown as a circle and has four actions available: UP, DOWN, RIGHT, and LEFT.

6. Navigation tasks: Description and Representation

I use a set of deterministic navigation tasks to illustrate the performance of CQ-L. Figure 2 shows an 8 × 8 grid room with three final locations designated *A*, *B* and *C*. The robot is shown as a circle, and the filled squares represent obstacles that the robot must avoid. In each state the robot has 4 actions: UP, DOWN, LEFT and RIGHT. Any action that would take the robot into an obstacle or boundary wall does not change the robot's location. There are three elemental tasks: "visit *A*," "visit B," and "visit *C*," labeled T_1, T_2 and T_3, respectively. Three composite tasks, C_1, C_2 and C_3, were constructed by temporally concatenating some subset of the elemental tasks. It is to be emphasized that the goal for each task is not merely to find any path that leads to the desired final state, but to find the shortest path. The six different tasks, along with their labels, are described in Table 1. $c(x, a) = -0.05$, for all $x \in S \cup S'$ and $a \in A$. $r_i(x) = 1.0$, if $x \in S$ is the desired final state of elemental task T_i, or if $x \in S'$ is the final state of composite task C_i, and $r_i(x) = 0.0$ in all other states. Thus, for composite tasks no intermediate payoff for successful completion of subtasks was provided.

The representation used for the task command determines the difficulty of discovering the decomposition of a composite task. At one extreme, using "linguistic" representations can reduce the problem of discovering the decomposition to that of "parsing" the task command. At the other extreme, unstructured task command representations force the system to learn the decomposition of each composite task separately. To avoid the issues arising from structured representations I use unit basis vectors to represent the task commands

Table 1. Tasks. Tasks T_1, T_2, and T_3 are elemental tasks; tasks C_1, C_2, and C_3 are composite tasks. The last column describes the compositional structure of the tasks.

Label	Command	Description	Decomposition
T_1	000001	visit A	T_1
T_2	000010	visit B	T_2
T_3	000100	visit C	T_3
C_1	001000	visit A and then C	T_1T_3
C_2	010000	visit B and then C	T_2T_3
C_3	100000	visit A, then B and then C	$T_1T_2T_3$

(Table 1). Lookup tables were used to implement all the modules of the CQ-Learning architecture. The elements of the set S are the possible positions of the robot in the grid room. The 3 augmenting bits, one for each the three elemental tasks, and the task command vector form the input to the bias and the gating modules (Figure 1).

7. Simulation results

In the simulations described below, the performance of CQ-L is compared to the performance of a "one-for-one" architecture that implements the "learn-each-task-separately" strategy. The one-for-one architecture has a pre-assigned distinct module for each task, which prevents transfer of learning, in other words, the one-for-one architecture treats the entire set of tasks as one unstructured MDT. Each module of the one-for-one architecture was provided with the augmented state ($\in S'$).

7.1. Simulation 1: Learning multiple elemental Markovian decision tasks

Both CQ-L and the one-for-one architecture were separately trained on the three elemental MDTs T_1, T_2, and T_3 until they could perform the three tasks optimally. Both architectures contained three Q-modules. The task for each trial and the starting location of the robot were chosen randomly. Each trial ended when the robot reached the desired final location for that task. Figure 3(i) shows the number of actions, with each data point an average over 50 trials, taken by the robot to get to the desired final state. The one-for-one architecture converged to an optimal policy faster than CQ-L did, because it took time for the gating module's outputs to become approximately correct, at which point CQ-L learned rapidly.

Figures 3(ii), 3(iii), and 3(iv) respectively show the three normalized outputs of the gating module for trials involving tasks T_1, T_2 and T_3. Each data point is the output of the gating module averaged over a complete trial. For each graph, the abscissa is the number of times the corresponding tasks occurred. The gating module outputs started close to 0.3. After approximately 100 trials the gating module had learned to assign a different Q-module to each task. This simulation shows that CQ-L is able to partition its "across-trial" experience and learn to engage one Q-module for each elemental task. This is similar in spirit to the

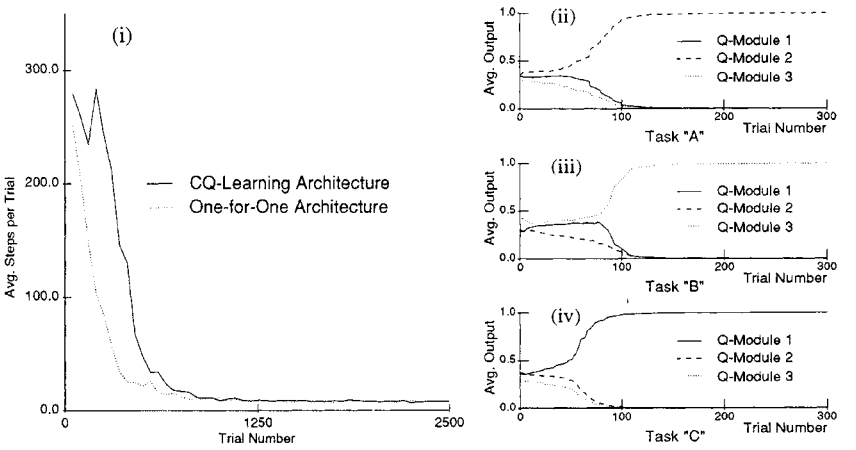


Figure 3. (i) Learning Curves for Multiple Elemental tasks. Each data point is the average, taken over 50 trials, of the number of actions taken by the robot to get to the desired final state. (ii) Module Section for Task T_1. The 3 normalized outputs of the gating module are shown averaged over each trial with task T_1. Initially the outputs were about 0.3 each, but as learning proceeded the gating module learned to select Q-module 2 for task T_1. (iii) Module Selection for Task T_2. Q-module 3 was selected. (iv) Module Selection for Task T_3. Q-module 1 was selected for task T_3.

simulations reported by Jacobs (1990), except that he does not apply his architecture to sequential decision tasks. See Appendix C for simulation details.

7.2. Simulation 2: Compositional learning

Both CQ-L and the one-for-one architecture were separately trained on the six tasks T_1, T_2, T_3, C_1, C_2, and C_3 until they could perform the six tasks optimally. CQ-L contained three Q-modules, and the one-for-one architecture contained six Q-modules. The task for each trial and the starting state of the robot were chosen randomly. Each trial ended when the robot reached the desired final state. Figure 4(i) shows the number of actions, averaged over 50 trials, taken by the robot to reach the desired final state. The one-for-one architecture performed better initially because it learned the three elemental tasks quickly, but learning the composite tasks took much longer due to the long action sequences required to accomplish the composite tasks. The CQ-Learning architecture performed worse initially, until the outputs of the gating module became approximately correct, at which point all six tasks were learned rapidly.

Figures 4(ii), 4(iii), and 4(iv) respectively show the three outputs of the gating module for one trial each of tasks C_1, C_2, and C_3. The trials shown were chosen after the robot had learned to do the tasks, specifically, after 10,000 learning trials. The elemental tasks T_1, T_2 and T_3 respectively were learned by the Q-modules 1, 3, and 2. For each composite task the gating module learned to compose the outputs of the appropriate elemental Q-modules

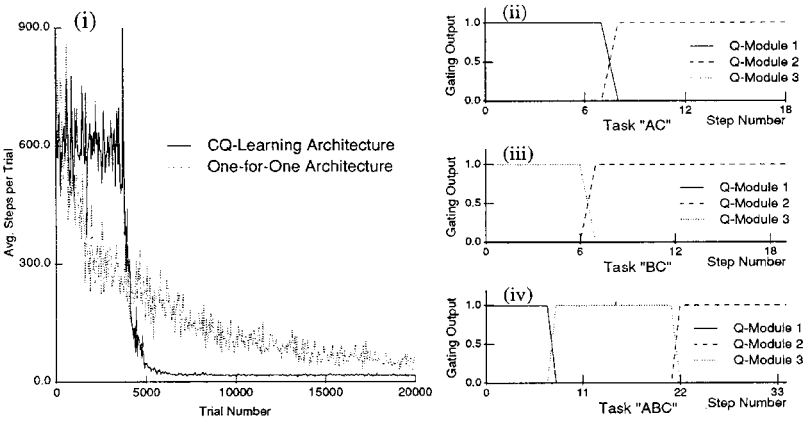


Figure 4. (i) Learning Curves for a Set of Elemental and Composite Tasks. Each data point is the average over 50 trials of the time taken by the robot to reach the desired final state. (ii) Temporal Composition for Task C_1. After 10,000 learning trials, the three outputs of the gating module during *one* trial of task C_1 are shown. Q-module 1 was turned on for the first seven actions to accomplish subtask T_1, and then Q-module 2 was turned on to accomplish subtask T_3. (iii) Temporal Composition for Task C_2. Q-module 3 was turned on for the first six actions to accomplish subtask T_2 and then Q-module 2 was turned on to accomplish task T_3. (iv) Temporal Composition for Task C_3. The three outputs of the gating module for one trial with task C_3 are shown. Q-modules 1, 3 and 2 were selected in that order to accomplish the composite task C_3.

over time. This simulation shows that CQ-L is able to learn the decomposition of a composite task, and that compositional learning, due to transfer of training across tasks, can be faster than learning each composite task separately. See Appendix C for simulation details.

7.3. Simulation 3: Shaping

One approach (cf. Section 7.2) for training a robot to learn a composite task is to train the robot on multiple tasks: all the elemental tasks in the decomposition of the composite task, and the composite task itself. Another approach is to train the robot on a succession of tasks, where each succeeding task requires some subset of the already learned elemental tasks, plus a new elemental task. This roughly corresponds to the "shaping" procedures (Skinner, 1938) used by psychologists to train animals to do complex motor tasks.

A simple simulation to illustrate shaping was constructed by training CQ-L with one Q-module on one elemental task, T_3, for 1,000 trials and then training on the composite task C_2. After the first 1,000 trials, the learning was turned off for the first Q-module and a second Q-module was added for the composite task. Figure 5 shows the learning curve for task T_3 composed with the learning curve for task C_2. The number of actions taken by the robot to get to the desired final state, averaged over 50 trials, were plotted by concatenating the data points for the two tasks, T_3 and C_2. Figure 5 shows that the average

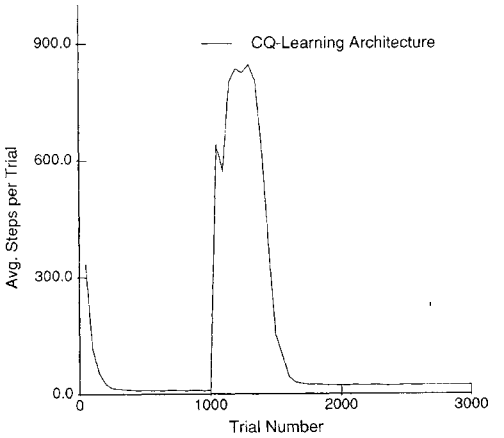


Figure 5. Shaping. The number of actions taken by the robot to reach the desired final state, averaged over 50 trials, is plotted. The CQ-Learning architecture containing one Q-module was trained for 1000 trials on task T_3. Then another Q-module was added, and the architecture was trained to accomplish task C_2. The only task-dependent payoff for the task C_2 was on reaching the desired final location C.

number of actions required to reach the final state increases when the composite task was introduced, but eventually the gating module learned to decompose the task and the average decreased. The second Q-module learned the task T_2 without ever being explicitly exposed to it.

A separate shaping simulation was conducted in the same way as above, except that learning was not turned off for the first Q-module after it had learned task T_3. The solution to task T_3 was partially unlearned before the architecture figured out the decomposition for the composite task. As a result, it took more time than in the earlier shaping simulation to learn the elemental task T_2 in the second Q-module. See Appendix C for simulation details.

8. Discussion

Learning to solve MDTs with large state sets is difficult due to the sparseness of the evaluative information and the low probability that a randomly selected sequence of actions will be optimal. Learning the long sequences of actions required to solve such tasks can be accelerated considerably if the agent has prior knowledge of useful subsequences. Such subsequences can be learned through experience in learning to solve other tasks. In this paper, I define a class of MDTs, called composite tasks, that are structured as the temporal concatenation of simpler MDTs, called elemental tasks. I present CQ-L, an architecture that combines the Q-learning algorithm of Watkins (1989) and the modular architecture of Jacobs, et al. (1991) to achieve transfer of learning by sharing the solutions of elemental tasks across

multiple composite tasks. CQ-L's use of acquired subsequences to solve composite tasks is similar to the use of macro-operators (Korf, 1985) in macro-learning systems (Iba, 1989).

Another architecture similar to CQ-L is the subsumption architecture for autonomous intelligent agents (Brooks, 1989), which is composed of several task-achieving modules along with precompiled switching circuitry that controls which module should be active at any time. Maes and Brooks (1990) showed how reinforcement learning can be used to learn the switching circuitry for a robot with hardwired task modules. Mahadevan and Connell (1990), on the other hand, showed how Q-learning can be used to acquire behaviors that can then be controlled using a hardwired switching scheme. The simulation illustrating shaping with CQ-L demonstrates that not only can the gating module compose sequences of elemental tasks on which it has been trained, it can also learn new elemental tasks in order to solve a composite task. Thus, for compositionally-structured MDTs, CQ-L combines the complementary objectives of Maes and Brooks's architecture with that of Mahadevan and Connell's architecture.

Given a set of composite and elemental MDTs, the sequence in which the learning agent receives training experiences on the different tasks determines the relative advantage of CQ-L over other architectures that learn the tasks separately. The simulation reported in Section 7.2 demonstrates that it is possible to train CQ-L on intermixed trials of elemental and composite tasks, while the simulation involving shaping (Section 7.3) demonstrates that it is possible for CQ-L to learn elemental tasks on which it has not been explicitly trained. Nevertheless, some training sequences on a set of tasks will result in faster learning of the set of tasks than other training sequences. The ability of CQ-L to scale well to complex sets of tasks is still dependent on the choice of the training sequence.

To focus on compositional learning I avoided the issues arising from the use of representations more structured than unit-basis vectors and function approximators more powerful than lookup tables. In addition, Watkins and Dayan's (1992) theorem on the convergence of Q-learning holds for lookup-table representations; it is not known to hold for other function approximation methods. Nevertheless, future work with connectionist networks may make it possible to use distributed representations for states and thus to take advantage of the ability of networks to generalize across states within the same task.

According to the definition used in this paper, composite tasks have only one decomposition and require the elemental tasks in their decomposition to be performed in a fixed order. A broader definition of a composite task allows it to be an unordered list of elemental tasks, or more generally, a disjunction of many ordered elemental task sequences. CQ-L should work with the broader definition for composite tasks without any modification because it should select the particular decomposition that is optimal with respect to its goal of maximizing expected returns. Further work is required to test this conjecture.

Although compositional Q-learning was illustrated using a set of simple navigational tasks, it is suitable for a number of different domains where multiple sequences from some set of elemental tasks have to be learned. CQ-L is a general mechanism whereby a "vocabulary" of elemental tasks can be learned in separate Q-modules, and arbitrary[8] temporal syntactic compositions of elemental tasks can be learned with the help of the bias and gating modules.

Appendix A: Proof of equation 2

Consider an elemental deterministic Markovian decision task (MDT) T_i and its desired final state $x_g \in S$. The payoff function for task T_i is $R_i(x, a) = \Sigma_{y \in S} P_{xy}(a) r_i(y) - c(x, a)$, for all $x \in S$ and $a \in A$. By assumptions A1–A4 (Section 4) we know that the reward $r_i(x) = 0$ for all $x \neq x_g$. Thus, for any state $y \in S$ and action $a \in A$,

$$Q^{T_i}(y, a) = r_i(x_g) - c(y, a) - \Phi(y, x_g),$$

where $\Phi(y, x_g)$ is the expected cost of going from state y to x_g under an optimal policy, π_i, for elemental task T_i.

Consider a composite task C_j that satisfies assumptions A1–A4 given in Section 4 and w.l.o.g. assume that for $C_j = [T(j, 1)T(j, 2) \ldots T(j, k)]$, $\exists 1 \leq l \leq k$, such that $T(j, l) = T_i$. Let the set $L \subset S'$ be the set of all $x' \in S'$ that satisfy the property that the augmenting bits corresponding to the tasks before T_i in the decomposition of C_j are equal to one and the rest are equal to zero. Let $y' \in L$ be the state that has the projected state $y \in S$. Let $x_g' \in S'$ be the state formed from $x_g \in S$ by setting to one the augmenting bits corresponding to all the subtasks before and including subtask T_i in the decomposition of C_j, and setting the other augmenting bits to zero. Let π_j' be an optimal policy for task C_j. $r_j^c(x')$ is the reward for state $x' \in S'$ while performing task C_j. Then by assumptions A1–A4, we know that $r_j^c(x') = 0$ for all $x' \in L$ and that $c(x', a) = c(x, a)$.

By the definition of C_j, the agent has to navigate from state y' to state x_g' to accomplish subtask T_i. Let $\Phi'(y', x_g')$ be the expected cost of going from state y' to state x_g' under policy π_j'. Then, given that $T(j, l) = T_i$,

$$Q_{T_i}^{C_j}(y', a) = Q_{T(j,l+1)}^{C_j}(x_g', b) + r_i^c(x_g') - c(y', a) - \Phi'(y', x_g'),$$

where $b \in A$ is an optimal action for state x_g' while performing subtask $T(j, l + 1)$ in task C_j. Clearly, $\Phi'(y', x_g') = \Phi(y, x_g)$, for if it were not, either policy π_i would not be optimal for task T_i, or given the independence of the solutions of the subtasks the policy π_j' would not be optimal for task C_j. Define $K(C_j, l) \equiv Q_{T(j,l+1)}^{C_j}(x_g', b) + r_i^c(x_g') - r_i(x_g)$. Then

$$Q_{T_i}^{C_j}(y', a) = Q^{T_i}(y, a) + K(C_j, l).$$

Q.E.D.

Appendix B: Learning rules for the CQ-learning architecture

The derivation of the learning rules in this section follows the derivation in Nowlan (1990) closely. The output of Q-module i, q_i, is the mean of a Gaussian probability distribution with variance σ. The ith output of the gating module, s_i determines the *a priori* probability, $g_i = e^{s_i}/\Sigma_j e^{s_j}$, of selecting Q-module i.

The final output of the architecture at time step t, is given by $\hat{Q}(t) = q_t(t) + K(t)$, where q_t is the output of the selected Q-module and K is the output of the bias module. The desired output at time t is, $D(t) = R(t) + \hat{Q}(t + 1)$, where $R(t)$ is the expected

payoff at time t. Then the probability that the Q-module i will generate the desired output is

$$p_i(D(t)) = \frac{1}{N\sigma} e^{\frac{\|(D(t)-K(t))-q_i(t)\|^2}{2\sigma^2}},$$

where N is a normalizing constant. The *a posteriori* probability that Q-module i was selected given the desired output $D(t)$, is

$$p(i|D(t)) = \frac{g_i(t)p_i(D(t))}{\Sigma_j g_j(t)p_j(D(t))}.$$

The likelihood of producing the desired output, $L(D(t))$, is given by $\Sigma_j g_j(t)p_j(D(t))$.

The objective of the architecture is to maximize the log likelihood of generating the desired Q-values for the task it is being trained on. The partial derivative of the log likelihood with respect to the output of the Q-module j is

$$\frac{\partial \log L(t)}{\partial q_j(t)} = \frac{1}{\sigma^2} p(j|D(t))((D(t) - K(t)) - q_j(t)).$$

The partial derivative of the log likelihood with respect to the ith output of the gating module simplifies to

$$\frac{\partial \log L(t)}{\partial s_i(t)} = (p(i|D(t)) - g_i(t)).$$

Using the above results the update rules for Q-module j, the ith output of the gating module, and the bias module[9] respectively are:

$$q_j(t + 1) = q_j(t) + \alpha_Q \frac{\partial \log L(t)}{\partial q_j(t)},$$

$$s_i(t + 1) = s_i(t) + \alpha_g \frac{\partial \log L(t)}{\partial s_i(t)}, \text{ and}$$

$$b(t + 1) = b(t) + \alpha_b(D(t) - Q(t)),$$

where α_Q, α_b and α_g are learning rate parameters.

Appendix C: Parameter values for simulations 1, 2 and 3

For all three simulations, the initial values for the lookup tables implementing the Q-modules were random values in the range 0.0–0.5, and the initial values for the gating module lookup

table were random values in the range 0.0–0.4. For all three simulations, the variance of the Gaussian noise, σ, was 1.0 for all Q-modules.

For Simulation 1, the parameter values for the both CQ-Learning and the one-for-one architectures were $\alpha_Q = 0.1$, $\alpha_b = 0.0$, $\alpha_g = 0.3$. The policy selection parameter, β, started at 1.0 and was incremented by 1.0 after every 100 trials.

For Simulation 2, the parameter values for CQ-L were: $\alpha_Q = 0.015$, $\alpha_b = 0.0001$, $\alpha_g = 0.025$, and β was incremented by 1.0 after every 200 trials. For the one-for-one architecture, the parameter values were: $\alpha_Q = 0.01$ and β was incremented by 1.0 after every 500 trials.[10]

For Simulation 3, the parameter values, $\alpha_Q = 0.1$, $\alpha_b = 0.0001$, and $\alpha_g = 0.01$ were used, and β was incremented by 1.0 every 25 trials.

