PROBLEM SOLVING WITH REINFORCEMENT LEARNING

Gavin Adrian Rummery

Cambridge University Engineering Department
Trumpington Street
Cambridge CB2 1PZ
England

This dissertation is submitted for consideration for the degree of Doctor of Philosophy at the University of Cambridge

Summary

This thesis is concerned with practical issues surrounding the application of *reinforcement learning* techniques to tasks that take place in high dimensional continuous state-space environments. In particular, the extension of *on-line* updating methods is considered, where the term implies systems that learn as each experience arrives, rather than storing the experiences for use in a separate off-line learning phase. Firstly, the use of alternative update rules in place of standard Q-learning (Watkins 1989) is examined to provide faster convergence rates. Secondly, the use of multi-layer perceptron (MLP) neural networks (Rumelhart, Hinton and Williams 1986) is investigated to provide suitable generalising function approximators. Finally, consideration is given to the combination of Adaptive Heuristic Critic (AHC) methods and Q-learning to produce systems combining the benefits of real-valued actions and discrete switching.

The different update rules examined are based on Q-learning combined with the TD(λ) algorithm (Sutton 1988). Several new algorithms, including Modified Q-Learning and Summation Q-Learning, are examined, as well as alternatives such as Q(λ) (Peng and Williams 1994). In addition, algorithms are presented for applying these Q-learning updates to train MLPs on-line during trials, as opposed to the backward-replay method used by Lin (1993b) that requires waiting until the end of each trial before updating can occur.

The performance of the update rules is compared on the Race Track problem of Barto, Bradtke and Singh (1993) using a lookup table representation for the Q-function. Some of the methods are found to perform almost as well as Real-Time Dynamic Programming, despite the fact that the latter has the advantage of a full world model.

The performance of the connectionist algorithms is compared on a larger and more complex robot navigation problem. Here a simulated mobile robot is trained to guide itself to a goal position in the presence of obstacles. The robot must rely on limited sensory feedback from its surroundings and make decisions that can be generalised to arbitrary layouts of obstacles. These simulations show that the performance of on-line learning algorithms is less sensitive to the choice of training parameters than backward-replay, and that the alternative Q-learning rules of Modified Q-Learning and Q(λ) are more robust than standard Q-learning updates.

Finally, a combination of real-valued AHC and Q-learning, called Q-AHC learning, is presented, and various architectures are compared in performance on the robot problem. The resulting reinforcement learning system has the properties of providing on-line training, parallel computation, generalising function approximation, and continuous vector actions.

Acknowledgements

I would like to thank all those who have helped in my quest for a PhD, especially Chen Tham with whom I had many heated discussions about the details of reinforcement learning. I would also like to thank my supervisor, Dr. Mahesan Niranjan, who kept me going after the unexpected death of my original supervisor, Prof. Frank Fallside. Others who have contributed with useful discussions have been Chris Watkins and Tim Jervis. I also owe Rich Sutton an apology for continuing to use the name Modified Q-Learning whilst he prefers SARSA, but thank him for the insightful discussion we had on the subject.

Special thanks to my PhD draft readers: Rob Donovan, Jon Lawn, Gareth Jones, Richard Shaw, Chris Dance, Gary Cook and Richard Prager.

This work has been funded by the Science and Engineering Research Council with helpful injections of cash from the Engineering Department and Trinity College.

Dedication

I wish to dedicate this thesis to Rachel, who has put up with me for most of my PhD, and mum and dad, who have put up with me for most of my life.

Declaration

This 38,000 word dissertation is entirely the result of my own work and includes nothing which is the outcome of work done in collaboration.

Gavin Rummery
Trinity College
July 26, 1995

Contents

Chapter 1

Introduction

Problem: *A system is required to interact with an environment in order to achieve a particular task or goal. Given that it has some feedback about the current state of the environment, what action should it take?*

The above represents the basic problem faced when designing a control system to achieve a particular task. Usually, the designer has to analyse a model of the task and decide on the sequence of actions that the system should perform to achieve the goal. Allowances must be made for noisy inputs and outputs, and the possible variations in the actual system components from the modelled ideals. This can be a very time consuming process, and so it is desirable to create systems that *learn* the actions required to solve the task for themselves. One group of methods for producing such autonomous systems is the field of *reinforcement learning*, which is the subject of this thesis.

With reinforcement learning, the system is left to experiment with actions and find the optimal policy by trial and error. The quality of the different actions is reinforced by awarding the system payoffs based on the outcomes of its actions — the nearer to achieving the task or goal, the higher the payoffs. Thus, by favouring taking actions which have been learnt to result in the best payoffs, the system will eventually converge on producing the optimal action sequences.