Acknowledgments

This work was supported by the Air Force Office of Scientific Research, Bolling AFB, under Grant AFOSR-89-0526 and by the National Science Foundation under Grant ECS-8912623. I thank Andrew Barto, Rich Sutton, and Robbie Jacobs for their guidance and help in formulating my ideas. Discussions with Richard Yee, Vijaykumar Gullapalli, and Jonathan Bachrach helped focus my ideas. I also thank Andrew Barto for his patience in reading and commenting on many hastily written drafts of this paper. Without his extensive help, no reader would have gotten far enough in this paper to be reading this sentence.

Notes

1. The environment is the system or plant being controlled by the agent.
2. In the machine learning literature closed loop control policies are also referred to as situation-action rules, or simply as "reactions."
3. For ease of exposition I assume that the same set of actions are available to the agent from each state. The extension to the case where different sets of actions are available in different states is straightforward.
4. Here, I assume that on executing action a in state x the agent receives the expected value of the immediate payoff $R(x, a)$, instead of just a sample.
5. However, this adds the computational effort of training the controller module. In addition, the mismatch between the controller module and $\hat{Q}$ can adversely affect the time required to learn an optimal policy.
6. The theory developed in this paper does not depend on the particular extension of S chosen, as long as the appropriate connection between the new states and the elements of S can be made.
7. Note that Equation 3 is an accurate implementation of Q-learning only if the action taken at time $t + 1$ is the greedy action. As the value of parameter β is increased, the probability of executing the greedy action increases.
8. This assumes that the state representation is rich enough to distinguish repeated performances of the same elemental task.
9. This assumes that the bias module is minimizing a mean square error criteria.
10. Incrementing the β values faster did not help the one-for-one architecture.

References

Barto, A.G., Bradtke, S.J., & Singh, S.P. (1991). *Real-time learning and control using asynchronous dynamic programming*. (Technical Report 91-57). Amherst, MA: University of Massachusetts, COINS Dept.

Barto, A.G. & Singh, S.P. (1990). On the computational economics of reinforcement learning. *Proceedings of the 1990 Connectionist Models Summer School*. San Mateo, CA: Morgan Kaufmann.

Barto, A.G., Sutton, R.S., & Anderson, C.W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE SMC*, 13, 835–846.

Barto, A.G., Sutton, R.S., & Watkins, C.J.C.H. (1990). Sequential decision problems and neural networks. In D.S. Touretzky, (Ed.), *Advances in neural information processing systems* 2, San Mateo, CA: Morgan Kaufmann.

Bertsekas, D.P. (1987). *Dynamic programming: Deterministic and stochastic models*. Englewood Cliffs, NJ: Prentice-Hall.

Brooks, R. (1989). A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation*, 1, 253–262.

Duda, R.O. & Hart, P.E. (1973). *Pattern classification and scene analysis*. New York: Wiley.

Iba, G.A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3, 285–317.

Jacobs, R.A. (1990). *Task decomposition through competition in a modular connectionist architecture*. Ph.D. Thesis, COINS Dept., Univ. of Massachusetts, Amherst, Mass.

Jacobs, R.A. & Jordan, M.I. (1991). A competitive modular connectionist architecture. *Advances in neural information processing systems*, 3.

Jacobs, R.A., Jordan, M.I., Nowlan, S.J., & Hinton, G.E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3.

Kaelbling, L.P. (1990). *Learning in embedded systems*. Ph.D. Thesis, Stanford University, Department of Computer Science, Stanford CA. Technical Report TR-90-04.

Korf, R.E. (1985). Macro-operators: A weak method for learning. *Artificial Learning*, 26, 35–77.

Maes, P. & Brooks, R. (1990). Learning to coordinate behaviours. *Proceedings of the Eighth AAAI* (pp. 796–802). Morgan Kaufmann.

Mahadevan, S. & Connell, J. (1990). Automatic programming of behavior-based robots using reinforcement learning. (Technical Report) Yorktown Heights, NY: IBM Research Division, T.J. Watson Research Center.

Nowlan, S.J. (1990). Competing experts: An experimental investigation of associative mixture models. (Technical Report CRG-TR-90-5). Toronto, Canada: Univ. of Toronto, Department of Computer Science.

Ross, S. (1983). *Introduction to stochastic dynamic programming*. New York: Academic Press.

Singh, S.P. (1992a). On the efficient learning of multiple sequential tasks. In J. Moody, S.J. Hanson, & R.P. Lippman, (Eds.), *Advances in neural information processing systems* 4, San Mateo, CA: Morgan Kaufmann.

Singh, S.P. (1992b). Solving multiple sequential tasks using a hierarchy of variable temporal resolution models. Submitted to Machine Learning Conference, 1992.

Skinner, B.F. (1938). *The behavior of organisms: An experimental analysis*. New York: D. Appleton Century.

Sutton, R.S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

Sutton, R.S. (1990). Integrating architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Workshop on Machine Learning* (pp. 216–224). San Mateo, CA: Morgan Kaufmann

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. Ph.D. Thesis, Cambridge Univ., Cambridge, England.

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

Whitehead, S.D. & Ballard, D.H. (1990). Active perception and reinforcement learning. *Proceedings of the Seventh International Conference on Machine Learning*. Austin, TX.

Machine Learning, 8, 341–362 (1992)

The Convergence of TD(λ) for General λ

PETER DAYAN dayan@helmholtz.sdsc.edu
Centre for Cognitive Science & Department of Physics, University of Edinburgh, EH8 9LW, Scotland

Abstract. The method of temporal differences (TD) is one way of making consistent predictions about the future. This paper uses some analysis of Watkins (1989) to extend a convergence theorem due to Sutton (1988) from the case which only uses information from adjacent time steps to that involving information from arbitrary ones.

It also considers how this version of TD behaves in the face of linearly dependent representations for states—demonstrating that it still converges, but to a different answer from the least mean squares algorithm. Finally it adapts Watkins' theorem that Q-learning, his closely related prediction and action learning method, converges with probability one, to demonstrate this strong form of convergence for a slightly modified version of TD.

Keywords. Reinforcement learning, temporal differences, asynchronous dynamic programming

1. Introduction

Many systems operate in temporally extended circumstances, for which whole sequences of states rather than just individual ones are important. Such systems may frequently have to predict some future outcome, based on a potentially stochastic relationship between it and their current states. Furthermore, it is often important for them to be able to learn these predictions based on experience.

Consider a simple version of this problem in which the task is to predict the expected ultimate terminal values starting from each state of an absorbing Markov process, and for which some further random processes generate these terminating values at the absorbing states. One way to make these predictions is to learn the transition matrix of the chain and the expected values from each absorbing state, and then solve a simultaneous equation in one fell swoop. A simpler alternative is to learn the predictions directly, without first learning the transitions.

The methods of temporal differences (TD), first defined as such by Sutton (1984; 1988), fall into this simpler category. Given some parametric way of predicting the expected values of states, they alter the parameters to reduce the inconsistency between the estimate from one state and the estimates from the next state or states. This learning can happen incrementally, as the system observes its states and terminal values. Sutton (1988) proved some results about the convergence of a particular case of a TD method.

Many control problems can be formalized in terms of controlled, absorbing, Markov processes, for which each *policy*, i.e., mapping from states to actions, defines an absorbing Markov chain. The engineering method of dynamic programming (DP) (Bellman & Dreyfus, 1962) uses the predictions of the expected terminal values as a way of judging and hence improving policies, and TD methods can also be extended to accomplish this. As discussed extensively by Watkins (1989) and Barto, Sutton and Watkins (1990), TD is actually very closely related to DP in ways that significantly illuminate its workings.

This paper uses Watkins' insights to extend Sutton's theorem from a special case of TD, which considers inconsistencies only between adjacent states, to the general case in which arbitrary states are important, weighted exponentially less according to their temporal distances. It also considers what TD converges to if the representation adopted for states is linearly dependent, and proves that one version of TD prediction converges with probability one, by casting it in the form of Q-learning.

Some of the earliest work in temporal difference methods was due to Samuel (1959; 1967). His checkers (draughts) playing program tried to learn a consistent function for evaluating board positions, using the discrepancies between the predicted values at each state based on limited depth games-tree searches, and the subsequently predicted values after those numbers of moves had elapsed. Many other proposals along similar lines have been made: Sutton acknowledged the influence of Klopf (1972; 1982) and in Sutton (1988) discussed Holland's bucket brigade method for classifier systems (Holland, 1986), and a procedure by Witten (1977). Hampson (1983; 1990) presented empirical results for a quite similar navigation task to the one described by Barto, Sutton and Watkins (1990). Barto, Sutton and Anderson (1983) described an early TD system which learns how to balance an upended pole, a problem introduced in a further related paper by Michie and Chambers (1968). Watkins (1989) also gave further references.

The next section defines TD(λ), shows how to use Watkins' analysis of its relationship with DP to extend Sutton's theorem, and makes some comments about unhelpful state representations. Section 3 looks at Q-learning, and uses a version of Watkins' convergence theorem to demonstrate in a particular case the strongest guarantee known for the behavior of TD(0).

2. TD(λ)

Sutton (1988) developed the rationale behind TD methods for prediction, and proved that TD(0), a special case with a time horizon of only one step, converges in the mean for observations of an absorbing Markov chain. Although his theorem applies generally, he illustrated the case in point with an example of the simple random walk shown in figure 1. Here, the chain always starts in state D, and moves left or right with equal probabilities from each state until it reaches the left absorbing barrier A or the right absorbing barrier G. The problem facing TD is estimating the probability it absorbs at the right hand barrier rather than the left hand one, given any of the states as a current location.

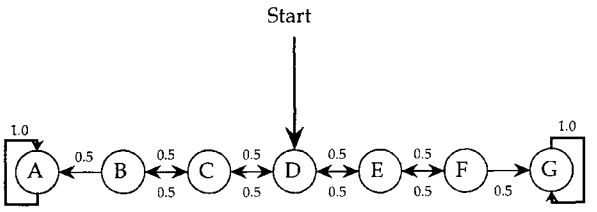


Figure 1. Sutton's Markov example. Transition probabilities given above (right to left) and below (left to right) the arrows.

The raw information available to the system is a collection of sequences of states and terminal locations generated from the random walk—it initially has no knowledge of the transition probabilities. Sutton described the supervised least mean squares (LMS) (Widrow & Stearns, 1985) technique, which works by making the estimates of the probabilities for each place visited on a sequence closer to 1 if that sequence ended up at the right hand barrier, and closer to 0 if it ended up at the left hand one. He showed that this technique is exactly TD(1), one special case of TD, and constrasted it with TD(λ) and particularly TD(0), which tries to make the estimate of probability from one state closer to the estimate from the next, without waiting to see where the sequence might terminate. The discounting parameter λ in TD(λ) determines exponentially the weights of future states based on their temporal distance—smoothly interpolating between $\lambda = 0$, for which only the next state is relevant, and $\lambda = 1$, the LMS case, for which all states are equally weighted. As described in the introduction, it is its obeisance to the temporal order of the sequence that marks out TD.

The following subsections describe Sutton's results for TD(0) and separate out the algorithm from the vector representation of states. They then show how Watkins' analysis provides the wherewithal to extend it to TD(λ) and finally re-incorporate the original representation.

2.1. The convergence theorem

Following Sutton (1988), consider the case of an absorbing Markov chain, defined by sets and values:

$\mathfrak{I}$		terminal states
$\mathfrak{N}$		non-terminal states
$q_{ij} \in [0, 1]$	$i \in \mathfrak{N}, j \in \mathfrak{N} \cup \mathfrak{I}$	transition probabilities
$\mathbf{x}_i \in \Re^c$	$i \in \mathfrak{N}$	vectors representing non-terminal states
$\bar{z}_j$	$j \in \mathfrak{I}$	expected terminal values from state j
μ_i	$i \in \mathfrak{N}$	probabilities of starting at state i, where $\sum_{i \in \mathfrak{N}} \mu_i = 1$.

The payoff structure of the chain shown in figure 1 is degenerate, in the sense that the values of the terminal states A and G are deterministically 0 and 1 respectively. This makes the expected value from any state just the probability of absorbing at G.

The estimation system is fed complete sequences $\mathbf{x}_{i_1}, \mathbf{x}_{i_2}, \ldots \mathbf{x}_{i_m}$ of observation vectors, together with their scalar terminal value z. It has to generate for every non-terminal state $i \in \mathfrak{N}$ a prediction of the expected value $\mathbb{E}[z \mid i]$ for starting from that state. If the transition matrix of the Markov chain were completely known, these predictions could be computed as:

$$\mathbb{E}[z \mid i] = \sum_{j \in \mathfrak{I}} q_{ij} \bar{z}_j + \sum_{j \in \mathfrak{N}} q_{ij} \sum_{k \in \mathfrak{I}} q_{jk} \bar{z}_k + \sum_{j \in \mathfrak{N}} q_{ij} \sum_{k \in \mathfrak{N}} q_{jk} \sum_{l \in \mathfrak{I}} q_{kl} \bar{z}_l + \ldots . \tag{1}$$

Again, following Sutton, let $[M]_{ab}$ denote the ab^{th} entry of any matrix M, $[\mathbf{u}]_a$ denote the a^{th} component of any vector $\mathbf{u}$. Q denote the square matrix with components $[Q]_{ab} = q_{ab}$, $a, b \in \mathfrak{N}$, and $\mathbf{h}$ denote the vector whose components are $[\mathbf{h}]_a = \Sigma_{b \in \mathfrak{I}} q_{ab} \bar{z}_b$, for $a \in \mathfrak{N}$. Then from equation (1):

$$\mathbb{E}[z \mid i] = \left[\sum_{k=0}^{\infty} Q^k \mathbf{h} \right]_i = [(I - Q)^{-1} \mathbf{h}]_i \tag{2}$$

As Sutton showed, the existence of the limit in this equation follows from the fact that Q is the transition matrix for the nonterminal states of an absorbing Markov chain, which, with probability one will ultimately terminate.

During the learning phase, linear TD(λ) generates successive vectors $\mathbf{w}_1^\lambda$, $\mathbf{w}_2^\lambda$, ...,[1] changing $\mathbf{w}^\lambda$ after each complete observation sequence. Define $V_n^\lambda(i) = \mathbf{w}_n^\lambda \cdot \mathbf{x}_i$ as the prediction of the terminal value starting from state i, at stage n in learning. Then, during one such sequence, $V_n^\lambda(i_t)$ are the intermediate predictions of these terminal values, and, abusing notation somewhat, define also $V_n^\lambda(i_{m+1}) = z$, the observed terminal value. Note that in Sutton (1988), Sutton used P_t^n for $V_n^\lambda(i_t)$. TD(λ) changes $\mathbf{w}^\lambda$ according to:

$$\mathbf{w}_{n+1}^\lambda = \mathbf{w}_n^\lambda + \sum_{t=1}^{m} \left\{ \alpha [V_n^\lambda(i_{t+1}) - V_n^\lambda(i_t)] \sum_{k=1}^{t} \lambda^{t-k} \nabla_{\mathbf{w}_n} V_n^\lambda(i_k) \right\} \tag{3}$$

where α is the learning rate.

Sutton showed that TD(1) is just the normal LMS estimator (Widrow & Stearns, 1985), and also proved that the following theorem:

Theorem T For any absorbing Markov chain, for any distribution of starting probabilities μ_i such that there are no inaccessible states, for any outcome distributions with finite expected values $\bar{z}_j$, and for any linearly independent set of observation vectors $\{\mathbf{x}_i \mid i \in \mathfrak{N}\}$, there exists an $\epsilon > 0$ such that, for all positive $\alpha < \epsilon$ and for any initial weight vector, the predictions of linear TD(λ) (with weight updates after each sequence) converge in expected value to the ideal predictions (2); that is, if $\mathbf{w}_n^\lambda$ denotes the weight vector after n sequences have been experienced, then

$$\lim_{n \to \infty} \mathbb{E}[\mathbf{w}_n^\lambda \cdot \mathbf{x}_i] = \mathbb{E}[z \mid i] = [(I - Q)^{-1} \mathbf{h}]_i, \ \forall i \in \mathfrak{N}.$$

is true in the case that $\lambda = 0$. This paper proves theorem T for general λ.

2.2. Localist representation

Equation (3) conflates two issues; the underlying TD(λ) algorithm and the representation of the prediction functions V_n^λ. Even though these will remain tangled in the ultimate proof of convergence, it is beneficial to separate them out, since it makes the operation of the algorithm clearer.

Consider representing V_n^λ as a look-up table, with one entry for each state. This is equivalent to choosing a set of vectors $\mathbf{x}_i$ for which just one component is 1 and all the others are 0 for each state, and no two states have the same representation. These trivially satisfy the conditions of Sutton's theorem, and also make the $\mathbf{w}_n$ easy to interpret, as each component is the prediction for just one state. Using them also prevents generalization. For this representation, the terms $\nabla_{\mathbf{w}_n} V_n^\lambda(i_k)$ in the sum

$$\sum_{k=1}^{t} \lambda^{t-k} \nabla_{\mathbf{w}_n} V_n^\lambda(i_k)$$

just reduce to counting the number of times the chain has visited each state, exponentially weighted in recency by λ. In this case, as in the full linear case, these terms do not depend on n, only on the states the chain visits. Define a characteristic function for state j:

$$\chi_j(k) = \begin{cases} 1 & \text{if } i_k = j \\ 0 & \text{otherwise} \end{cases}$$

and the prediction function $V_n^\lambda(i)$ as the entry in the look-up table for state i at stage n during learning. Then equation (3) can be reduced to its elemental pieces

$$V_{n+1}^\lambda(i) = V_n^\lambda(i) + \sum_{t=1}^{m} \left\{ \alpha[V_n^\lambda(i_{t+1}) - V_n^\lambda(i_t)] \sum_{k=1}^{t} \lambda^{t-k} \chi_i(k) \right\} \tag{4}$$

in which the value for each state is updated separately.

To illustrate this process, take the punctate representation of the states *B*, *C*, *D*, *E* and *F* in figure 1 to be:[2]

$$\begin{matrix} \mathbf{x}_B & \mathbf{x}_C & \mathbf{x}_D & \mathbf{x}_E & \mathbf{x}_F \end{matrix}$$
$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

If the observed sequence is *D*, *C*, *D*, *E*, *F*, *E*, *F*, *G*, then the sums

$$\sum_{k=1}^{t} \lambda^{t-k} \nabla_{\mathbf{w}_n} V_n^\lambda(i_k)$$

after each step are:

$$\begin{matrix} D \\ \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \end{matrix} \begin{matrix} C \\ \begin{pmatrix} 0 \\ 1 \\ \lambda \\ 0 \\ 0 \end{pmatrix} \end{matrix} \begin{matrix} D \\ \begin{pmatrix} 0 \\ \lambda \\ 1 + \lambda^2 \\ 0 \\ 0 \end{pmatrix} \end{matrix} \begin{matrix} E \\ \begin{pmatrix} 0 \\ \lambda^2 \\ \lambda + \lambda^3 \\ 1 \\ 0 \end{pmatrix} \end{matrix} \begin{matrix} F \\ \begin{pmatrix} 0 \\ \lambda^3 \\ \lambda^2 + \lambda^4 \\ \lambda \\ 1 \end{pmatrix} \end{matrix} \begin{matrix} E \\ \begin{pmatrix} 0 \\ \lambda^4 \\ \lambda^3 + \lambda^5 \\ 1 + \lambda^2 \\ \lambda \end{pmatrix} \end{matrix} \begin{matrix} F \\ \begin{pmatrix} 0 \\ \lambda^5 \\ \lambda^4 + \lambda^6 \\ \lambda + \lambda^3 \\ 1 + \lambda^2 \end{pmatrix} \end{matrix}$$

and component i in this sum is clearly

$$\sum_{k=1}^{t} \lambda^{t-k} \chi_i(k)$$

and time t.

2.3. *Contraction mappings*

Watkins (1989) showed that a fruitful way of looking at TD estimators is through dynamic programming and its associated contraction mappings. The method starts from the current prediction function $V_n(i)$, $\forall i \in \mathfrak{N}$, shows how to define a whole collection of statistically better estimators $V_{n+1}(i)$, $\forall i \in \mathfrak{N}$ based on an observed sequence, and then uses for TD(λ) a linear combination of these estimators. Except where explicitly noted, this section follows Watkins exactly—the equations developed have their exact analogues for the linear representation, as will be seen in section 2.5.

Imagine the chain starts at some state i_0, and runs forward through states $i_1, i_2, \ldots$, ultimately absorbing. Define the r-step estimate of i_0 as *either* the estimate $V_n(i_r)$ of state i_r, if the chain is not absorbed after r steps and so $i_r \in \mathfrak{N}$, *or* the observed terminal value z of the sequence, if the chain has absorbed before this time.

Formally, define the random variables

$$\begin{aligned} V^1_{n,i_0} &= \begin{cases} V_n(i_1) & \text{if } i_1 \in \mathfrak{N} \\ z & \text{otherwise} \end{cases} \\ V^2_{n,i_0} &= \begin{cases} V_n(i_2) & \text{if } i_2 \in \mathfrak{N} \\ z & \text{otherwise} \end{cases} \\ &\vdots \\ V^r_{n,i_0} &= \begin{cases} V_n(i_r) & \text{if } i_r \in \mathfrak{N} \\ z & \text{otherwise} \end{cases} \\ &\vdots \end{aligned} \tag{5}$$

where i_1 is the first state accessed by the Markov chain in one particular sequence starting from i_0; i_2 is the second, and so on, and z is the observed terminal value if the chain gets absorbed before time step r is reached. These are random variables, since they depend on the particular sequence of states which will be observed. Naturally, they also depend on the values $V_n(i)$.

The point of these is that if the chain absorbs after completing s steps from i_0, then each of the V^r_{n,i_0}, for $r \geq s$ will be based on the terminal value provided by the world, rather than one derived from the 'bootstraps' $V_n(i)$. V^r_{n,i_0} should therefore on average be more accurate than V_n and so can be used incrementally to improve it. This can be shown by looking at the difference between $\mathbb{E}[V^r_{n,i_0}]$ and $\mathbb{E}[z \mid i_0]$, the ideal predictions.

Here:

$$\mathbb{E}[V^r_{n,i_0}] = \sum_{t_1 \in \mathfrak{I}} q_{i_0 t_1} \bar{z}_{t_1} + \sum_{i_1 \in \mathfrak{N}} Q_{i_0 i_1} \sum_{t_2 \in \mathfrak{I}} q_{i_1 t_2} \bar{z}_{t_2} + \ldots + \sum_{i_{r-1} \in \mathfrak{N}} Q^{r-1}_{i_0 i_{r-1}} \sum_{t_r \in \mathfrak{I}} q_{i_{r-1} t_r} \bar{z}_{t_r} + \sum_{i_r \in \mathfrak{N}} Q^r_{i_0 i_r} V_n(i_r) \tag{6}$$

whereas it can easily be shown that

$$\mathbb{E}[z \mid i_0] = \sum_{t_1 \in \mathfrak{I}} q_{i_0 t_1} \bar{z}_{t_1} + \sum_{i_1 \in \mathfrak{N}} Q_{i_0 i_1} \sum_{t_2 \in \mathfrak{I}} q_{i_1 t_2} \bar{z}_{t_2} + \ldots + \tag{7}$$

$$\sum_{i_{r-1} \in \mathfrak{N}} Q^{r-1}_{i_0 i_{r-1}} \sum_{t_r \in \mathfrak{I}} q_{i_{r-1} t_r} \bar{z}_{t_r} + \sum_{i_r \in \mathfrak{N}} Q^r_{i_0 i_r} \mathbb{E}[z \mid i_r] \tag{8}$$

Therefore,

$$\mathbb{E}[V^r_{n,i_0}] - \mathbb{E}[z \mid i_0] = \sum_{i_r \in \mathfrak{N}} Q^r_{i_0 i_r} (V_n(i_r) - \mathbb{E}[z \mid i_r]) \tag{9}$$

Watkins actually treated a slightly different case, in which the target values of the predictors are based on *discounted* future values whose contribution diminishes exponentially with the time until they happen. In this case it is easier to see how the reduction in error is brought about. His analogue of equation (9) was:

$$\mathbb{E}[V^r_{n,i_0}] - \mathbb{E}[z \mid i_0] = \gamma^r \sum_{i_r \in \mathfrak{N}} Q^r_{i_0 i_r} (V_n(i_r) - \mathbb{E}[z \mid i_r])$$

where $\gamma < 1$ is the discount factor. Since

$$\sum_{i_r \in \mathfrak{N}} Q^r_{i_0 i_r} \leq 1,$$

as Q is the matrix of a Markov chain, Watkins could guarantee that

$$\max_{i_0} |\mathbb{E}[V^r_{n,i_0}] - \mathbb{E}[z|i_0]| \leq \gamma^r \max_{i_r} |V_n(i_r) - \mathbb{E}[z|i_r]|$$

which provides a (weak) guarantee that the error of V^r_n will be less than that of V_n.

The nondiscounted case, when $\gamma = 1$, is somewhat different. Here, for some initial states i_0, there is a nonzero probability that the chain will absorb before finishing r steps. In this case, the value of V^r_{n,i_0}, being z, will be unbiased, and so should provide for error reduction. Even if the chain does not absorb, its value can be no farther from what it should be than is the most inaccurate component of V_n. Although there is no error reduction due to γ, it is guaranteed that

$$\sum_{i_r \in \mathfrak{I}} Q^r_{i_0 i_r} \leq 1$$

with inequality for all those states from which it is possible to absorb within r steps.

This does not ensure that

$$\max_{i_0} |\mathbb{E}[V^r_{n,i_0}] - \mathbb{E}[z|i_0]| < \max_{i_r} |V_n(i_r) - \mathbb{E}[z|i_r]|$$

since the maximum could be achieved, pathologically, at a state from which it is impossible to absorb in only r steps. However, the estimates for the states that are within r steps of absorption will, on average, improve, and this should, again on average, filter back to the other states.

Watkins demonstrated that TD(λ) is based on a weighted average of the V^a_{n,i_0}. Consider

$$V^\lambda_{n,i_0} = (1 - \lambda) \sum_{a=1}^{\infty} \lambda^{a-1} V^a_{n,i_0} \tag{10}$$

which is also a valid estimator of the terminal value starting at i_0. He points out that in choosing the value of λ, there is a tradeoff between the bias caused by the error in V_n, and the variance of the real terminal value z. The higher λ, the more significant are the V^r_n for higher values of r, and the more effect the unbiased terminal values will have. This leads to higher variance and lower bias. Conversely, the lower λ, the less significant are the contributions from higher values of r, and the less the effect of the unbiased terminal values. This leads to smaller variance and greater bias.

It remains to be shown that TD(λ) is indeed based on this combination estimator. Expanding out the sum in equation (10).

$$V^\lambda_{n,i_0} - V_n(i_0) = [V_n(i_1) - V_n(i_0)] + \lambda[V_n(i_2) - V_n(i_1)] + \lambda^2[V_n(i_3) - V_n(i_2)] + \ldots \tag{11}$$

defining $V_n(i_s) = z$ for $s > \max\{t | i_t \in \mathfrak{I}\}$.

The whole point of defining V^{λ}_{n,i_0} is so that it can be used to make V more accurate. The obvious incremental update rule to achieve this has

$$V_{n+1}(i_0) = V_n(i_0) + \alpha[V^{\lambda}_{n,i_0} - V_n(i_0)]. \tag{12}$$

From equation (11) it is apparent that the changes to $V_n(i_0)$ involve summing future values of $V_n(i_{t+1}) - V_n(i_t)$ weighted by powers of λ. Again following Watkins, these differences can be calculated through an activity trace based on the characteristic functions $\chi_i(t)$ that were defined earlier as a way of counting how often and how recently the chain has entered particular states. Using index t for the members of the observed sequence, the on-line version of the TD(λ) rule has

$$V_{t+1}(i) = V_t(i) + \alpha[V_t(i_{t+1}) - V_t(i_t)] \sum_{k=1}^{t} \lambda^{t-k}\chi_i(k). \tag{13}$$

For the problem that Sutton treated, the change to V_n is applied off-line, after a complete sequence through the chain. Therefore, if the states through which the chain passes on one sequence are $i_0, i_1, \ldots, i_{m-1} \in \mathfrak{N}$, and $i_m \in \mathfrak{I}$, it absorbs with terminal value $V_n(i_m) \equiv z$, and V_{n+1} is the new estimator after experiencing the sequence, then

$$V_{n+1}(i_0) = V_n(i_0) + \sum_{t=1}^{m} \alpha[V_n(i_{t+1}) - V_n(i_t)] \sum_{k=1}^{t} \lambda^{t-k}\chi_{i_0}(k)$$

$$V_{n+1}(i_1) = V_n(i_1) + \sum_{t=2}^{m} \alpha[V_n(i_{t+1}) - V_n(i_t)] \sum_{k=1}^{t} \lambda^{t-k}\chi_{i_1}(k)$$

$$\vdots$$

$$V_{n+1}(i_{m-1}) = V_n(i_{m-1}) + \alpha[z - V_n(i_{m-1})] \sum_{k=1}^{m} \lambda^{t-k}\chi_{i_{m-1}}(k),$$

summing over terms where $i_a = i_b$ (so $\chi_{i_a} \equiv \chi_{i_b}$). Note that these expressions are exactly the same as the TD(λ) weight change formula in equation (4).

Thus, the actual TD(λ) algorithm is based on the exponentially weighted sum defined in equation (10) of the outcomes of the V^r_i random variables. The mean contraction properties of these variables will therefore determine the mean contraction properties of the overall TD(λ) estimator.

2.4. Linear representation

The previous subsection considered the TD(λ) algorithm isolated from the representation Sutton used. Although a number of different representations might be employed, the simplest is the linear one he adopted. Identifying the vectors **x** with the states they represent gives

$$V_n(\mathbf{x}) = \mathbf{w}_n \cdot \mathbf{x}$$

where $\mathbf{w}_n$ is the weight vector at stage n of learning.

The basic algorithm is concerned with the V_n^λ predictor random variables rather than how their values can be used to change the initial predictor V_n. Under the new representation, equation (12) no longer makes sense since the states cannot be separated in the appropriate manner. Rather, the information about the error has to be used to update all the weights on which it depends. The appropriate formula, derived from the delta-rule is

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \alpha[V_{n,i_0}^\lambda - V_n(i_0)] \nabla_{\mathbf{w}_n} V_n(i_0)$$

weighting the error due to state i_0 by the vector representation of i_0. Then the equivalent of equation (13) is just Sutton's main TD(λ) equation (3).

More sophisticated representations such as kd-trees (see Omohundro (1987) for a review) or CMACs (Albus, 1975) may lead to faster learning and better generalization, but each requires a separate convergence proof. Dayan (1991) compares the qualities of certain different representations for Barto, Sutton and Watkins' grid task (Barto, Sutton & Watkins, 1990).

2.5. The proof of theorem T

The strategy for proving theorem T is to follow Sutton (1988) in considering the expected value of the new prediction weight vector given the observation of a complete sequence, and to follow Watkins in splitting this change into the components due to the equivalents of the V^r random variables, and then summing them. Mean error reduction over iterations will be assured by the equivalent of equation (9) for the linear representation.

Define the $V_{\cdot,\cdot}^r$ random variables as in equation (5) as

$$V_{n,i_0}^r = \begin{cases} \mathbf{w}_n^r \cdot \mathbf{x}_{i_r} & \text{if } \mathbf{x}_{i_r} \in \mathfrak{N} \\ z & \text{otherwise} \end{cases}$$

where $\mathbf{x}_i$ are identified with the states in the observed sequence, $\mathbf{w}_n^r$ is the current weight vector defining the estimated terminal values, and z is the actual value. Then, after observing the whole sequence, $\mathbf{w}_n^r$ is updated as:

$$\begin{aligned} \mathbf{w}_{n+1}^r &= \mathbf{w}_n^r + \alpha \sum_{\mathbf{x}_i \in \mathfrak{N} \text{ visited}} [V_{n,i}^r - V_n(i)] \nabla_{\mathbf{w}_n} V_n(i) \\ &= \mathbf{w}_n^r + \alpha \sum_{\mathbf{x}_i \in \mathfrak{N} \text{ visited}} [V_{n,i}^r - \mathbf{w}_n \cdot \mathbf{x}_i] \mathbf{x}_i. \end{aligned} \tag{14}$$

An exact parallel of Sutton's proof procedure turns out to apply to $\mathbf{w}^r$. Define η_{ij}^s as the number of times the s-step transition

$$\mathbf{x}_i \to \mathbf{x}_{k_1} \to \mathbf{x}_{k_2} \ldots \to \mathbf{x}_{k_{s-1}} \to \mathbf{x}_j$$

occurs, for any intermediate states $\mathbf{x}_{k_t} \in \mathfrak{N}$.

The sum in equation (14) can be regrouped in terms of source and destination states of the transitions:

$$\begin{aligned}
\mathbf{w}_{n+1}^r = \mathbf{w}_n^r &+ \alpha \sum_{i \in \mathfrak{N}} \sum_{j_r \in \mathfrak{N}} \eta_{ij_r}^r [\mathbf{w}_n^r \cdot \mathbf{x}_{j_r} - \mathbf{w}_n^r \cdot \mathbf{x}_i] \, \mathbf{x}_i \\
&+ \alpha \sum_{i \in \mathfrak{N}} \sum_{j_r \in \mathfrak{I}} \eta_{ij_r}^r [z_{j_r} - \mathbf{w}_n^r \cdot \mathbf{x}_i] \, \mathbf{x}_i \\
&+ \alpha \sum_{i \in \mathfrak{N}} \sum_{j_{r-1} \in \mathfrak{I}} \eta_{ij_{r-1}}^{r-1} [z_{j_{r-1}} - \mathbf{w}_n^r \cdot \mathbf{x}_i] \, \mathbf{x}_i \\
&\vdots \\
&+ \alpha \sum_{i \in \mathfrak{N}} \sum_{j_1 \in \mathfrak{I}} \eta_{ij_1}^1 [z_{j_1} - \mathbf{w}_n^r \cdot \mathbf{x}_i] \, \mathbf{x}_i
\end{aligned} \tag{15}$$

where z_j indicates that the terminal value is generated from the distribution due to state j, and the extra terms are generated by the possibility that, from visiting any $\mathbf{x}_i \in \mathfrak{N}$, the chain absorbs before taking r further steps.

Taking expected values over sequences, for $i \in \mathfrak{N}$

$$\begin{aligned}
\mathbb{E}[\eta_{ij}^r] &= d_i Q_{ij}^r && \text{for } j \in \mathfrak{N} \\
\mathbb{E}[\eta_{ij}^r] &= \sum_{k \in \mathfrak{N}} d_i Q_{ik}^{r-1} q_{kj} && \text{for } j \in \mathfrak{I} \\
\mathbb{E}[\eta_{ij}^{r-1}] &= \sum_{k \in \mathfrak{N}} d_i Q_{ik}^{r-2} q_{kj} && \text{for } j \in \mathfrak{I} \\
&\vdots \\
\mathbb{E}[\eta_{ij}^1] &= d_i q_{ij} && \text{for } j \in \mathfrak{I}
\end{aligned}$$

where d_i is the expected number of times the Markov chain is in state i in one sequence. For an absorbing Markov chain, it is known that the dependency of this on the probabilities μ_i of starting in the various states is:

$$d_i = \sum_{j \in \mathfrak{N}} \mu_j (I - Q)_{ji}^{-1} = [\mu^T (I - Q)^{-1}]_i \tag{16}$$

Substituting into equation (15), after taking expectations on both sides, noting that the dependence of $\mathbb{E}[\mathbf{w}_{n+1}^r \mid \mathbf{w}_n^r]$ on $\mathbf{w}_n^r$ is linear, and using $\bar{\mathbf{w}}$ to denote expected values, a close relation of equation (6) emerges for the linear representation:

$$\bar{\mathbf{w}}_{n+1}^r = \bar{\mathbf{w}}_n^r + \alpha \sum_{i \in \mathfrak{N}} d_i \mathbf{x}_i \left[\sum_{j_r \in \mathfrak{N}} Q_{ij_r}^r (\mathbf{x}_{j_r} \cdot \bar{\mathbf{w}}_n^r) \right.$$

$$- (\mathbf{x}_i \cdot \bar{\mathbf{w}}_n^r) \left\{ \sum_{j_r \in \mathfrak{N}} Q_{ij_r}^r + \sum_{\substack{j_r \in \mathfrak{I}, \\ k \in \mathfrak{N}}} Q_{ik}^{r-1} q_{kj_r} + \ldots + \sum_{j_1 \in \mathfrak{I}} q_{ij_1} \right\}$$

$$\left. + \sum_{\substack{j_r \in \mathfrak{I}, \\ k \in \mathfrak{N}}} Q_{ik}^{r-1} q_{kj_r} \bar{z}_{j_r} + \sum_{\substack{j_{r-1} \in \mathfrak{I}, \\ k \in \mathfrak{N}}} Q_{ik}^{r-2} q_{kj_{r-1}} \bar{z}_{j_{r-1}} + \ldots + \sum_{j_1 \in \mathfrak{I}} q_{ij_1} \bar{z}_{j_1} \right].$$

Define X to be the matrix whose columns are $\mathbf{x}_i$, so $[X]_{ab} = [\mathbf{x}_a]_b$, and D to be the diagonal matrix $[D]_{ab} = \delta_{ab} d_a$, where δ_{ab} is the Kronecker delta. Remembering that $h_i = \Sigma_{j \in \mathfrak{I}} q_{ij} \bar{z}_j$, and converting to matrix form

$$\bar{\mathbf{w}}_{n+1}^r = \bar{\mathbf{w}}_n^r + \alpha XD[Q^r X^{\mathrm{T}} \bar{\mathbf{w}}_n^r - X^{\mathrm{T}} \bar{\mathbf{w}}_n^r + (Q^{r-1} + Q^{r-2} + \ldots + I)\mathbf{h}] \tag{17}$$

since

$$\sum_{j_r \in \mathfrak{N}} Q_{ij_r}^r + \sum_{\substack{j_r \in \mathfrak{I}, \\ k \in \mathfrak{N}}} Q_{ik}^{r-1} q_{kj_r} + \ldots + \sum_{j_1 \in \mathfrak{I}} q_{ij_1} = 1$$

as this covers all the possible options for r-step moves from state i.

Define the correct predictions $[\bar{\mathbf{e}}^*]_i = \mathbb{E}[z \mid i]$; then also, from equation (2),

$$\begin{aligned} \bar{\mathbf{e}}^* &= [\mathbb{E}[z \mid i]] \\ &= \mathbf{h} + Q\mathbf{h} + Q^2\mathbf{h} + \ldots \\ &= (I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h} + Q^r(I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h} + \ldots \\ &= \sum_{k=0}^{\infty} [Q^r]^k (I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h} \\ &= (I - Q^r)^{-1}(I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h} \end{aligned} \tag{18}$$

where the sum converges since the chain is absorbing. This is another way of writing equation (7).

Multiplying equation (17) on the left by X^{T},

$$\begin{aligned} X^{\mathrm{T}} \bar{\mathbf{w}}_{n+1}^r &= X^{\mathrm{T}} \bar{\mathbf{w}}_n^r + \alpha X^{\mathrm{T}} XD[(I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h} + Q^r X^{\mathrm{T}} \bar{\mathbf{w}}_n^r - X^{\mathrm{T}} \bar{\mathbf{w}}_n^r] \\ &= [I - \alpha X^{\mathrm{T}} XD(I - Q^r)] X^{\mathrm{T}} \bar{\mathbf{w}}_n^r + \alpha X^{\mathrm{T}} XD(I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h} \end{aligned}$$

Subtracting $\bar{\mathbf{e}}^*$ from both sides of the equation, and noting that from equation (18) $(I - Q^r)\bar{\mathbf{e}}^* = (I + Q + Q^2 + \ldots + Q^{r-1})\mathbf{h}$, this gives the update rule, which is the equivalent of equation (9):

$$[X^T\bar{\mathbf{w}}_{n+1}^r - \bar{\mathbf{e}}^*] = [I - \alpha X^T XD(I - Q^r)]X^T\bar{\mathbf{w}}_n^r + \alpha X^T XD(I - Q^r)\bar{\mathbf{e}}^* - \bar{\mathbf{e}}^*$$

$$= [I - \alpha X^T XD(I - Q^r)][X^T\bar{\mathbf{w}}_n^r - \bar{\mathbf{e}}^*].$$

The Watkins construction of TD(λ) developed in equation (10) in the previous section reveals that, starting from $\mathbf{w}_n^r = \mathbf{w}_n^\lambda$, $\forall r$,

$$\mathbf{w}_{n+1}^\lambda = (1 - \lambda)\sum_{r=1}^{\infty} \lambda^{r-1}\mathbf{w}_{n+1}^r$$

Therefore, since for $0 < \lambda < 1$, $(1 - \lambda)\Sigma_{r=1}^{\infty} \lambda^{r-1} = 1$,

$$[X^T\bar{\mathbf{w}}_{n+1}^\lambda - \bar{\mathbf{e}}^*] = \left\{(1 - \lambda)\sum_{r=1}^{\infty} \lambda^{r-1}[I - \alpha X^T XD(I - Q^r)]\right\}[X^T\bar{\mathbf{w}}_n^\lambda - \bar{\mathbf{e}}^*]$$

$$= \{I - \alpha X^T XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\}[X^T\bar{\mathbf{w}}_n^\lambda - \bar{\mathbf{e}}^*]$$

where $\bar{\mathbf{w}}^\lambda$ are the expected weights from the TD(λ) procedure. The sum

$$(1 - \lambda)\sum_{r=1}^{\infty} \lambda^{r-1}Q^r = (1 - \lambda)Q[I - \lambda Q]^{-1} \tag{19}$$

converges since $0 < \lambda < 1$.