The motivation behind the work presented in this thesis comes from attempts to design a reinforcement learning system to solve a simple mobile robot navigation task (which is used as a testbed in chapter 4). The problem is that much of the theory of reinforcement learning has concentrated on discrete Markovian environments, whilst many tasks cannot be easily or accurately modelled by this formalism. One popular way around this is to partition continuous environments into discrete states and then use the standard discrete methods, but this was not found to be successful for the robot task. Consequently, this thesis is primarily concerned with examining the established reinforcement learning methods to extend and improve their operation for large continuous state-space problems.

The next two sections briefly discuss alternative methods to reinforcement learning for creating systems to achieve tasks, whereas the remainder of the chapter concentrates on providing an introduction to reinforcement learning.

1.1 Control Theory

Most control systems are designed by mathematically modelling and analysing the problem using methods developed in the field of *control theory*. Control theory concentrates on trajectory tracking, which is the task of generating actions to move stably from one part of an environment to another. To build systems capable of performing more complex tasks, it is necessary to decide the overall sequence of trajectories to take. For example, in a robot navigation problem, control theory could be used to produce the motor control sequences necessary to keep the robot on a pre-planned path, but it would be up to a higher-level part of the system to generate this path in the first place.

Although many powerful tools exist to aid the design of controllers, the difficulty remains that the resulting controller is limited by the accuracy of the original mathematical model of the system. As it is often necessary to use approximate models (such as linear approximations to non-linear systems) owing to the limitations of current methods of analysis, this problem increases with the complexity of the system being controlled. Furthermore, the final controller must be built using components which match the design within a certain tolerance. Adaptive methods do exist to tune certain parameters of the controller to the particular system, but these still require a reasonable approximation of the system to be controlled to be known in advance.

1.2 Artificial Intelligence

At the other end of the scale, the field of Artificial Intelligence (AI) deals with finding sequences of high-level actions. This is done by various methods, mainly based on performing searches of action sequences in order to find one which solves the task. This sequence of actions is then passed to lower-level controllers to perform. For example, the kind of action typically used by an AI system might be `pick-up-object`, which would be achieved by invoking increasingly lower levels of AI or control systems until the actual motor control actions were generated.

The difficulty with this type of system is that although it searches for solutions to tasks by itself, it still requires the design of each of the high-level actions, including the underlying low-level control systems.

1.3 Reinforcement Learning

Reinforcement learning is a class of methods whereby the problem to be solved by the control system is defined in terms of *payoffs* (which represent rewards or punishments). The aim of the system is to maximise[1] the payoffs received over time. Therefore, high payoffs are given for desirable behaviour and low payoffs for undesirable behaviour. The system is otherwise unconstrained in its sequence of actions, referred to as its *policy*, used to maximise the payoffs received. In effect, the system must find its own method of solving the given task.

For example, in chapter 4, a mobile robot is required to guide itself to a goal location in the presence of obstacles. The reinforcement learning method for tackling this problem

[1] Or minimise, depending on how the payoffs are defined. Throughout this thesis, increasing payoffs imply increasing rewards and therefore the system is required to maximise the payoffs received.

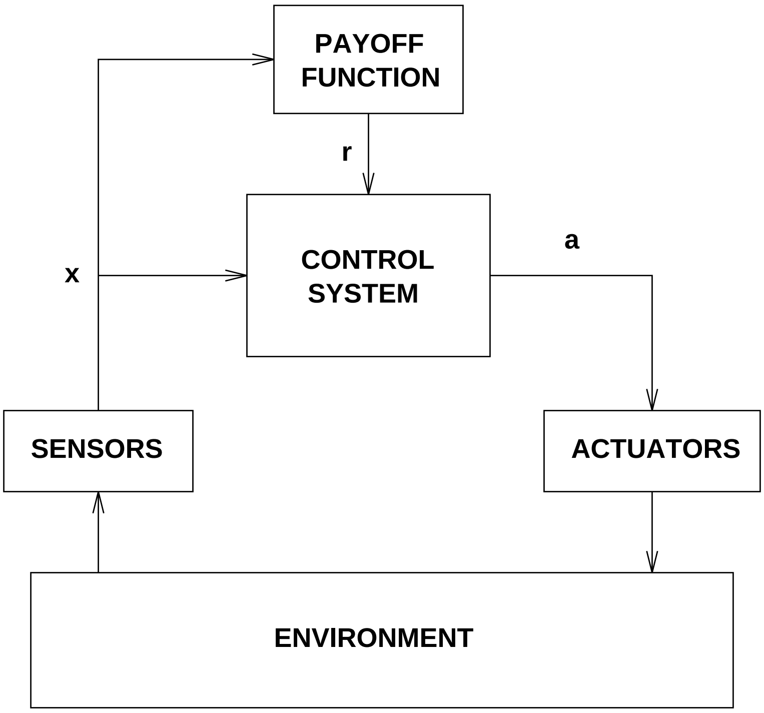


Figure 1.1: Diagram of a reinforcement learning system.

is to give the system higher payoffs for arriving at the goal than for crashing into the obstacles. The sequence of control actions to use can then be left to the system to determine for itself based on its motivation to maximise the payoffs it receives.