Define

$$\Delta_\lambda = I - \alpha X^T XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$$

then the truth of theorem T will be shown if it can be demonstrated that $\exists \epsilon > 0$ such that for $0 < \alpha < \epsilon$, $\lim_{n\to\infty} \Delta_\lambda^n = 0$. For then $[X^T\bar{\mathbf{w}}_n^r - \bar{\mathbf{e}}^*] \to \mathbf{0}$ as $n \to \infty$, and all the estimates will tend to be correct.

Almost all of Sutton's (1988) proof of this applies *mutatis mutandis* to the case that $\lambda \neq 0$, always provided the crucial condition holds that X has full rank. For completeness, the entire proof is given in the appendix. Overall it implies that the expected values of the estimates will converge to their desired values as more sequences are observed under the conditions stated in theorem T.

2.6 Non-independence of the $\mathbf{x}_i$

In moving from Watkins' representation-free proof to Sutton's treatment of the linear case, one assumption was that the $\mathbf{x}_i$, the vectors representing the states, were independent. If they are not, so that matrix X does not have full rank, the proof breaks down. $D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$ is still positive, however $X^{\mathrm{T}}XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$ will no longer have a full set of eigenvalues with positive real parts, since the null subspace

$$Y = \{\mathbf{y} \mid XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\mathbf{y} = \mathbf{0}\} \neq \{\mathbf{0}\}$$

is not empty. Any nonzero member of this is an eigenvector with eigenvalue 0 of $X^{\mathrm{T}}XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$.

Saying what will happen to the expected values of the weights turns out to be easier than understanding it. Choose a basis:

$$\{\mathbf{b}_1, \ldots, \mathbf{b}_p, \mathbf{b}_{p+1}, \ldots, \mathbf{b}_n\} \text{ for } \Re^n,$$

with $\mathbf{b}_i \in Y$, for $1 \leq i \leq p$ being a basis for Y.

Then the proof in the appendix applies exactly to $\mathbf{b}_{p+1}, \ldots, \mathbf{b}_n$; that is there exists some $0 < \epsilon < 1$ such that:

$$\lim_{n \to \infty} [I - \alpha X^{\mathrm{T}}XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})]^n \mathbf{b}_i = \mathbf{0}, \text{ for } p < i \leq n, \text{ and } 0 < \alpha < \epsilon.$$

Also,

$$[I - \alpha X^{\mathrm{T}}XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})]^n \mathbf{b}_i = \mathbf{b}_i, \text{ for } 1 \leq i \leq p$$

by the definition of Y.

Writing

$$X^{\mathrm{T}}\bar{\mathbf{w}}_0^{\lambda} - \bar{\mathbf{e}}^* = \sum_{i=1}^{n} \beta_i \mathbf{b}_i,$$

then

$$\begin{aligned} X^{\mathrm{T}}\bar{\mathbf{w}}_n^{\lambda} - \bar{\mathbf{e}}^* &= [I - \alpha X^{\mathrm{T}}XD(I - Q^r)]^n [X^{\mathrm{T}}\bar{\mathbf{w}}_0{}^{\lambda} - \bar{\mathbf{e}}^*] \\ &= [I - \alpha X^{\mathrm{T}}XD(I - Q^r)]^n \left[\sum_{i=1}^{n} \beta_i \mathbf{b}_i \right] \\ &\to \sum_{i=1}^{p} \beta_i \mathbf{b}_i, \text{ as } n \to \infty \end{aligned}$$

and so

$$XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})[X^{\mathrm{T}}\bar{\mathbf{w}}_n^{\lambda} - \bar{\mathbf{e}}^*] \to \mathbf{0} \text{ as } n \to \infty. \tag{20}$$

To help understand this result, consider the equivalent for the LMS rule, TD(1). There

$$XD[X^{\mathrm{T}}\bar{\mathbf{w}}_n^{1} - \bar{\mathbf{e}}^*] \to \mathbf{0} \text{ as } n \to \infty. \tag{21}$$

and so, since D is symmetric,

$$\frac{\partial}{\partial \bar{\mathbf{w}}_n^{1}} [X^{\mathrm{T}}\bar{\mathbf{w}}_n^{1} - \bar{\mathbf{e}}^*]^{\mathrm{T}} D[X^{\mathrm{T}}\bar{\mathbf{w}}_n^{1} - \bar{\mathbf{e}}^*] = X(D + D^{\mathrm{T}})[X^{\mathrm{T}}\bar{\mathbf{w}}_n^{1} - \bar{\mathbf{e}}^*] \tag{22}$$

$$= 2XD[X^{\mathrm{T}}\bar{\mathbf{w}}_n^{1} - \bar{\mathbf{e}}^*] \tag{23}$$

$$\to \mathbf{0} \text{ as } n \to \infty, \tag{24}$$

by equation (21). For weights $\mathbf{w}$, the square error for state i is $|[X^{\mathrm{T}}\mathbf{w} - \bar{\mathbf{e}}^*]|_i^2$, and the expected number of visits to i in one sequence is d_i. Therefore the quadratic form

$$[X^T\mathbf{w} - \bar{\mathbf{e}}^*]^T D[X^T\mathbf{w} - \bar{\mathbf{e}}^*]$$

is just the loaded square error between the predictions at each state and their desired values, where the loading factors are just the expected frequencies with which the Markov chain hits those states. The condition in equation (24) implies that the expected values of the weights tend to be so as to minimize this error.

This does not happen in general for $\lambda \neq 1$. Intuitively, bias has returned to haunt. For the case where X is full rank, Sutton shows that it is harmless to use the inacurrate estimates from the next state $\mathbf{x}_{i_{t+1}} \cdot \mathbf{w}$ to criticize the estimates for the current state $\mathbf{x}_{i_t} \cdot \mathbf{w}$. Where X is not full rank, these successive estimates become biased on account of what might be deemed their 'shared' representation. The amount of extra bias is then related to the amount of sharing and the frequency with which the transitions happen from one state to the next.

Formalizing this leads to a second issue; the interaction between the two statistical processes of calculating the mean weight and calculating the expected number of transitions. Comparing equations (20) and (21), one might expect

$$\lim_{n\to\infty} \frac{\partial}{\partial \bar{\mathbf{w}}_n^{\lambda}} [X^{\mathrm{T}}\bar{\mathbf{w}}_n^{\lambda} - \bar{\mathbf{e}}^*]^{\mathrm{T}} D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})[X^{\mathrm{T}}\bar{\mathbf{w}}_n^{\lambda} - \bar{\mathbf{e}}^*] = \mathbf{0} \tag{25}$$

However, the key step in proving equation (24) was the transition between equations (22) and (23), which relied on the symmetry of D. Since Q is not in general symmetric, this will not happen.

Defining

$$\begin{aligned}\mathbf{g}(\mathbf{w}') &= \frac{\partial}{\partial \mathbf{w}} [X^{\mathrm{T}}\mathbf{w} - \bar{\mathbf{e}}^*]^{\mathrm{T}} D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})[X^{\mathrm{T}}\mathbf{w}' - \bar{\mathbf{e}}^*] \\ &= XD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})[X^{\mathrm{T}}\mathbf{w}' - \bar{\mathbf{e}}^*]\end{aligned} \tag{26}$$

all that will actually happen is that $\mathbf{g}(\bar{\mathbf{w}}_n^{\lambda}) \to \mathbf{0}$ as $n \to \infty$.

Although the behavior described by equation (25) is no more satisfactory than that described by equation (26), it is revealing to consider what happens if one attempts to arrange for it to hold. This can be achieved by 'completing' the derivative, i.e., by having a learning rule whose effect is

$$[X^{\mathrm{T}}\bar{\mathbf{w}}_{n+1}^{\lambda} - \bar{\mathbf{e}}^*] =$$

$$\left\{ I - \alpha X^T X \left(D - \frac{1 - \lambda}{2} [DQ(I - \lambda Q)^{-1} + (I - \lambda Q^{\mathrm{T}})^{-1} Q^{\mathrm{T}} D^{\mathrm{T}}] \right) \right\} [X^{\mathrm{T}}\bar{\mathbf{w}}_n^{\lambda} - \bar{\mathbf{e}}^*]$$

The Q^T term effectively arranges for backwards as well as forwards learning to occur, so that not only would state i_t adjust its estimate to make it more like state i_{t+1}, but also state i_{t+1} would adjust its estimate to make it more like state i_t.

Werbos (1990) and Sutton (personal communication) both discussed this point in the context of the gradient descent of TD(λ) rather than its convergence for non-independent $\mathbf{x}_i$. Werbos presented an example based on a learning technique very similar to TD(0), in which completing the derivative in this manner makes the rule converge away from the true solution. He faulted this procedure for introducing the unhelpful correlations between the learning rule and the random moves from one state to the next which were mentioned above. He pointed out the convergence in terms of functions $\mathbf{g}$ in equation (26) in which the $\mathbf{w}'$ weights are fixed.

Sutton presented an example to help explain the result. At first sight, augmenting TD(λ) seems quite reasonable; after all it could quite easily happen by random chance of the training sequences that the predictions for one state are more accurate than the predictions for the next at some point. Therefore, training the second to be more like the first would be helpful. However, Sutton pointed out that time and choices always move forward, not backwards. Consider the case shown in figure 2, where the numbers over the arrows represent the transition probabilities, and the numbers at the terminal nodes represent terminal absorbing values.

Here, the value of state A is reasonably 1/2, as there is 50% probability of ending up at either Y or Z. The value of state B, though, should be 1, as the chain is certain to end up at Y. Training forwards will give this, but training backwards too will make the value of B tend to 3/4. In Werbos' terms, there are correlations between the weights and the possible transitions that count against the augmented term. Incidentally, this result does not affect TD(1), because the training values, being just the terminal value for the sequence, bear no relation to the transitions themselves, just the number of times each state is visited.

Coming back to the case where X is not full rank. TD(λ) for $\lambda \neq$ will still converge, but away from the 'best' value, to a degree that is determined by the matrix

$$(I - (1 - \lambda)Q[I - \lambda Q]^{-1}).$$

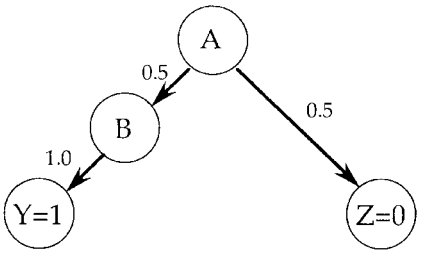


Figure 2. Didactic example of the pitfalls of backwards training. If Y and Z are terminal states with values 1 and 0 respectively, what values should be assigned to states A and B respectively.

3. Convergence with probability one

Sutton's proof and the proofs in the previous section accomplish only the nadir of stochastic convergence, *viz* convergence of the mean, rather than the zenith, *viz* convergence with probability one. Watkins (1989) proved convergence with probability one for a form of prediction and action learning he called $\mathcal{Q}$-learning. This section shows that this result can be applied almost directly to the discounted predictive version of TD(0), albeit without the linear representations, and so provides the first strong convergence proof for a temporal difference method.

Like dynamic programming (DP), $\mathcal{Q}$-learning combines prediction and control. Consider a controlled, discounted, nonabsorbing Markov-process, i.e., one in which at each state $i \in \mathfrak{N}$ there is a finite set of possible actions $a \in \mathfrak{A}$. Taking one action leads to an immediate reward, which is a random variable $r_i(a)$ whose distribution depends on both i and a, and a stochastic transition according to a Markov matrix $\mathcal{P}_{ij}(a)$ for $j \in \mathfrak{N}$. If an agent has some policy $\pi(i) \in \mathfrak{A}$, which determines which action it would perform at state i, then, defining the value of i under π as $V^\pi(i)$, this satisfies:

$$V^\pi(i) = \mathbb{E}[r_i(\pi(i))] + \gamma \sum_{j \in \mathfrak{N}} \mathcal{P}_{ij}(\pi(i)) V^\pi(j),$$

where γ is the discount factor. Define the $\mathcal{Q}$ value of state i and action a under policy π as:

$$\mathcal{Q}^\pi(i, a) = \mathbb{E}[r_i(a)] + \gamma \sum_{j \in \mathfrak{N}} \mathcal{P}_{ij}(a) V^\pi(j),$$

which is the value of taking action a at i followed by policy π thereafter. Then the theory of DP (Bellman & Dreyfus, 1962) implies that a policy which is at least as good as π is to take the action a^* at state i where $a^* = \text{argmax}_b\{\mathcal{Q}^\pi(i, b)\}$, and to follow π in all other states. In this fact lies the utility of the $\mathcal{Q}$ values. For discounted problems, it turns out that there is at least one optimal policy π^*; define $\mathcal{Q}^*(i, a) = \mathcal{Q}^{\pi^*}(i, a)$.

$\mathcal{Q}$-learning is a method of determining $\mathcal{Q}^*$, and hence an optimal policy, based on exploring the effects of actions at states. Consider a sequence of observations (i_n, a_n, j_n, z_n),

where the process at state i_n is probed with action a_n, taking it to state j_n and giving reward z_n. Then define recursively:

$$\mathcal{Q}_{n+1}(i, a) = \begin{cases} (1 - \alpha_n)\mathcal{Q}_n(i, a) + \alpha_n(z_n + \gamma U_n(j_n)) & \text{if } i = i_n \text{ and } a = a_n, \\ \mathcal{Q}_n(i, a) & \text{otherwise} \end{cases} \tag{27}$$

for any starting values $\mathcal{Q}_0(i, a)$, where $U_n(j_n) = \max_b\{\mathcal{Q}_n(j_n, b)\}$. The α_n are a set of learning rates that obey standard stochastic convergence criteria:

$$\sum_{k=1}^{\infty} \alpha_{n^k(i,a)} = \infty, \sum_{k=1}^{\infty} \alpha^2_{n^k(i,a)} < \infty, \forall i \in \mathfrak{N}, a \in \mathfrak{A} \tag{28}$$

where $n^k(i, a)$ is the k^{th} time that $i_n = i$ and $a_n = a$. Watkins (1989) proved that if, in addition, the rewards are all bounded, then, with probability one:

$$\lim_{n\to\infty} \mathcal{Q}_n(i, a) = \mathcal{Q}^*(i, a),$$

Consider a degenerate case of a controlled Markov process in which there is only one action possible from every state. In that case, the $\mathcal{Q}^\pi$, V^π, and the (similarly defined) U^π values are exactly the same and equal to $\mathcal{Q}^*$, and equation (27) is exactly the on-line form of TD(0) for the case of a nonabsorbing chain in which rewards (i.e., the terminal values discussed above in the context of absorbing Markov chains) arrive from every state rather than just some particular set of absorbing states. Therefore, under the conditions of Watkins' theorem, the on-line version of TD(0) converges to the correct predictions, with probability one.

Although clearly a TD procedure, there are various differences between this and the one described in the previous section. Here, learning is on-line, that is the $V(=\mathcal{Q})$ values are changed for every observation. Also, learning need not proceed along an observed sequence—there is no requirement that $j_n = i_{n+1}$, and so uncoupled or disembodied moves can be used.[3] The conditions in equation (28) have as a consequence that every state must be visited infinitely often. Also note that Sutton's proof, since it is confined to showing convergence in the mean, works for a fixed learning rate α, whereas Watkins', in common with other stochastic convergence proofs, requires α_n to tend to 0.

Also, as stated, the $\mathcal{Q}$-learning theorem only applies to discounted, nonabsorbing, Markov chains, rather than the absorbing ones with $\gamma=1$ of the previous section. $\gamma<1$ plays the important rôle in Watkins' proof of bounding the effect of early $\mathcal{Q}_n$ values. It is fairly easy to modify his proof to the case of an absorbing Markov chain with $\gamma=1$, as the ever increasing probability of absorption achieves the same effect. Also, the conditions of Sutton's theorem imply that every nonabsorbing state will be visited infinitely often, and so it suffices to have one set of α_i that satisfy the conditions in (28) and apply them sequentially for each visit to each state in the normal running of the chain.

4. Conclusions

This paper has used Watkins' analysis of the relationship between temporal difference (TD) estimation and dynamic programming to extend Sutton's theorem that TD(0) prediction converges in the mean, to the case of theorem T; TD(λ) for general λ. It also demonstrated that if the vectors representing the states are not linearly independent, then TD(λ) for $\lambda \neq 1$ converges to a different solution from the least mean squares algorithm.

Further, it has applied a special case of Watkins' theorem that Q-learning, his method of incremental dynamic programming, converges with probability one, to show that TD(0) using a localist state representation, also converges with probability one. This leaves open the question of whether TD(λ), with punctate or distributed representations, also converges in this manner.

Appendix: Existence of appropriate α

Defining

$$\Delta_\lambda = I - \alpha X^T X D(I - (1 - \lambda)Q[I - \lambda Q]^{-1}),$$

it is necessary to show that there is some ϵ such that for $0 < \alpha < \epsilon$, $\lim_{n\to\infty} \Delta_\lambda^n = 0$. In the case that $\lambda = 0$ (for which this formula remains correct), and X has full rank, Sutton proved this on pages 26–28 of (Sutton, 1988), by showing successively that $D(I - Q)$ is positive, that $X^T X D(I - Q)$ has a full set of eigenvalues all of whose real parts are positive, and finally that α can thus be chose such that all eigenvalues of $I - \alpha X^T X D(I - Q)$ are less than 1 in modulus. This proof requires little alteration to the case that $\lambda \neq 0$, and its path will be followed exactly.

The equivalent of $D(I - Q)$ is $D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$. This will be positive definite, according to a lemma by Varga (1962) and an observation by Sutton, if

$$S = D(I - (1 - \lambda)Q[I - \lambda Q]^{-1}) + \{D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\}^T$$

can be shown to be strictly diagonally dominant with positive diagonal entries. This is the part of the proof that differs from Sutton, but even here, its structure is rather similar.

Define

$$S_r = D(I - Q^r) + \{D(I - Q^r)\}^T.$$

Then

$$\begin{aligned} [S_r]_{ii} &= [D(I - Q^r)]_{ii} + [\{D(I - Q^r)\}^T]_{ii} \\ &= 2d_i[I - Q^r]_{ii} \\ &= 2d_i(1 - [Q^r]_{ii}) \\ &> 0, \end{aligned}$$

since Q is the matrix of an absorbing Markov chain, and so Q^r has no diagonal elements ≥ 1. Therefore S_r has positive diagonal elements.

Also, for $i \neq j$,

$$\begin{aligned}[S_r]_{ij} &= d_i[I - Q^r]_{ij} + d_j[I - Q^r]_{ji} \\ &= -d_i[Q^r]_{ij} - d_j[Q^r]_{ji} \\ &\leq 0\end{aligned}$$

since all the elements of Q, and hence also those of Q^r, are positive.

In this case, S_r will be strictly diagonally dominant if, and only if, $\Sigma_j[S_r]_{ij} \geq 0$, with strict inequality for some i.

$$\begin{aligned}\sum_j [S_r]_{ij} &= \sum_j d_i[I - Q^r]_{ij} + \sum_j d_j[I - Q^r]_{ji} \\ &= d_i \sum_j [I - Q^r]_{ij} + [\mathbf{d}^T(I - Q^r)]_i \\ &= d_i \left(1 - \sum_j [Q^r]_{ij}\right) + [\boldsymbol{\mu}^T(I - Q)^{-1}(I - Q^r)]_i \qquad (29) \\ &= d_i \left(1 - \sum_j [Q^r]_{ij}\right) + [\boldsymbol{\mu}^T(I + Q + Q^2 + \ldots + Q^{r-1})]_i \qquad (30) \\ &\geq 0 \qquad (31)\end{aligned}$$

where equation (29) follows from equation (16), equation (30) holds since

$$I - Q^r = (I - Q)(I + Q + Q^2 + \ldots + Q^{r-1})$$

and equation (31) holds since $\Sigma_j[Q^r]_{ij} \leq 1$, as the chain is absorbing, and $[Q^s]_{ij} \geq 0$, $\forall s$. Also, there exists at least one i for which $\mu_i > 0$, and the inequality is strict for that i.

Since S_r is strictly diagonally dominant for all $r \geq 1$,

$$\begin{aligned}S_\lambda &= (1 - \lambda) \sum_{r=1}^{\infty} \lambda^{r-1} S_r \\ &= D(I - (1 - \lambda)Q[I - \lambda Q]^{-1}) + \{D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\}^T \\ &= S\end{aligned}$$

is strictly diagonally dominant too, and therefore $D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$ is positive definite.

The next stage is to show that $X^TXD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$ has a full set of eigenvalues all of whose real parts are positive. X^TX, D and $(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$ are all nonsingular, which ensures that the set is full. Let ψ and $\mathbf{u}$ be any eigenvalue-eigenvector pair, with $\mathbf{u} = \mathbf{a} + \mathbf{b}i$ and $\mathbf{v} = (X^TX)^{-1}\mathbf{u} \neq \mathbf{0}$, so $\mathbf{u} = (X^TX)\mathbf{v}$. Then

$$\begin{aligned}\mathbf{u}^*D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\mathbf{u} &= \mathbf{v}^*X^TXD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\mathbf{u} \\ &= \mathbf{v}^*\psi\mathbf{u} \\ &= \psi\mathbf{v}^*(X^TX)\mathbf{v} \\ &= \psi(X\mathbf{v})^*X\mathbf{v}\end{aligned}$$

where '*' indicates conjugate transpose. This implies that

$$\mathrm{Re}[\mathbf{u}^*D(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\mathbf{u}] = \mathrm{Re}(\psi(X\mathbf{v})^*X\mathbf{v})$$

or equivalently,

$$\begin{aligned}\{(X\mathbf{v})^*X\mathbf{v}\}\mathrm{Re}[\psi] &= \mathbf{a}^TD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\mathbf{a} \\ &\quad + \mathbf{b}^TD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})\mathbf{b}.\end{aligned}$$

Since the right side (by positive definiteness) and $(X\mathbf{v})^*X\mathbf{v}$ are both strictly positive, the real part of ψ must be strictly positive too.

Furthermore, $\mathbf{u}$ must also be an eigenvector of

$$I - \alpha X^TXD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$$

since

$$\begin{aligned}[I - \alpha X^TXD(I - (1 - \lambda)Q[I - \lambda Q]^{-1}]\mathbf{u} &= \mathbf{u} - \alpha\psi\mathbf{u} \\ &= (1 - \alpha\psi)\mathbf{u}.\end{aligned}$$

Therefore, all the eigenvalues of $I - \alpha X^TXD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})$ are of the form $1 - \alpha\psi$ where $\psi \equiv \nu + \phi i$ has positive ν. Take

$$0 < \alpha < \frac{2\nu}{\nu^2 + \phi^2},$$

for all eigenvalues ψ, and then all the eigenvalues $1 - \alpha\psi$ of the iteration matrix are guaranteed to have modulus less than one. By another theorem of Varga (1962)

$$\lim_{n\to\infty} [I - \alpha X^TXD(I - (1 - \lambda)Q[I - \lambda Q]^{-1})]^n = 0.$$

Acknowledgments

This paper is based on a chapter of my thesis (Dayan, 1991). I am very grateful to Andy Barto, Steve Finch, Alex Lascarides, Satinder Singh, Chris Watkins, David Willshaw, the large number of people who read drafts of the thesis, and particularly Rich Sutton and two anonymous reviewers for their helpful advice and comments. Support was from SERC. Peter Dayan's current address is CNL, The Salk Institute, P.O. Box 85800, San Diego, CA 92186-5800.

Notes

1. Here and subsequently, a superscript λ is used to indicate a TD(λ)-based estimator.
2. States A and G are absorbing and so are not represented.
3. This was one of Watkins' main motivations, as it allows his system to learn about the effect of actions it believes to be suboptimal.

References

Albus, J.S. (1975). A new approach to manipulator control: The Cerebellar Model Articulation Controller (CMAC). *Transactions of the ASME: Journal of Dynamical Systems, Measurement and Control, 97,* 220–227.

Barto, A.G., Sutton, R.S. & Anderson, C.W. (1983). Neuronlike elements that can solve difficult learning problems. *IEEE Transactions on Systems, Man, and Cybernetics, 13,* 834–846.

Barto, A.G., Sutton, R.S. & Watkins, C.J.C.H. (1990). Learning and sequential decision making. In M. Gabriel & J. Moore (Eds.), *Learning and computational neuroscience: Foundations of adaptive networks.* Cambridge, MA: MIT Press, Bradford Books.

Bellman, R.E. & Dreyfus, S.E. (1962). *Applied dynamic programming.* RAND Corporation.

Dayan, P. (1991). *Reinforcing connectionism: Learning the statistical way.* Ph.D. Thesis, University of Edinburgh, Scotland.

Hampson, S.E. (1983). *A neural model of adaptive behavior.* Ph.D. Thesis. University of California, Irvine, CA.

Hampson, S.E. (1990). *Connectionistic problem solving: computational aspects of biological learning.* Boston, MA: Birkhäuser Boston.

Holland, J.H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R.S. Michalski, J.G. Carbonell & T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach, 2.* Los Altos, CA: Morgan Kaufmann.

Klopf, A.H. (1972). *Brain function and adaptive systems—A heterostatic theory.* Air Force Research Laboratories Research Report, AFCRL-72-0164. Bedford, MA.

Klopf, A.H. (1982). *The hedonistic neuron: A theory of memory, learning, and intelligence.* Washington, DC: Hemisphere.

Michie. D. & Chambers, R.A. (1968). BOXES: An experiment in adaptive control. *Machine Intelligence, 2,* 137–152.

Moore, A.W. (1990). *Efficient memory-based learning for robot control.* Ph.D. Thesis, University of Cambridge Computer Laboratory, Cambridge, England.

Omohundro, S. (1987). Efficient algorithms with neural network behaviour. *Complex Systems, 1,* 273–347.

Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. Reprinted in E.A. Feigenbaum & J. Feldman (Eds.) (1963). *Computers and thought.* McGraw-Hill.

Samuel, A.L. (1967). Some studies in machine learning using the game of checkers II: Recent progress. *IBM Journal of Research and Development, 11,* 601–617.

Sutton, R.S. (1984). *Temporal credit assignment in reinforcement learning.* Ph.D. Thesis, University of Massachusetts, Amherst, MA.

Sutton, R.S. (1988). Learning to predict by the methods of temporal difference. *Machine Learning, 3,* 9-44.

Varga, R.S. (1962). *Matrix iterative analysis.* Englewood Cliffs, NJ: Prentice-Hall.

Watkins, C.I.C.H. (1989). *Learning from delayed rewards.* Ph.D. Thesis. University of Cambridge, England.

Werbos, P.J. (1990). Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks, 3,* 179–189.

Widrow, B. & Stearns, S.D. (1985). *Adaptive signal processing.* Englewood Cliffs, NJ: Prentice-Hall.

Witten, I.H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control, 34,* 286–295.

Machine Learning, 8, 363-395 (1992)

A Reinforcement Connectionist Approach to Robot Path Finding in Non-Maze-Like Environments

JOSÉ DEL R. MILLÁN j_millan@jrc.it
Institute for System Engineering and Informatics, Commission of the European Communities, Joint Research Centre, TP 361, 21020 ISPRA (VA), Italy

CARME TORRAS torras@ic.upc.es
Institut de Cibernètica (CSIC-UPC), Diagonal, 647, 08028 Barcelona, Spain

Abstract. This paper presents a reinforcement connectionist system which finds and learns the suitable situation-action rules so as to generate feasible paths for a point robot in a 2D environment with circular obstacles. The basic reinforcement algorithm is extended with a strategy for discovering stable solution paths. Equipped with this strategy and a powerful codification scheme, the path-finder (i) learns quickly, (ii) deals with continuous-valued inputs and outputs, (iii) exhibits good noise-tolerance and generalization capabilities, (iv) copes with dynamic environments, and (v) solves an instance of the path finding problem with strong performance demands.

Keywords. Connectionism, reinforcement learning, robot path finding, stability, reactive systems

1. Introduction

An important problem in robotics is that of generating a path between initial and goal robot configurations that avoids collisions with obstacles. This is called the *robot path finding problem*. Many attempts at solving the problem have been made in the fields of Artificial Intelligence and Computational Geometry (for reviews, see: Brady, et al., 1982; Whitesides, 1985; Yap, 1987; Torras, 1990). All these approaches are based on *planning*. The time complexity of exact geometric approaches grows exponentially with the number of degrees of freedom of robot motion (Canny, 1988), thus being of practical use only when this number is very low. This fact has led to the emergence of numerous heuristic approaches, which either rely on potential fields (Khatib, 1986) or carry out a search through a state space. These approaches trade reliability for speed, in that they do not guarantee to find a solution when it exists and they can even produce unfeasible solutions because of the use of discretized representations.

There have been some attempts at combining exact and heuristic approaches, so as to exploit their respective strong points and minimize their deficiencies. Along this line, Donald (1987) proposes to carry out a local heuristic search process on an algebraic (exact) model of configuration space,[1] while Ilari and Torras (1990) propose a two-stage process, where first an exact model of *physical space*—i.e., disregarding both the shape and the kinematics of the robot—is used to plan a path and then this path is refined through local search to conform to a *trajectory* in *configuration space*.

The above two-stage decomposition of the path finding problem suggests a possible separation between the path-planning and motion-control aspects of the problem. In *path planning*,

the environment is considered at a coarse enough level of detail to be assumed fixed and known a priori—through access to a map—and the goal is to determine, prior to its execution, a sequence of displacements in physical space for reaching a given place. Once a path through physical space is planned at this coarse level, a *high-level motion control* problem arises since commands must be sent to the different motors to make the robot follow the path previously computed, while transforming a discrete path in physical space into a continuous, obstacle-avoiding trajectory in configuration space. Note that, with this problem decomposition, it seems natural to include within motion control the possibility to accommodate for slight deviations in the positions of obstacles, so that it is no longer required that the environment be completely known a priori and static.

In Torras (1990), we have argued that motion planning and motion control involve two different types of processing. Global planning involves reasoning *symbolically* upon an explicit representation of the environment, whilst local obstacle-avoidance capabilities rely on *subsymbolic* pattern processing. This distinction applies to learning as well. Learning to plan would be done symbolically (at a central level), while the learning of reflexes would be carried out at a subsymbolic peripheral level.

This paper focuses on the second aspect above, namely the *subsymbolic learning of obstacle-avoidance reflexes* for an instance of the robot path finding problem characterized by (i) a continuous set of robot configurations and of robot actions, (ii) a partially unknown and dynamic environment, (iii) a need to react in real-time, and (iv) strong performance demands such as finding short paths with wide clearances. Specifically, we present a *reinforcement-based connectionist system* able to generate feasible paths for a mobile robot in a non-maze-like 2D environment, while appropriately dealing with the four problem characteristics above. By saying "non-maze-like" we stress that finding a path in that environment does not require sophisticated planning capabilities, but mainly obstacle-avoidance skills. Maze-like environments require the previous concourse of a path-planner that provides the connectionist system with a series of subgoals along a feasible path, so that the environment around every two consecutive subgoals becomes non-maze-like.

Discovering suitable obstacle-avoidance reflexes by using only a reinforcement signal is a very general approach whose simplest formulation could be characterized as a *weak search method*. This means that reinforcement methods have theoretically limited learning abilities; i.e., they might require heavy learning phases and they might be unable to capture complex features of the problem. These theoretical limitations can be overcome if domain-specific heuristics are incorporated into the basic reinforcement-based search method (Langley, 1985). The *codification scheme* adopted in the present work and the *algorithm used to discover stable solution paths* are instances of such heuristics for the path finding domain. The algorithm allows to greatly *speed up learning* and to deal with *continuous-valued actions*. To the best of our knowledge, this is the first reinforcement system in which continuous-valued actions are used in conjunction with a critic. The codification scheme, besides contributing to solving the problem in a short time, is responsible for the *noise-tolerance* and *generalization* capabilities, for satisfying the *strong performance demands* concerning path length and clearance and, partially, for the ability to cope with *dynamic environments* exhibited by the path-finder.

The paper is structured as follows. In Section 2, previous works relevant to the path-finder developed are reviewed. These are connectionist approaches that either deal with

the same particular problem, namely path finding, or tackle the general class of associative reinforcement learning (ARL) problems. The relation of the latter approaches to the system developed becomes clear in Section 3, where path finding is formulated as an ARL problem. Section 4 is devoted to the description of the connectionist path-finder developed. The results of an experimental comparative study of different versions of the general learning rule adopted, as well as the simulation results obtained with the path-finder equipped with the best of such versions are presented next in Section 5. Finally, in Section 6, some conclusions from this work are provided and future work is addressed.

2. Previous work

A general approach to path finding in partially unknown environments is *to build a mapping from perceived situations to correct actions, and iterate this mapping until the goal is reached.* Systems that use this kind of *situation-action rules* are known as *reactive* systems.

Reactive systems normally rely on knowledge-based techniques. However, most of them are not adaptive, i.e., they do not learn (Agre & Chapman, 1987; Arkins, 1987; Schoppers, 1987; Blythe & Mitchell, 1989). In some systems, the situation-action rules are preprogrammed explicitly by their designers (Agre & Chapman, 1987; Arkins, 1987). In other systems, the rules result from a compilation of previous planning activities (Schoppers, 1987; Blythe & Mitchell, 1989). In any case, the main limitation of the knowledge-based systems developed up to now is the need to specify actions to be taken in all possible situations the agent may find itself in.

The knowledge-based reactive systems able to construct by themselves an internal model of the environment have only been applied to simple tasks (Rivest & Schapire, 1987). Mozer and Bachrach (1989) have implemented the Rivest and Schapire's system in a connectionist network, which has been shown to perform better that the original symbolic version in several respects.

Brooks' approach (1986) is a novel way of building reactive systems. Its originality is based on the subsumption architecture and also on hard-wiring sensors to effectors rather than representing the rules. This approach does not overcome the above-mentioned limitation, but it emphasizes a key issue: reactive behavior should involve parallel distributed processing.

2.1. Connectionist approaches to path finding

Subsymbolic systems—connectionist or not—process information in a massively-parallel distributed manner. In Torras (1990) and Millán and Torras (1991a), we stress that subsymbolic techniques should be used to handle the motion control part of robot path finding. Previous subsymbolic attempts to tackle the problem above fall into two categories: constructive and pattern-matching systems.

Constructive path-finders codify the environment through connectionist techniques so as to permit the exploration of configuration space in parallel. Some of them (Steels, 1988) work on a direct discretized representation of the environment, while others capture complex

relationships in the environment through recourse to a learning phase (Jorgensen, 1987; Graf & LaLonde, 1988; Mel, 1989). Constructive path-finders, however, suffer from the inherent limitations of needing to carry out a search through a state space, namely off-line processing, static environment and discretization.

Pattern-matching connectionist path-finders are *adaptive* reactive systems, where the mapping from perceived situations to correct actions is learned as paths are generated. They do not need to codify explicitly all the situation-action rules, since they exhibit strong generalization capabilities. However, these systems have only been applied to relatively simple instances of the robot path finding problem. The system Dyna (Sutton, 1990) is the only pattern-matching path-finder that considers the existence of obstacles in the robot's workspace. Although it tackles the path-finding problem as an ARL problem (see the following subsection), as the system presented in this paper does, it has some of the limitations that we would like to overcome. Thus, Dyna relies on a discretization of the workspace, considers a limited set of possible robot steps, and assumes a static environment. In addition, it needs to perform a new learning phase each time the workspace changes.

For a detailed review of these previous connectionist approaches to robot path finding, together with an assessment of their merits and limitations, see Millán and Torras (1991a).

2.2. *Connectionist approaches to the ARL problem*

The *associative reinforcement learning (ARL)* problem (Barto, et al., 1981; Sutton, 1984; Barto & Anandan, 1985) offers a simple and general framework for developing adaptive reactive systems. Simply stated, the ARL problem is that of learning to associate with each stimulus **X** the action **Y** that maximizes reinforcement z—either present, future or cumulative.

Supervised learning can be used to tackle the ARL problem through the *system identification approach*. As illustrated in Figure 1, this approach consists in training a connectionist

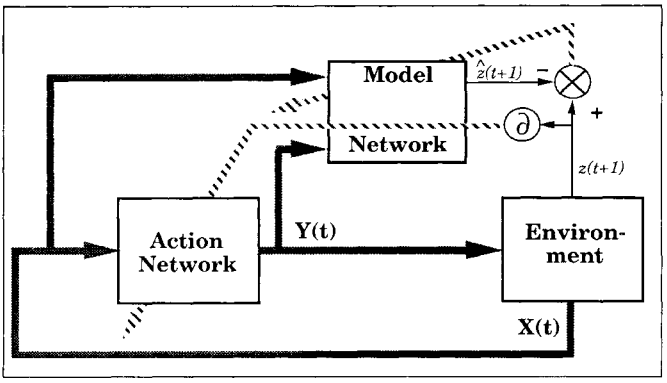


Figure 1. System identification approach to the ARL problem.

network—the *model network*—to identify the function that relates each agents' output with the subsequent reinforcement received from the environment. Once this function is encoded in the model network, the weights of this network are frozen and the supervised learning process shift to the *action network*—i.e., the agent. The gradient of the reinforcement signal with regard to the agent's output, $\partial z/\partial \mathbf{Y}$, required for training the action network is estimated by back-propagating derivatives through the model network. This approach has been suggested by Werbos (1987), but the only implementations are those of Munro (1987), Robinson (1989), and Jordan and Jacobs (1990). Although ingenious and analytically well-defined, this approach has the important limitation of assuming that the model has been learned with enough accuracy to allow to compute a good approximation of the reinforcement gradients.

A simpler approach to the ARL problem is *to approximate* $\partial z/\partial \mathbf{Y}$ *by its sign*. This idea has been proposed, and successfully applied, by Saerens and Socquet (1989) in the field of control. The signs of the partial derivatives are obtained from qualitative knowledge about the direction in which the action components must be modified to increase the reinforcement signal.

Finally, the most common approach to the ARL problem is *to estimate* $\partial z/\partial \mathbf{Y}$ *by measuring the correlation between variations in actions and variations in reinforcement*. This learning paradigm, known as *reinforcement learning*, relies on performing different actions in response to each stimulus, observing the resultant reinforcement, and incorporating the best action into the situation-action rules of the system. Several reinforcement systems have been developed for solving nontrivial problems (Barto, et al., 1983; Anderson, 1987; Chapman & Kaelbling, 1990; Lin, 1990; Mahadevan & Connell, 1990). Anderson (1987) illustrates the kind of situation-action rules learned by a particular reinforcement system.

The two main limitations of the basic reinforcement learning algorithm are a possibly long learning phase and the inability to capture complex features of the problem. To palliate these limitations, several extensions to the basic algorithm have been proposed, such as adding an action model for relaxation planning (Lin, 1990; Sutton, 1990) and combining several modules, each one specialized in solving a particular primitive task (Singh, 1991).

One contribution of this paper is in this direction, since we show that adding a stabilizing strategy to the learning algorithm and incorporating domain-specific knowledge in the codification scheme permits avoiding the above limitations.

Note also that the reinforcement learning approach is open to improvement through the incorporation of supervised learning methods. One classic example of this is the introduction of a *critic* element to predict the amount of reinforcement that would follow a given action (Barto, et al., 1983; Sutton, 1984). Anderson (1986) also used a supervised learning rule for updating the hidden layers of a reinforcement-based system.

3. Path finding as an ARL problem

Because the aim of this work is to carry out a feasibility study on a new approach to robot path finding, we assume a *point robot* and *circular obstacles*. The *evaluation criterion* used is a *compromise between minimizing path length and maximizing the distance to the obstables and walls*. Finally, the goal is considered to have been reached if the current configuration of the robot is below a certain constant distance from the goal, namely $goal_\epsilon$:

$$\text{dist}[conf_{start}, conf_{goal}] < goal_{\epsilon}. \quad (1)$$

To formulate the above instance of the path finding problem as an ARL problem, we need to specify what the input, output and reinforcement signal are.

3.1. Input information

The input to the path-finder consists of an *attraction force* exerted by the goal and several *repulsion forces* exerted by the obstacles. The robot has a limited perception range, giving higher consideration to closer obstacles.

Let the *shortest path vector (SPV)* be the vector that connects the current and the goal robot configurations. The line supporting the SPV and its perpendicular at the current configuration divide the workspace into four quadrants.

The intensity *ia* of the attraction force is an inverse exponential function of the distance x between the current and goal configurations:

$$ia(x) = e^{-k_{att}*x}, \quad (2)$$

where k_{att} is a constant. The direction of the attraction force is that of the SPV, but since the output of the path-finder is computed with respect to this direction (see the next subsection), it need not be included in the input to the system.

Each repulsion force represents the resistance of an obstacle to the fact that the robot follow the SPV. Each such force follows the bisector of the quadrant where the obstacle lies and heads towards the opposite quadrant. For example, in the upper situation in Figure 2, an obstacle is to the southeast of the current configuration and its repulsion force is northwestward. Because the directions of the repulsion forces are specified with respect to the direction of the attraction force, they are implicit in the codification and therefore are not included in the input to the system. Note also that this specification permits grouping

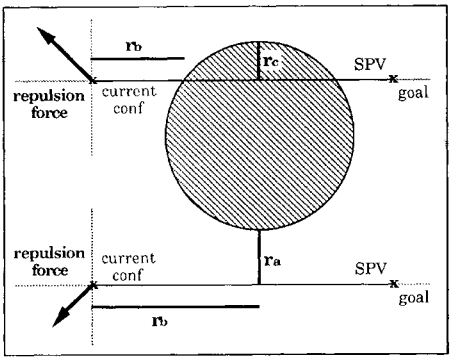


Figure 2. Factors of the repulsion force.

all the repulsion forces into four resulting signals, each being the magnitude of a vector starting at the current configuration and following the bisector of a quadrant.

The intensity of each single repulsion force depends on three factors: (i) the shortest distance between the obstacle and the SPV, r_a, (ii) the distance from the current configuration of the robot to the *point of conflict*—i.e., the point of the SPV nearest to the obstacle—r_b, and (iii) in the case that the SPV intersects the obstacle, the shortest translation of the obstacle perpendicular to the SPV that leads to nonintersection, r_c. If for a configuration and an obstacle several points of conflict exist, then the one nearest to the current robot configuration is taken. Figure 2 illustrates these factors for two different situations and the resulting repulsion forces.

The first factor is aimed at avoiding obstacles in the proximity of the SPV. The second allows to avoid obstacles near to the robot current configuration. The third ensures that, in the case that the SPV intersects an obstacle, the next robot movement is the more distant from the SPV, the deeper is the penetration into the obstacle.

In sum, *the number of input signals* to the path-finder *is independent of the number of obstacles in the environment*, and these signals are five: the intensity of the attraction force, *ia*, and the *intensity of the environmental repulsion from each quadrant*, r^1, r^2, r^3 and r^4, quadrants being numbered counterclockwise. The detailed way in which the repulsion signals are computed is described in Appendix A. Since the input signals are factors in the learning equations of the system (see Section 4.4), signals with larger magnitudes would have a greater influence on learning than would other signals. To remove this bias all input signals are scaled to lie within the same interval, and are thus codified as real numbers in [0, 1].

It is worth noting that one of the aims of adopting the above codification scheme has been to favor the generalization abilities of the path-finder. Since the goal is codified in the input information and the input is independent of the environment used during the learning phase, the knowledge acquired for a situation should be transferable to a different one.

3.2. Output information

The output of the path-finder represents the step taken by the robot and it is codified as a move in *relative cartesian coordinates* with regard to the SPV. That is, the positive x axis is the SPV. Both increments, Δy and Δx, are real values in the interval $[-1, 1]$.

The output signals determine a direction and a length. Nevertheless, the actual length of the move made by the robot is computed by the following expression:

$$actual\_length = \begin{cases} length * radius, & \text{if } length * radius > step_\epsilon, \\ step_\epsilon, & \text{otherwise,} \end{cases} \tag{3}$$

where $step_\epsilon$ is a constant and

$$radius = \min(perception_{range}, \text{dist}[conf_{start}, conf_{goal}]). \tag{4}$$

The motivation of this postprocessing of the output signals is twofold. Firstly, because the robot concentrates on its neighboring obstacles, the maximum distance the robot can cover in a single step should be limited by the perception range. Otherwise, the robot could collide with obstacles "not completely" perceived. Secondly, the path-finder is intended to be a reactive system, so it should react to each stimulus with some action. $step_\epsilon$ represents the minimum action.

Two important consequences of the manner in which the output signals are codified and postprocessed are that *the space of attainable configurations is continuous* and that *there is no predetermination whatsoever of the direction and length of each step*, with the only constraint imposed by the perception range. A final benefit is that the output postprocessing, in combination with the input codification that supports the generalization capabilities and the reactive nature of the path-finder, offers the robot the possibility of coping with dynamic environments. The step the robot takes is mainly aimed at avoiding neighboring obstacles and it never goes beyond the perception range, therefore the probability of colliding with a mobile obstacle is low.

3.3. *Reinforcement signal*

The reinforcement signal is a measure of how good is the answer of the system to a particular stimulus. It is calculated on the basis of the quality of the configuration reached by the robot—a combination of its attraction and repulsion factors—and the way in which this configuration has been reached—as measured by the step clearance. The quality of a configuration is an indication of the effort still remaining to reach the goal.

Each robot configuration has two values associated. The first is the *attraction factor*, a, that corresponds to the intensity of the attraction force. The second is the *repulsion factor*, r, that is a function of both the repulsion intensities of the obstacles in front of the robot and the *step clearance*—i.e., the shortest distance to the obstacles and walls of the step with which that configuration is reached:

$$r = \max(r^1, r^4, 1 - step\_clearance). \tag{5}$$

The reinforcement signal, z, is a real number in $[-1, 1]$. It equals 1 when the goal configuration is reached and it equals -1 when a collision with an obstacle or a wall happens:

$$z = \begin{cases} 1, & \text{if } a = 1, \\ -r, & \text{if } r > 1 - rep_\epsilon, \\ a - k_{ar} * r, & \text{otherwise,} \end{cases} \tag{6}$$

where rep_ϵ and k_{ar} are constants.

The present system differs from previous systems tackling the ARL problem in that the latter were intended for single-criterion tasks. In this work we demonstrate that by enriching the reinforcement signal with information that both is available from sensorial data and reflects all the desired criteria (proximity to the goal and clearance), then it is possible to solve problems with *strong performance demands* (reaching the goal quickly and safely).

Table 1. Actual values of the factors and thresholds used for computing the signals.

Name	Value
$goal_\epsilon$	0.05
k_{att}	0.1
k_{pa}	100.0
$perception_{range}$	2.0
k_{r_a}	0.2
k_{r_b}	0.1
k_{r_c}	5.0
k_{rep}	1.0
$step_\epsilon$	0.01
k_{ar}	0.75
rep_ϵ	0.1

Finally, the way in which the reinforcement signal is computed allows to reduce the complexity of the task to be solved. In the robot path finding problem, the consequences of an action can emerge later in time. Thus, actions must be selected based on both their short- and long-term consequences. Since the reinforcement signal is computed using *global information*—i.e., it is based not on the current robot configuration but on the SPV—the path-finder gets a measure of the short- and long-term consequences of an action only one time-step after executing it. Thus the task is reduced to learn, for each stimulus, to perform the action which maximizes the reinforcement signal.[2]

The determination of both the input and the reinforcement signals is reminiscent of the *potential field approach* to path finding (Khatib, 1986).

Table 1 summarizes all the constants—coefficients and thresholds—and their actual values used to compute the different signals.

4. A reinforcement connectionist path-finder

The path-finder is made of two elements, namely the *step generator* and the *critic*, and interacts with its environment as depicted in Figure 3. At each time t, the environment provides the path-finder with the input pattern $\mathbf{X}(t) = (x_1(t), x_2(t), x_3(t), x_4(t), x_5(t))$, together with the *environmental reinforcement signal* $z(t)$. The input pattern is fed to both the step generator and the critic. Nevertheless, the step generator does not receive directly the environmental reinforcement signal but the *heuristic reinforcement signal* $h(t)$ elaborated by the critic. The latter is an enrichment of the former based on past experience of the path-finder when interacting with the environment. The step generator produces instantaneously an output pattern $\mathbf{Y}(t) = (y_1(t), y_2(t))$ that it is the output of the path-finder. The environment receives this action $\mathbf{Y}(t)$ and, at time $t + 1$, sends to the path-finder both an evaluation $z(t + 1)$ of the appropriateness of the action $\mathbf{Y}(t)$ for the stimulus $\mathbf{X}(t)$ and a new stimulus $\mathbf{X}(t + 1)$.

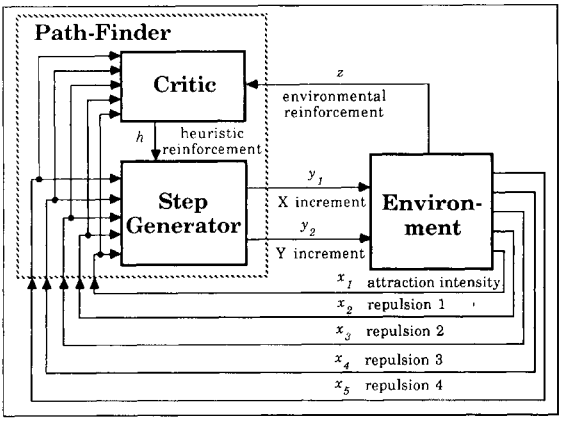


Figure 3. Connectionist path-finder according to the ARL architecture.

Of the two kinds of reinforcement learning algorithms proposed to solve the ARL problem, namely *associative reward-penalty*, A_{R-P}, (Barto & Anandan, 1985) and *associative search*, *AS*, (Barto, et al., 1981; Sutton, 1984; Williams, 1986, 1987), we adopt the second one because it fits better the specifications of the output and reinforcement signals we have. The A_{R-P} algorithm has been mainly designed for instances of the ARL problem where the action network—the step generator in our case—produces binary output signals and where the reinforcement has only two possible values, namely success or failure.

A central issue for any reinforcement system is to explore alternative actions for the same stimulus. *Stochastic units* provide this source of variation. Thus, the step generator—or, at least, its output layer—is built out of this kind of unit. The different architectures of the step generator—i.e., number of hidden layers, kinds of hidden units, and connectivity—tested during the simulations will be described in the next section. Nevertheless, the stochastic behavior of the path-finder should tend to be *deterministic* with learning. Otherwise, the path-finder could not generate *stable solution paths* after it eventually discovers them.

4.1. The basic AS algorithm

Continuous stochastic units compute their output in three steps (Gullapalli, 1988), as depicted in Figure 4 and expressed in Equations (7) through (10).

Since the signals we are interested in are continuous, a separate control of the location being sought (*mean*) and the breadth of the search around that location (*variance*) is needed. The first step is to determine the value of these parameters. The mean μ should be an estimation of the optimal output. A simple way is to let $\mu_i(t)$ equal a weighted sum of the inputs $s_j(t)$ to the unit i plus a threshold $\theta_i(t)$:

$$\mu_i(t) = \sum_{j=1}^{n} (w_{ij}(t)s_j(t)) + \theta_i(t). \tag{7}$$

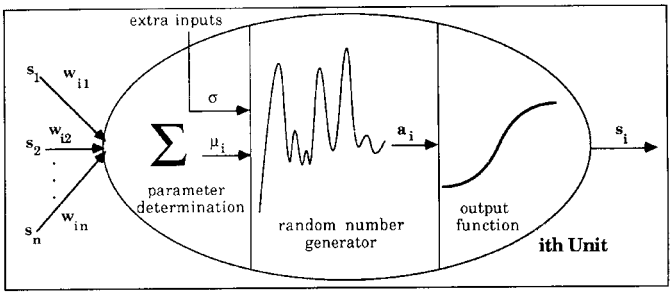


Figure 4. Structure and functionality of a stochastic unit.

The standard deviation σ should be small if the expected output of the step generator is close to the optimal, and it should be high in the opposite case. Since the heuristic reinforcement signal provides this comparative measure of output goodness, $\sigma(t)$ should depend on the *expected heuristic reinforcement*, $\hat{h}(t)$:

$$\sigma(t) = g(\hat{h}(t)), \tag{8}$$

where the function g will be made explicit in Section 4.3, once the different ways of computing the heuristic reinforcement signal will be presented.

As a second step, the unit calculates its *activation level* $a_i(t)$ which is a normally distributed random variable:

$$a_i(t) = N(\mu_i(t), \sigma(t)). \tag{9}$$

Finally, the unit computes its output $s_i(t)$:

$$s_i(t) = f(a_i(t)) = \frac{2}{1 + e^{-\beta a_i(t)}} - 1, \tag{10}$$

where β is a constant in [0, 1].

In the AS family of algorithms, the weights are modified according to the following general expression:

$$\Delta w_{ij}(t) = \alpha h(t) e_{ij}(t - 1), \tag{11}$$

where α is the *learning rate* and e_{ij} is the *eligibility factor* of w_{ij}. The eligibility factor of a given weight measures how influential that weight was in choosing the action. We have explored twenty-five versions of this general rule, each particular version differing from the others in the way h and e_{ij} are calculated. These versions result from combining the five heuristic reinforcement signals with the five eligibility factors which shall be presented in Sections 4.3 and 4.4.

4.2. A strategy for discovering stable (quasi) optimal paths

It has been stated above that in order to obtain stable solution paths, stochastic units should become deterministic as learning proceeds. The way in which σ is computed guarantees that the breadth of search will diminish asymptotically to zero as the path-finder learns. But this is not acceptable to solve the problem efficiently.

We have extended the basic AS algorithm with a mechanism for accelerating the generation of stable solution paths (Millán & Torras, 1990). When the path-finder, after a certain experience with the environment, discovers an *acceptable* path, the search for a solution path proceeds in a more "controlled" manner by *transforming the stochastic units of the step generator into deterministic units*. In addition, the weights are not updated after each step, but after the path has been generated and only if it is not a *(quasi) optimal* path, *qo-path*. The acceptability and optimality criteria are defined by the boolean expressions:

$$acc\_path = (length < k_{acc_{length}} * \text{dist}[conf_{start}, conf_{goal}]) \wedge (minimum\_clearance < k_{acc_{clear}}), \quad (12)$$

$$qo\_path = (length < k_{qo_{length}} * \text{dist}[conf_{start}, conf_{goal}]) \wedge (minimum\_clearance < k_{qo_{clear}}), \quad (13)$$

where $k_{acc_{length}}$, $k_{acc_{clear}}$, $k_{qo_{length}}$ and $k_{qo_{clear}}$ are workspace-dependent constants.

A deterministic unit is like a stochastic one but without the random component: the activation level is a weighted sum of the input. The weights arriving at a deterministic unit are modified using the same learning rule as for the weights arriving at stochastic units.

The weight updates are delayed and not applied when a qo-path is generated because, otherwise, the changes to the weights could probably prevent the path-finder from reproducing it. Since the output signals are continuous and the optimality criterion is very severe, even little weight modifications could alter the step generator so as to produce actions sufficiently away from the quasi-optimal ones.

The process for discovering stable qo-paths requires a further refinement. The fact of obtaining an acceptable path does not necessarily imply that a qo-path is nearby. The acceptable path discovered could be around a local optimum. Thus, if after consuming a fixed quantity of computational resources the path-finder does not discover a qo-path, then the deterministic units are turned back to being stochastic again and another acceptable path is looked for. Specifically, the deterministic phase is finished when either a single path reaches a certain number of steps or a given number of paths are generated without finding a qo-path.

The strategy above is sufficient for finding a stable qo-path. Nevertheless, we have found that sometimes the trajectory followed by the weight configuration of the step generator reaches bad states. The algorithm eventually allows to escape from them, but at the cost of spending a lot of time. This time could be significantly reduced if these states were detected as quickly as possible and a new state (at least as good as the initial one) was readily imposed. An easy way of implementing this idea is not to allow the path-finder

```
Algorithm discover_stable_qo-path
begin
    repeat
        counter := 0
        initializations
        repeat
            repeat
                generate_path (immediate_learning, stochastic_units)
            until (acceptable_path and enough_experience)
            repeat
                generate_path (delayed_learning, deterministic_units)
            until (qo-path or no_more_resources)
            counter := counter + 1
        until (qo-path or limit_of_iterations)
    until qo-path
end
```

Figure 5. Strategy for discovering a stable qo-solution.

to alternate the deterministic and stochastic phases more than a fixed number of times. If this limit is reached, the step generator and the critic are reinitialized.

Figure 5 shows the final algorithm for discovering a stable qo-path.

4.3. The critic

As stated above, the goal of the critic is to transform the environmental reinforcement signal into a more informative signal, namely the heuristic reinforcement signal. This improvement is based on past experience of the path-finder when interacting with the environment, as represented by the *reinforced baseline b*:

$$h(t) = z(t) - b(t - 1). \tag{14}$$

The critic receives the input pattern $\mathbf{X}(t)$ and predicts the reinforcement baseline $b(t)$ with which to compare the associated environment reinforcement signal $z(t + 1)$.

A first alternative for the reinforcement baseline is to make it a *constant*—zero in the simulations presented here. This is equivalent to shutting the critic off. One has to expect a long learning phase before obtaining a qo-path, since the robot may receive a positive environmental reinforcement when moving away from the goal, in cases where it can approach it.

In order to avoid this shortcoming, the environmental reinforcement received at the current step should be compared with that received at previous steps. Two possibilities for making this comparison arise: *short-term*, and *long-term*. In the first case, the reinforcement baseline is the environmental reinforcement received at the preceding time step. In the second case, it is a trace of all the environmental reinforcement received by the path-finder. There are, however, reasons to doubt that these techniques would work well on the task at hand. The main of them is that the best move the robot can make in a certain situation may correspond an environmental reinforcement lower than the preceding one.

A third comparison technique, *predicted comparison* (Sutton, 1984), tries to overcome this limitation by computing the reinforcement baseline as a prediction based on the environmental reinforcement received when the same—or similar—input patterns occurred. That is, the critic has to predict the environmental reinforcement signal $z(t + 1)$ to be received by the path-finder when the stimulus $\mathbf{X}(t)$ is present. In order to undertake this task, the critic is built as a second network out of deterministic units, and since it is provided with input/output pairs, a supervised learning algorithm is used to learn to associate each stimulus with the corresponding environmental reinforcement signal. In particular, we have used the "on-line" version of the backpropagation algorithm with a *momentum* term:

$$\Delta v_{ij}(t) = -\epsilon \frac{\partial(z(t) - \hat{z}(t - 1))^2/2}{\partial v_{ij}} + \eta \Delta v_{ij}(t - 1), \tag{15}$$

where $\mathbf{V}$ is the weight matrix of the critic, ϵ is the learning rate, η is the momentum factor, and $\hat{z}$ is the output of the critic—i.e., the expected environmental reinforcement signal.

The critic has to be updated after each path-finder/environment interaction because as the step generator improves, the mapping from stimuli to reinforcement changes.

We have found that using a large learning rate ϵ for the critic and a small learning rate α for the step generator accelerates the learning process. The rationale is the following. By allowing the critic to adapt more rapidly than the step generator, the critic has the opportunity to predict acceptably the next environmental reinforcement signal associated to the current stimulus as the step generator is, during a certain period of time, almost stable. Then, the step generator can take full advantage of the reinforcement baseline.

A potential benefit of the momentum term when applied to *dynamic tasks*—i.e., tasks where the training information changes over time—like that faced by the critic is that it prevents $\mathbf{V}$ from oscillating dramatically. Since, at the beginning of the learning phase, the step generator explores a wide range of alternative actions, to each stimulus perceived by the path-finder will correspond a large variety of environmental reinforcement signals. Consequently, the error between the actual and the predicted environmental reinforcement is not only due to a misfunction of the critic, but also to the "chaotic" mapping the critic is dealing with. So, there is the risk that the weight modifications made at a certain time change $\mathbf{V}$ too much. The momentum tackles this problem by reducing the intensity of opposite weight modifications at consecutive times.

A second advantage of using a predicted-comparison reinforcement baseline is to mitigate the possibility of *overlearning*. Since the goal of the step generator is to produce the optimal action in the presence of every stimulus, learning should stop when the step generator has discovered the suitable situation-action rules. A predicted-comparison mechanism accomplishes this effect because the expected and the actual environmental reinforcement signals tend to be the actual and the optimal ones, respectively, as the critic and the step generator improve.

Predicted comparison, however, may not cope adequately with collisions—a frequent situation during the learning phase. The reason is that when a collision happens, one would like to punish severely the step generator. This is satisfied if no baseline is used, whereas predicted comparison fails when the prediction is close to the environmental reinforcement

signal. Consequently, the performance of the path-finder should improve if the reinforcement baseline were computed as a *heuristic predicted comparison*:

$$\tilde{z}(t) = \begin{cases} 0, & \text{if } z(t+1) = -1, \\ \hat{z}(t), & \text{otherwise} \end{cases} \tag{16}$$

The following expression summarizes the five alternative ways used to calculate the reinforcement baseline:

$$b(t) = \begin{cases} 0, \\ z(t), \\ bt(t), \\ \hat{z}(t), \\ \tilde{z}(t), \end{cases} \tag{17}$$

where

$$bt(t) = \lambda z(t) + [1 - \lambda] bt(t-1), \tag{18}$$

with λ being a constant in [0, 1].

Let us now specify the function g used to calculate the standard deviation σ of the stochastic units in the step generator. Remember that σ should be small if the expected output of the step generator is close to optimal, and it should be high in the opposite case. The last four ways of computing the reinforcement baseline lead to a heuristic reinforcement signal whose absolute value is close to zero if the environmental reinforcement is close to the expected one, and it is high in the opposite case. Consequently, $\sigma(t)$ should be proportional to the expected heuristic reinforcement $\hat{h}(t)$:

$$\sigma(t) = k_\sigma * \hat{h}(t), \tag{19}$$

where k_σ is a constant and $\hat{h}(t)$ is a trace of the absolute value of past heuristic reinforcement received:

$$\hat{h}(t) = \xi * \text{abs}(h(t)) + [1 - \xi]\hat{h}(t-1), \tag{20}$$

ξ being a constant in [0, 1]. Gullapalli proposes the following expression to compute σ:

$$\sigma(t) = k_\sigma * [1 - \hat{z}(t)], \tag{19'}$$

where $\hat{z}(t)$ is the expected environmental reinforcement. Both expressions are equivalent only in the case that the highest environmental reinforcement is recieved for every pair (stimulus, optimal action). This condition does not hold in the path finding problem where the highest environmental reinforcement is only received when the goal is reached.

In the case that no reinforcement baseline is used, the heuristic reinforcement is the environmental one. This means that (19) provides a standard deviation that is not the desired one when the expected output is good, as σ will be high when it should be small. For this particular case, (19′) will be used.

4.4. The eligibility factor

The eligibility factor of a weight measures the contribution of that weight to the action taken. We have used two kinds of rules to compute this contribution. The first measures the correlation between the outputs of the units linked by the connection under consideration. Two ways of implementing this *intensity-based* mechanism are the *Hebbian rule* and a trace of it.

The second kind of rule consists of a *discrepancy-based* mechanism which evaluates the difference between the actual output and the expected one:

$$e_{ij}(t) = s_j(t)[s_i(t) - f(\mu_i(t))], \tag{21}$$

where $f(\cdot)$ is the same function as in (10). A slightly different implementation proposed by Gullapalli (1988) is the following:

$$e_{ij}(t) = s_j(t) \frac{a_i(t) - \mu_i(t)}{\sigma(t)}. \tag{22}$$

Finally, a third discrepancy-based rule makes the learning algorithm perform *gradient ascent* on the expected environmental reinforcement, as proved by Williams (1986, 1987):

$$e_{ij}(t) = \frac{\partial ln N}{\partial w_{ij}}(t) = s_j(t) \frac{a_i(t) - \mu_i(t)}{\sigma^2(t)}, \tag{23}$$

where N is the normal distribution function in (9).

In the three cases, the greater the discrepancy between the actual and the expected outputs, the greater the intensity of the weight modification. The sign of this modification is such that if the heuristic reinforcement signal is positive, then the next time the current stimulus—or similar ones—will be presented, the expected output of the stochastic unit will be closer to the current one. Conversely, if the heuristic reinforcement signal is negative, then the weights are modified so as to reduce the probability of generating the current output.

Sutton (1984) proved experimentally that such a mechanism allows a binary stochastic unit to discriminate between similar input vectors, and Barto (1985) argued that it helps to cope with situations where two or more of the actions the path-finder can take in the presence of a given stimulus have associated environmental reinforcement signals of similar intensity.

The following expression summarizes the five alternative ways in which the eligibility factor has been calculated:

$$e_{ij}(t) = \begin{cases} s_i(t)s_j(t), \\ et_{ij}(t), \\ s_j(t)[s_i(t) - f(\mu_i(t))], \\ s_j(t)\dfrac{a_i(t) - \mu_i(t)}{\sigma(t)}, \\ s_j(t)\dfrac{a_i(t) - \mu_i(t)}{\sigma^2(t)}, \end{cases} \tag{24}$$

where

$$et_{ij}(t) = \gamma s_j(t)s_i(t) + [1 - \gamma]et_{ij}(t - 1), \tag{25}$$

γ being a constant in [0, 1].

Note that only the first two rules—i.e., those corresponding to the intensity-based mechanism—are applicable when deterministic units are in use. The other three eligibility factors depend on the stochastic behavior of the units.

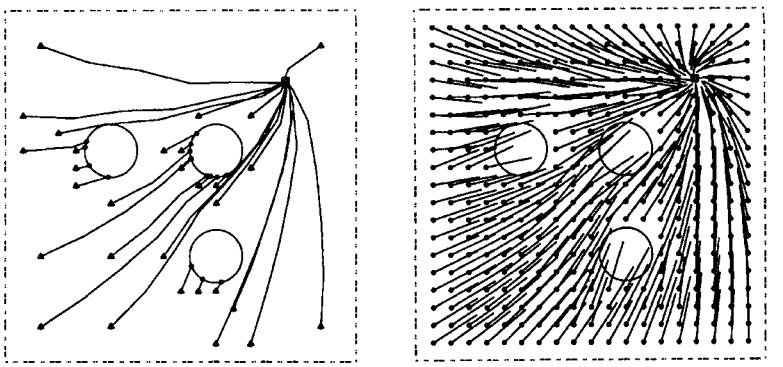
5. Experimental results

The system has been implemented in Common Lisp on a VAX station 2000. The feasibility study has been carried out in two main steps. The first step is an experimental comparison of the twenty five versions of the general AS algorithm described in the preceding section. The objective of this comparative study is to identify the version that best suits the robot path finding problem. Then, the most promising version is used to try to build a path-finder with powerful generalization abilities so as to produce, after a limited learning process, qo-paths in any non-maze-like workspace and to cope with dynamic environments.

The workspace depicted in Figure 6 has been employed in the simulations. There exist three obstacles—shown as circles—and the goal configuration is represented by a square. The workspace is a room of 10 * 10 units. The centers of the obstacles are located at (3.0, 6.0), (6.0, 6.0) and (6.0, 3.0), and their radii are equal to 0.75 units. The goal configuration is located at (8.0, 8.0).

In the simulations, a path is considered to end either at the goal or with a collision. A path could be alternatively defined to finish only when the goal is reached. This second definition, however, presents the following shortcoming. During the learning phase, the path-finder needs to be *sufficiently stimulated*—i.e., the training data should be both sufficiently varied to reveal the reinforcement function and sufficiently repetitive to make learning possible—to discover the set of suitable situation-action rules. But, if the robot is allowed to leave the collision situations by itself, then the proportion of inputs corresponding to a collision with respect to all the other kinds of inputs is very high. So, the training data is biased to a particular kind of stimuli, which makes learning harder.

A desirable feature for any system is its robustness to changes in the values of the parameters governing it. Thus, no attempt has been made to search for the best set of parameter



(a) (b)

Figure 6. Behavior of the first path-finder.

values. Instead, we have chosen values that intuitively seem adequate for all the versions of the general AS rule. The values of the constants used in the learning rules appear in Table 2.

In the implementation of the strategy described in Section 4.2, the deterministic phase is finished when either a single path reaches 500 steps or 500 paths are generated without finding a qo-path.

Table 2. Actual values of the factors used in the learning rules.

Name	Value
β	0.4
α	0.125
ϵ	0.5
η	0.5
λ	0.2
γ	0.2
k_a	2.0
ξ	0.75
$k_{acc_{length}}$	1.5
$k_{acc_{clear}}$	0.15
$k_{qo_{length}}$	1.15
$k_{qo_{clear}}$	0.2

5.1. Tuning the learning rule: A comparative study

The step generator used to carry out the comparative study has only two layers of units, input and output. The critic has a hidden layer of four units.

The following computer experiment has been designed. The initial configuration is located at (3.0, 3.0), in such a manner that the obstacles lie in between it and the goal. Thus the path-finder is required to produce a nontrivial qo-path. A run finishes when a qo-path is discovered.

For each version, twenty qo-paths have been generated. Before beginning the generation of each qo-path, all the weights of the step generator are set to zero and the weights of the critic are initialized randomly.

Given the difficulty of generating acceptable paths, the experience acquired by the path-finder for producing the first acceptable path is considered sufficient to finish the stochastic phase of the algorithm.

Versions using a discrepancy-based rule for computing the eligibility factor turn to using the Hebbian rule when units become deterministic (see the next subsection for a justification of this choice).

Table 3 gives the average number of steps required to generate a stable qo-path for every version. The types of reinforcement baseline and eligibility factor are numbered according to their positions in (17) and (24), respectively. The results of these experiments show that every version, if given enough time, produces a stable qo-path from the initial to the goal configurations. Nevertheless, important differences exist in the performance of the various versions.

5.1.1. Eligibility factor

Let us firstly concentrate on the role of the eligibility factor in the performance of the path-finder. Three main conclusions can be extracted from the data in Table 3.

First, discrepancy-based rules (three last columns in the table) clearly outperform intensity-based ones.

The second conclusion is that the performance of the versions adopting the Hebbian rule—first column—is always better than the performance of the versions using a trace of this rule—second column. This is the reason for using the Hebbian rule in the deterministic phase, as mentioned in the preceding subsection.

Table 3. Average number of steps for finding a stable qo-path.

		Eligibility				
		1	2	3	4	5
Baseline	1	34939	46122	15559	14527	10457
	2	59599	66791	48753	24810	32332
	3	31870	52202	28379	17494	27645
	4	25997	27999	12675	13839	08473
	5	21362	22840	16301	11789	17907

Finally, of the three discrepancy-based rules, none outperforms the others for all choices of reinforcement baseline. However, the fifth eligibility factor leads to the two best versions of the algorithm.

5.1.2. Reinforcement baseline

The influence of the reinforcement baseline on the performance of the path-finder is not so clear and does not agree so exactly with the theoretical expectations as the influence of the eligibility factor.

As expected, short-term and long-term comparisons—second and third rows of the table, respectively—are not suitable for solving our formulation of the path finding problem. In addition, the long-term versions work better than the short-term ones.

Surprisingly, the performance of the versions adopting a null reinforcement baseline—first row—is very similar to the performance of the versions using predicted comparison as reinforcement baseline—fourth row—when combined with discrepancy-based eligibility factors. Nevertheless, the former never outperforms the latter. An explanation for this fact could be that a null reinforcement baseline deals better with collisions than predicted comparison, as we hypothesized in Section 4.3. However, the performance of the versions using heuristic predicted comparison—fifth row—which combines a null reinforcement baseline and predicted comparison, does not support this hypothesis, since it is considerably worse than the performance of plain predicted comparison when two of the three best eligibility factors are used—third and fifth columns.

So, it seems that predicted comparison is the best kind of reinforcement baseline.

5.1.3. Conclusions

Since the performance of the versions combining the first, fourth and fifth types of reinforcement baseline—i.e., null, predicted comparison and heuristic predicted comparison, respectively—with the third, fourth and fifth types of eligibility factors—i.e., the three discrepancy-based rules—are all quite similar, we will look at some other information to better discriminate which of these nine versions is the best.

One of these criteria is the average number of times that the path-finder has been reinitialized until a stable qo-path is found. This information is implicit in Table 3, as all the steps taken by the robot for finding a stable qo-path are recorded. Nevertheless, this information provides a picture of the sensitivity of each version to the initial conditions. Clearly, the less a version depends on its initial conditions, the more it should be preferred. Table 4 shows the average number of reinitializations of the nine most promising versions identified before.

Table 4 confirms the ranking we have outlined in the preceding subsections. The reinforcement baseline and the eligibility factor most suitable for our formulation of the robot path finding problem are predicted comparison and the discrepancy-based rule proposed by Williams (1986), respectively. This version, however, is not far better than some other versions.

Appendix B provides an analysis of variance for the most promising versions that supports these conclusions.

Table 4. Average number of reinitializations of the most promising versions.

		Eligibility		
		3	4	5
Baseline	1	1.5	1.4	1.1
	4	1.4	1.4	0.9
	5	1.6	1.3	1.5

5.2. Configuring the path-finder

The second phase of the feasibility study is aimed at building a path-finder with powerful *generalization abilities* which, after a limited learning process, produces qo-paths in any non-maze-like workspace and is able to cope with dynamic environments.

The workspace and the goal used in the experiments are the same as before. The training set consists of a certain number of initial configurations that are representative of the whole workspace. Because of the symmetry of this workspace, the training set consists of pairs of symmetrical configurations.

Training is carried out in an incremental way, using an extension of the algorithm shown in Figure 5. That is, the path-finder has to learn to generate a qo-path from the first starting configuration in the training set. Then, it tries to generate a qo-path from the second starting configuration. As the necessary knowledge required for solving this new situation is discovered and codified, part of the previous knowledge could be lost. So, the path-finder must learn again to generate a qo-path from the first starting configuration, but this time using directly deterministic units. The iteration "learn to generate a qo-path from the new starting configuration—recover the ability to generate qo-paths from the previous starting configurations" is repeated until the path-finder is able to generate qo-paths from all the starting configurations.

The fact that the path-finder deals with the starting configurations one at a time is a constraint imposed by the stability strategy.

To verify if it is still able to generate qo-paths from a previous starting configuration, the path-finder is made to produce a path, without modifying the critic and step generator networks, and the optimality criterion is applied. The steps in this path are not counted in the performance records.

All the system parameters but two have the values appearing in Table 2. The two parameters modified are k_σ and $k_{qo_{length}}$. In the simulations reported in this section, $k_\sigma = 1.0$ and the following distinction has been introduced with regard to $k_{qo_{length}}$. The path-finder has to deal now with two kinds of initial configurations, namely those such that there is no obstacle in between them and the goal and those such that there are. For the first group of initial configurations, the path-finder should generate qo-paths as straight as possible. So, a stronger optimality criterion has been defined by choosing a smaller $k_{qo_{length}}$. This value is 1.05. For the second kind of initial configurations, $k_{qo_{length}}$ continues to be 1.15.

5.2.1. *Problem non-linearity*

The first path-finder we have experimented with is the very simple one used to carry out the comparative study. That is, the step generator does not have any hidden layer of units and the critic has four hidden units arranged in a layer.

Unfortunately, the path-finder is not able to tackle satisfactorily the problem at hand. The reason is that, for our formulation of the robot path finding problem, *nonlinear associations* between the input and output signals are required. An illustration of one of these nonlinear associations follows.

Let us assume that there is no obstacle behind the current configuration of the robot, that is, $r^2 = r^3 = 0$. Let us consider now how the output of the Δx unit should be for different obstacle situations. Since the current and goal configurations are fixed, the intensity of the attraction force is constant. So, we will only consider the values of Δx for different instances of the pair (r^1, r^4). Figure 7 depicts the sign of Δx when the values r^1 and r^4 lie in three different ranges, namely *low*, *medium* and *high*.

The robot tries normally to approach the goal—i.e., Δx is positive. The only exception is when there are obstacles at the two sides of the SPV and they are close to the current configuration. The figure shows that the two regions of the space, one where Δx is positive and the other where Δx is negative, are not linearly separable.

Even though the simple path-finder used in the comparative study does not solve completely the instance of the path finding problem we are interested in, it illustrates the potentiality of a reinforcement learning connectionist approach. Figure 6 shows the behavior of the path-finder after knowing how to produce qo-paths from a particular set of starting configurations, namely those such that there is no obstacle in between them and the goal. Panel A depicts the paths generated by the path-finder from every starting configuration in the training set. Each initial configuration is represented by a triangle. In panel B, instances of the situation-action rules discovered by the path-finder are illustrated; for every starting

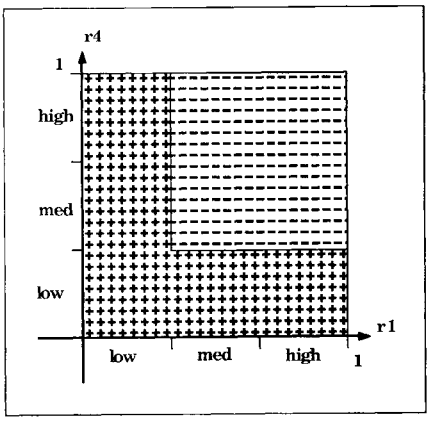


Figure 7. Nonlinearity of the mapping $(r^1, r^4) \rightarrow \Delta x$.

configuration considered—little circles—the move to be taken by the robot is shown. From the figure it is evident that the path-finder is able to tackle many more situations than those perceived during the learning phase. But, the most relevant feature is that *this simple path-finder knows the right direction, left or right, for the moves to avoid obstacles*. The cost of generating this particular path-finder has been 515 steps.

5.2.2. System performance

Given the abilities of this simple path-finder, we would like to add a hidden layer to the step generator in such a manner that these abilities are preserved when the missing nonlinear constraints are incorporated and the situation-action rules already discovered are tuned. To this end, the second path-finder we have tried has the same critic above, and its step generator is a network where all the units in a layer are connected to all the units in all the layers above. The weights associated to the direct links between the input and output layers are those of the first step generator and are not modified during the second learning phase. The hidden layer of the step generator is made of four units. The weights of the critic are initialized to the weights resulting from the first phase.

During this second learning phase, the new path-finder is made to deal with those starting configurations that have obstacles in between them and the goal, following the same incremental strategy as above with regard to previously learned situations. Besides, the path-finder is considered to have the necessary experience with the environment when it has generated three acceptable paths.

The purpose of the hidden layer is to encode the missing nonlinear features of the robot path finding problem. This layer can be built according to an *extension-and-homogeneity (EH)* principle or to a *modular-and-functionality (MF)* principle. If the first principle is adopted, the resulting step generator is totally made up of stochastic/deterministic units following the same version of the AS learning rule.

Anderson (1986) has found that this approach to the ARL problem leads to an inefficient learning process. He overcomes this limitation by adopting the second principle above. That is, the step generator is broken down into two modules—input-hidden layers and hidden-output layers—and the most suitable learning algorithm for each module's functionality is applied. In particular, Anderson developed an *approximate gradient method* for discovering the internal representations required. This method is an adaptation of the backpropagation algorithm where the error back-propagated from the i^{th} output unit to the j^{th} hidden unit at time t is $h(t)e_{ij}(t-1)/s_j(t-1)$.

Formally, if Anderson's version of the MF principle is adopted, the step generator is made of two modules. The first module consists of the output units and the connections arriving at them. The output units are stochastic/deterministic and the AS learning rule is used to update the connections. The second module consists of the hidden units and the connections arriving at them. The hidden units are deterministic and the weights associated to those connections are updated according to the expression:

$$\Delta w_{jk}(t) = \alpha_h \delta_j(t) s_k(t-1), \tag{26}$$

where α_h is the learning rate and

$$\delta_j(t) = \sum_{i=1}^{2} \left(\frac{h(t)e_{ij}(t-1)}{s_j(t-1)} w_{ij}(t-1) \right) s_j(t-1)[1 - s_j(t-1)], \quad (27)$$

where k ranges over the input units, j over the hidden units, and i over the output units.

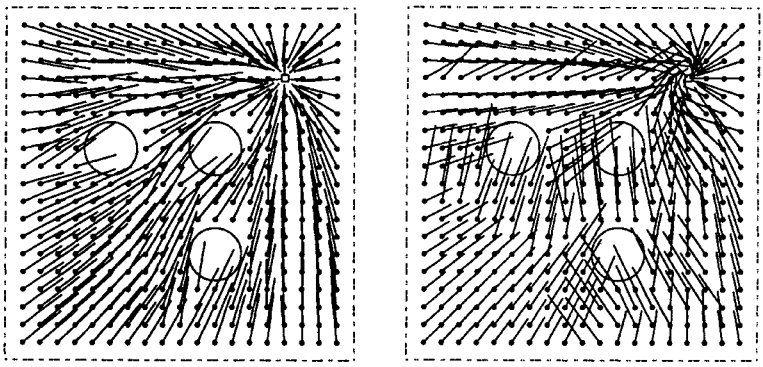
We have compared these two principles on the generation of qo-paths for a pair of symmetrical initial configurations that have obstacles in between them and the goal. In the simulations, $\alpha_h = 0.125$, therefore the learning rate is the same for all the units of the step generator. For each principle, five qo-paths have been generated. The average number of steps taken for networks adopting each principle appear in Table 5.

The computer experiments confirm that the performance of the MF principle is better than that of the EH principle. Nevertheless, the difference is negligible, perhaps because the task is not sufficiently difficult and, above all, because the system does not begin from scratch but with the "approximate" desired knowledge.

The most important result of this comparative study is illustrated in Figure 8. Panel A and panel B depict the behavior of two of the path-finders generated, one using the EH principle and the other the MF principle. Similar behaviors have been observed for other path-finders using these principles. The unexpected feature is that *the EH principle is able to modify correctly all the situation-action rules already discovered*, while *the MF principle is not.*

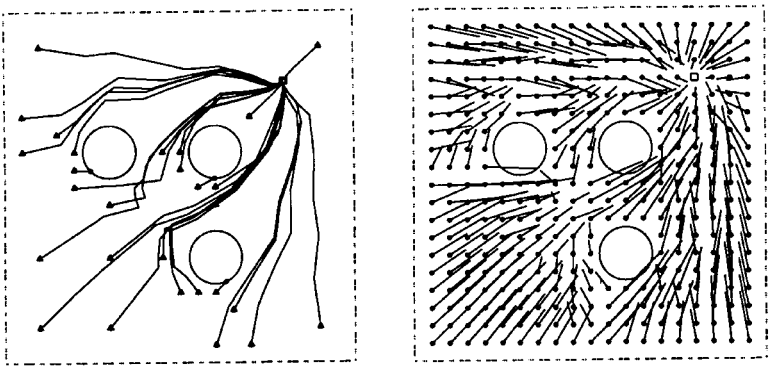
Table 5. Average number of steps required by each principle.

Principle	Steps
EH	3487
MF	3307



(a) (b)

Figure 8. Behavior of two path-finders adopting each principle.



(a) (b)

Figure 9. Behavior of the second path-finder.

Figure 9 shows the behavior of a pure reinforcement path-finder (EH principle) that is able to produce qo-paths from almost all the initial configurations in the training set. The number of steps required to reach this state of the path-finder has been 77315. The only situations not properly handled by the path-finder are a subset of the most difficult ones, that is those in which an obstacle lies in between the goal and the current robot configuration and is very close to the latter. These situations may be handled by getting appropriate guidance from a symbolic path-planner (Millán & Torras, 1991b).

It is worth noting that no two-layered step generator trained in one phase succeeded.

5.2.3. Noise-tolerance, generalization and dynamic capabilities

Until now, we have assumed that the robot can perceive the workspace perfectly. Nevertheless, a robot navigating in a real environment is subject to noisy and inaccurate measurements. Figure 10 depicts the behavior of the path-finder when sensory data are disrupted with 20% of white noise. In panel A, the noise is added to each input signal of the path-finder—i.e., r^1, r^2, r^3, r^4 and *ia*. In panel B, it is added to the basic sensory input of the robot—i.e., r_a, r_b, r_c and *ia*. Comparing the two panels of Figure 10 with Figure 9, panel B, it is evident that the path-finder exhibits a *large noise tolerance*, since the "maps" are very similar. The behavior of the path-finder is slightly better when the noise is added to the basic sensory inputs, because its effects are reduced by the preprocessing applied to obtain the input signals.

Figure 9, panel B, illustrates some of the generalization abilities of the path-finder. It can face many more situations than those perceived during the learning phase. In addition, the path-finder is also able to navigate in workspaces different from the one used during learning. Figures 11 through 13 show the behavior of the path-finder when only the goal is changed, more obstacles are added to the original workspace, and both the goal and

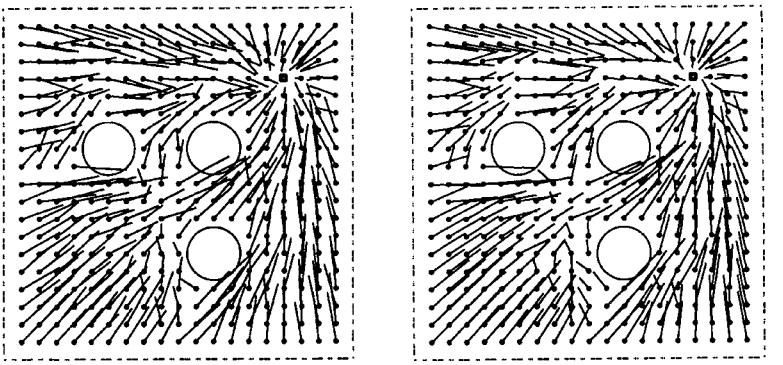


Figure 10. Noise tolerance exhibited by the path-finder.

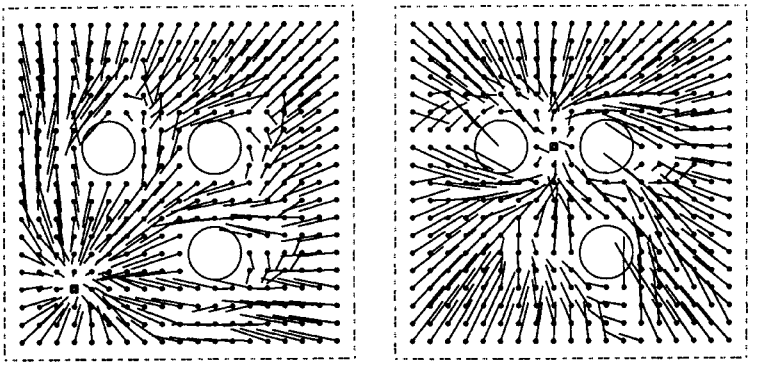


Figure 11. Generalization abilities: New goals.

the number of obstacles have changed. The results of these three experiments show that the *situation-action rules learned are both goal-independent and workspace-independent* and that, even in the worst cases, only a "light" additional learning phase suffices to readapt the path-finder to the new workspace.

Finally, Figure 14 shows how the *path-finder copes with dynamic environments*. If the robot has taken one or more steps toward the goal and either the obstacles (panel A) or the goal (panel B) move, the path-finder is still able to generate feasible paths. In the first case, the obstacles are moving toward the northwest, therefore approaching the goal, and

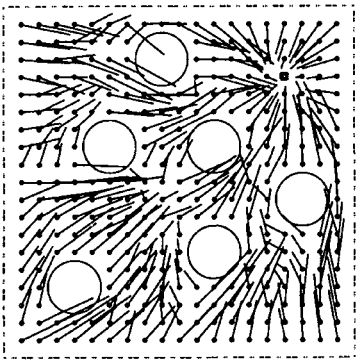


Figure 12. Generalization abilities: More obstacles.

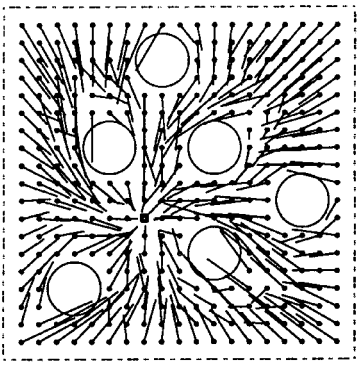


Figure 13. Generalization abilities: New goal and more obstacles.

the path-finder avoids them. In the second case, the goal is moving toward the northeast and the path-finder tracks it. This ability could be enhanced if a module to predict the motion of the goal and the obstacles were incorporated into the system.

6. Conclusions and future work

The simulations carried out in this paper demonstrate the adequacy of a reinforcement connectionist learning approach to implement local obstacle-avoidance capabilities. The formulation of the problem used to test this approach is a difficult one, since the input and output are continuous, the environment is partially unknown, and the optimality criterion (finding short paths with wide clearances) is severe. The problem, however, has been simplified by assuming a point robot and circular obstacles. These simplifications are dropped in the

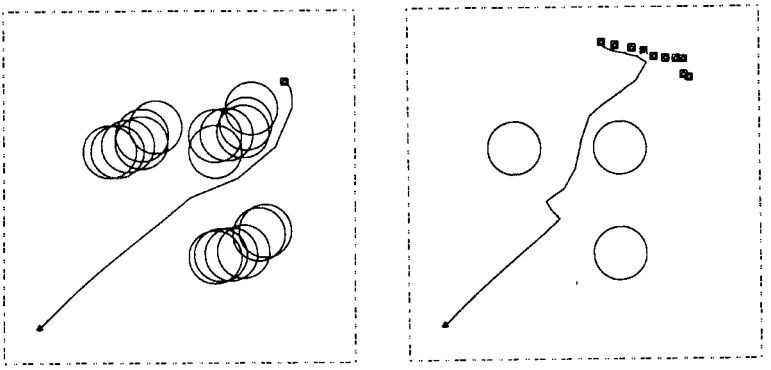


Figure 14. (a) Dynamic obstacles. (b) Dynamic goal.

extended prototype we are currently working on, which deals with a 2D mobile robot and polygonal obstacles.

The codification scheme adopted and the algorithm used to discover qo-paths can be thought of as domain-specific heuristics (Langley, 1985) for the robot path finding problem, which greatly improve the basic reinforcement-based weak search method. Equipped with these two modules, the path-finder not only learns the necessary situation-action rules in a very reduced time, but also exhibits good noise-tolerance and generalization capabilities, and is able to cope with dynamic environments.

In the Introduction we claimed that the robot path finding problem could be solved efficiently if symbolic planning were interfaced to subsymbolic obstacle avoidance. Our current research is oriented towards designing a *hybrid path-finder* by coupling a geometric global planning approach such as that in Ilari and Torras (1990) with the connectionist local obstacle avoidance approach described in this paper.

In Millán and Torras (1991b) we illustrate with a very simple example the potential benefits of integrating symbolic and subsymbolic techniques according to this general framework. A symbolic path-planner suggests intermediate configurations—subgoals or landmarks—that the path has to go through. The path-planner is invoked both at the beginning of the task and whenever the course of action seems to be wrong. This happens when the path-finder cannot find suitable action to handle the current situation or when the robot deviates considerably from the planned physical path.

Acknowledgments

We thank Rich Sutton whose suggestions have been really valuable for improving the presentation of our work, and Aristide Varfis for helpful discussions. We would like also to express our gratitude to two of the reviewers for their detailed comments. Carme Torras has been partially supported by the ESPRIT Basic Research Action number 3234.

Notes

1. Although the notion of *configuration space* is that of a "state space" and thus it is quite old, the term itself was first introduced by Lozano-Pérez and Wesley (1979) and subsequently developed in Lozano-Pérez (1983). Roughly speaking, it is the *space of degrees of freedom of motion* in which a given motion planning problem is to be solved.
2. If the reinforcement signal were computed using *local information*, then the task would be to select actions that optimize the *cumulative reinforcement* received by the path-finder over time. A prediction problem would have to be solved, *temporal-difference methods* (Sutton, 1988; Barto, et al., 1989; Watkins, 1989) having proved to be useful for this kind of task.

References

Agre, P.E., & Chapman, D. (1987). Pengi: An implementation of a theory of activity. *Proceedings of the Seventh AAAI Conference* (pp. 268–272).

Anderson, C.W. (1986). *Learning and problem solving with multilayer connectionist systems*. Ph.D. Thesis, Dept. of Computer and Information Science, University of Massachusetts, Amherst.

Anderson, C.W. (1987). Strategy learning with multilayer connectionist representations. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 103–114).

Arkins, R.C. (1987). Motor schema based navigation for a mobile robot: An approach to programming by behavior. *Proceedings of the IEEE International Conference on Robotics and Automation* (pp. 264–271).

Barto, A.G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology, 4*, 229–256.

Barto, A.G., & Anandan, P. (1985). Pattern-recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics, 15*, 360–374.

Barto, A.G., Sutton, R.S., & Anderson, C.W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics, 13*, 835–846.

Barto, A.G., Sutton, R.S., & Brouwer, P.S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics, 40*, 201–211.

Barto, A.G., Sutton, R.S. & Watkins, C.J.C.H. (1989). *Learning and sequential decision making* (Technical Report COINS-89-95). University of Massachusetts, Amherst, MA: Dept. of Computer and Information Science.

Blythe, J., & Mitchell, T.M. (1989). On becoming reactive. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 255–259).

Brady, M., Hollerbach, J.M., Johnson, T.L., Lozano-Pérez, T., & Mason, M.T., (Eds.) (1982). *Robot motion: Planning and control*. Cambridge, MA: MIT Press.

Brooks, R.A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation, 2*, 14–23.

Canny, J.F. (1988). *The complexiity of robot motion planning*. Cambridge, MA: MIT Press.

Chapman, D., & Kaelbling, L.P. (1990). *Learning from delayed reinforcement in a complex domain* (Technical Report 90-11). Palo Alto, CA: Teleos Research.

Donald, B.R. (1987). A search algorithm for robot motion planning with six degrees of freedom. *Artificial Intelligence, 31*, 295–353.

Graf, D.H., & LaLonde, W.R. (1988). A neural controller for collision-free movement of general robot manipulators. *Proceedings of the IEEE Second International Conference on Neural Networks, Vol I* (pp. 77–84).

Gullapalli, V. (1988). *A stochastic algorithm for learning real-valued functions via reinforcement feedback* (Technical Report COINS-88-91). University of Massachusetts, Amherst, MA: Dept. of Computer and Information Science.

Ilari, J., & Torras, C. (1990). 2D path planning: A configuration space heuristic approach. *The International Journal of Robotics Research, 9*, 75–91.

Jordan, M.I., & Jacobs, R.A. (1990). Learning to control an unstable system with forward modeling. In D.S. Touretzky (Ed.), *Advances in neural information procesing systems 2*, 324–331. San Mateo, CA: Morgan Kaufmann.

Jorgensen, C.C. (1987). Neural network representation of sensor graphs in autonomous robot navigation. *Proceedings of the IEEE First International Conference on Neural Networks, Vol IV* (pp. 507–515).

Khatib, O. (1986). Real time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research, 5,* 90–98.

Langley, P. (1985). Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science, 9,* 217–260.

Lin, L.-J. (1990). Self-improving reactive agents: Case studies of reinforcement learning frameworks. *Proceedings of the First International Conference on the Simulation of Adaptive Behavior: From Animals to Animats* (pp. 297–305).

Lozano-Pérez, T. (1983). Spatial planning: A configuration space approach. *IEEE Transactions on Computers, 32,* 108–120.

Lozano-Pérez, T., & Wesley, M. (1979). An algorithm for planning collison-free paths among polyhedral obstacles. *Communications of the ACM, 22,* 560–570.

Mahadevan, S., & Connell, J. (1990). *Automatic programming of behavior-based robots using reinforcement learning* (Technical Report RC 16359). Yorktown Heights, NY: IBM, T.J. Watson Research Center.

Mel, B.W. (1989). MURPHY: *A neurally-inspired connectionist approach to learning and performance in vision-based robot motion planning.* Ph.D. Thesis, Graduate College, University of Illinois, Urbana-Champaign.

Millán, J. del R., & Torras, C. (1990). Reinforcement learning: Discovering stable solutions in the robot path finding domain. *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 219–221).

Millán, J. del R., & Torras, C. (1991a). Connectionist approaches to robot path finding. In O.M. Omidvar (Ed.), *Progress in neural networks series, Vol 3.* Norwood, NJ: Ablex.

Millán, J. del R., & Torras, C. (1991b). Learning to avoid obstacles through reinforcement. In L. Birnbaum & G. Collins (Eds.) *Machine learning: Proceedings of the Eighth International Workshop,* 298–302. San Mateo, CA: Morgan Kaufmann.

Mozer, M.C., & Bachrach, J. (1989). *Discovering the structure of a reactive environment by exploration* (Technical Report CU-CS-451-89). Boulder, CO: University of Colorado, Dept. of Computer Science.

Munro, P. (1987). A dual back-propagation scheme for scalar reward learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 165–176).

Rivest, R.L., & Schapire, R.E. (1987). A new approach to unsupervised learning in deterministic environments. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 364–375).

Robinson, A.J. (1989). *Dynamic error propagation networks.* Ph.D. Thesis, Engineering Department, Cambridge University, Cambridge, England.

Saerens, M., & Soquet, A. (1989). A neural controller. *Proceedings of the First IEE International Conference on Artificial Neural Networks* (pp. 211–215).

Schoppers, M.J. (1987). Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039–1046).

Singh, S.P. (1991). Transfer of learning across compositons of sequential tasks. In L. Birnbaum & G. Collins (Eds.) *Machine learning: Proceedings of the Eighth International Workshop,* 348–352. San Mateo, CA: Morgan Kaufmann.

Steels, L. (1988). Steps towards common sense. *Proceedings of the Eighth European Conference on Artificial Intelligence* (pp. 49–54).

Sutton, R.S. (1984). *Temporal credit assignment in reinforcement learning.* Ph.D. Thesis, Dept. of Computer and Information Science, 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. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 216–224).

Torras, C. (1990). Motion planning and control: Symbolic and neural levels of computation. *Proceedings of the Third COGNITIVA Conference* (pp. 207–218).

Watkins, C.J.C.H. (1989). *Learning with delayed rewards.* Ph.D. Thesis, Psychology Department, Cambridge University, Cambridge, England.

Werbos, P.J. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics, 17,* 7–20.

Whitesides, S.H. (1985). Computational geometry and motion planning. In G. Toussaint (Ed.), *Computational geometry.* Amsterdam, New York, Oxford: North-Holland.

Williams, R.J. (1986). *Reinforcement learning in connectionist networks: A mathematical analysis* (Technical Report ICS-8605). San Diego, CA: University of California, Institute for Cognitive Science.

Williams, R.J. (1987). *Reinforcement-learning connectionist systems* (Technical Report NU-CCS-87-3). Northeastern University, Boston, MA: College of Computer Science.

Yap, C.-K. (1987). Algorithmic motion planning. In J.T. Schwartz & C.-K. Yap (Eds.), *Advances in robotics, Vol. I. Algorithmic and geometric aspects of robotics*, 95–143. Hillsdale, NJ: Lawrence Erlbaum.

Appendix A: Computation of the repulsion forces

The intensity of each repulsion force depends on three factors, namely r_a, r_b and r_c. Assuming circular obstacles, each obstacle is determined by three parameters: the *coordinates* of the *center* of the circle (x_o, y_o) and the *radius ra*. If the current and goal configurations are the points (x_a, y_a) and (x_g, y_g), respectively, then the points of the SPV are defined by the equation:

$$(x_{SPV}, y_{SPV}) = (1 - p)(x_a, y_a) + p(x_g, y_g), \tag{28}$$

where $p \in [0, 1]$. So, each point of conflict (x_c, y_c) is determined by the value p_c which solves the differential equation:

$$\frac{d\ \text{dist}[(x_o, y_o), (x_c, y_c)]}{dp} = 0, \tag{29}$$

and by the additional constraint that it must be outside the corresponding obstacle. This value is:

$$p_c = \text{box}_1\left(-\frac{(x_o - x_a)(x_a - x_g) + (y_o - y_a)(y_a - y_g)}{(x_a - x_g)^2 + (y_a - y_g)^2}\right), \tag{30}$$

where

$$\text{box}_1(x) = \begin{cases} 1, & \text{if } x > 1, \\ 0, & \text{if } x < 0, \\ x, & \text{otherwise.} \end{cases} \tag{31}$$

Now, the three repulsion factors are given by the expressions:

$$r_a = \text{box}_2(\text{dist}[(x_o, y_o), (x_c, y_c)], ra), \tag{32}$$

$$r_b = \text{box}_3(\text{dist}[(x_a, y_a), (x_c, y_c)], \text{dist}[(x_o, y_o), (x_c, y_c)], ra), \tag{33}$$

$$r_c = \text{box}_4(\text{dist}[(x_o, y_o), (x_c, y_c)], ra), \tag{34}$$

where

$$\text{box}_2(x, y) = \begin{cases} 0, & \text{if } x < y, \\ limit(x - y), & \text{otherwise,} \end{cases} \tag{35}$$

$$\text{box}_3(x, y, z) = \begin{cases} limit(x), & \text{if } y > z, \\ limit(x - \sqrt{z^2 - y^2}), & \text{otherwise,} \end{cases} \tag{36}$$

$$\text{box}_4(x, y) = \begin{cases} 0, & \text{if } x > y, \\ y - x, & \text{otherwise,} \end{cases} \tag{37}$$

and

$$limit(x) = \begin{cases} k_{pa} * x, & \text{if } x > perception_{range}, \\ x, & \text{otherwise.} \end{cases} \tag{38}$$

This *limit* function is defined in such a way as to keep the repulsion forces very small in those cases where obstacles are located outside the *perception range* of the robot (see (39) below). k_{pa} and $perception_{range}$ are constants, $k_{pa} \gg 1$, and $perception_{range}$ is chosen to be a fifth of the dimension of the workspace

Once r_a, r_b and r_c have been calculated, the intensity of a repulsion force is:

$$ir(r_a, r_b, r_c) = e^{-(k_{r_a} * r_a + k_{r_b} * r_b)} + \frac{1}{1 - e^{k_{r_c} * r_c}} - \frac{1}{2}, \tag{39}$$

where k_{r_a}, k_{r_b} and k_{r_c} are constants.

The intensity of the environmental repulsion force from a quadrant is:

$$r^i(x) = \begin{cases} 1, & \text{if a collision happens in the quadrant } i, \\ \dfrac{1}{1 + e^{-(k_{rep} * x)}}, & \text{otherwise,} \end{cases} \tag{40}$$

where k_{rep} is a constant and x is the sum of the intensities of the repulsion forces from quadrant i.

Appendix B: Analysis of variance

Table 6 provides the standard deviation in the number of steps for finding a stable qo-path of the most promising versions. Table 7 shows the *significance degree of the means differences* when every version is compared with each other.

Table 6. Standard deviation in the number of steps for finding a stable qo-path.

		Eligibility		
		3	4	5
Baseline	1	15654	11477	5249
	4	9541	9814	8655
	5	9145	9318	10948

Table 7. Significance degrees of the means differences. $\mu_1 < \mu_2$, where μ_i is the mean of the sample i.

	Sample 2								
Sample 1	v_{45}	v_{15}	v_{54}	v_{43}	v_{44}	v_{14}	v_{13}	v_{53}	v_{55}
v_{45}		80.23%	87.08%	92.07%	96.25%	96.64%	95.73%	99.65%	99.84%
v_{15}			70.54%	81.06%	90.66%	91.92%	90.99%	99.20%	99.62%
v_{54}				61.03%	74.54%	78.81%	81.59%	93.32%	96.78%
v_{43}					64.43%	70.54%	75.49%	88.49%	94.18%
v_{44}						57.53%	65.54%	78.81%	88.49%
v_{14}							59.10%	69.85%	82.12%
v_{13}								56.75%	70.19%
v_{53}									68.79%
v_{55}									

In order to compute the significance degree of the difference between a pair of sample means μ_1 and μ_2, the following *discriminant function* is built:

$$u = (\mu_1 - \mu_2) \sqrt{\frac{1}{\frac{\sigma_1^2}{n1 - 1} + \frac{\sigma_2^2}{n2 - 1}}}, \tag{41}$$

where $n1$ and $n2$ are the size of each sample, and σ_1 and σ_2 are the sample standard deviations. Now, if experimentally $\mu_2 > \mu_1$, then for estimating the significance degree $1 - e$ it is necessary to find u_{1-e}, in the table of the normal distribution, such that $u < -u_{1-e}$.

Data in Table 7 confirm the results of Section 5.1. In addition, it is possible to draw the following two main conclusions. First, the versions are ranked in the following order: v_{45}—i.e., the version using the fourth baseline an fifth eligibility factor—v_{15}, v_{54}, v_{43}, v_{44}, v_{14}, v_{13}, v_{53} and v_{55}. Second, v_{45} is *significantly better* then v_{53} and v_{55}—the corresponding significance degrees are greater than 99%—it is *quasi-significantly better* than v_{13}, v_{14} and v_{44}—the corresponding significance degrees are greater than 95%—it is *presumably better* than v_{43}—the significance degree is greater than 90%—and it is *likely better* than v_{15} and v_{54}—the corresponding significance degrees are greater than 80%.

INDEX