A block diagram of a reinforcement system is shown in Fig. 1.1, which shows the basic interaction between a controller and its environment. The payoff function is fixed, as are the sensors and actuators (which really form part of the environment as far as the control system is concerned). The control system is the adaptive part, which learns to produce the control action $\mathbf{a}$ in response to the state input $\mathbf{x}$ based on maximising the payoff $\mathbf{r}$.

1.3.1 The Environment

The information that the system knows about the environment at time step t can be encoded in a *state description* or *context* vector, $\mathbf{x}_t$. It is on the basis of this information that the system selects which action to perform. Thus, if the state description vector does not include all salient information, then the system's performance will suffer as a result.

The *state-space*, $\mathbf{X}$, consists of all possible values that the state vector, $\mathbf{x}$, can take. The state-space can be discrete or continuous.

Markovian Environments

Much of the work (in particular the convergence proofs) on reinforcement learning has been developed by considering finite-state Markovian domains. In this formulation, the environment is represented by a discrete set of state description vectors, $\mathbf{X}$, with a discrete set of *actions*, A, that can be performed in each state (in the general case, the available actions may be dependent on the state i.e. $A(\mathbf{x})$). Associated with each action in each state is a set of *transition probabilities* which determine the probability $P(\mathbf{x}_j|\mathbf{x}_i, a)$ of moving from state $\mathbf{x}_i \in \mathbf{X}$ to state $\mathbf{x}_j \in \mathbf{X}$ given that action $a \in A$ is executed. It should be noted that in most environments $P(\mathbf{x}_j|\mathbf{x}_i, a)$ will be zero for the vast majority of states $\mathbf{x}_j$ — for example, in a deterministic environment, only one state can be reached from $\mathbf{x}_i$ by action a, so the state transition probability is 1 for this transition and 0 for all others.

The set of state transition probabilities models the environment in which the control system is operating. If the probabilities are known to the system, then it can be said to possess a *world model.* However, it is possible for the system to be operating in a Markovian domain where these values are not known, or only partially known, *a-priori.*

1.3.2 Payoffs and Returns

The payoffs are scalar values, $r(\mathbf{x}_i, \mathbf{x}_j)$, which are received by the system for transitions from one state to another. In the general case, the payoff may come from a probability distribution, though this is rarely used. However, the payoffs seen in each state of a discrete model may appear to come from a probability distribution if the underlying state-space is continuous.

In simple reinforcement learning systems, the most desirable action is the one that gives the highest *immediate* payoff. Finding this action is known as the *credit assignment* problem. In this formulation long term considerations are not taken into account, and the system therefore relies on the payoffs being a good indication of the optimal action to take at each time step. This type of system is most appropriate when the result to be achieved at each time step is known, but the action required to achieve it is not clear. An example is the problem of how to move the tip of a multi-linked robot arm in a particular direction by controlling all the motors at the joints (Gullapalli, Franklin and Benbrahim 1994).

This type of payoff strategy is a subset of the more general *temporal credit assignment* problem, wherein a system attempts to maximise the payoffs received over a number of time steps. This can be achieved by maximising the expected sum of discounted payoffs received, known as the *return*, which is equal to,

$$E\left\{\sum_{t=0}^{\infty} \gamma^t r_t\right\} \tag{1.1}$$

where the notation r_t is used to represent the payoff received for the transition at time step t from state $\mathbf{x}_t$ to $\mathbf{x}_{t+1}$ i.e. $r(\mathbf{x}_t, \mathbf{x}_{t+1})$. The constant $0 \leq \gamma \leq 1$ is called the *discount factor.* The discount factor ensures that the sum of payoffs is finite and also adds more weight to payoffs received in the short-term compared with those received in the long-term. For example, if a non-zero payoff is only received for arriving at a goal state, then the system will be encouraged to find a policy that leads to a goal state in the shortest amount of time. Alternatively, if the system is only interested in immediate payoffs, then this is equivalent to $\gamma = 0$.

The payoffs define the problem to be solved and the constraints on the control policy used by the system. If payoffs, either good or bad, are not given to the system for desirable/undesirable behaviour, then the system may arrive at a solution which does not satisfy the requirements of the designer. Therefore, although the design of the system is simplified by allowing it to discover the control policy for itself, the task must be fully described by the payoff function. The system will then tailor its policy to its specific environment, which includes the controller sensors and actuators.

1.3.3 Policies and Value Functions

The overall choice of actions that is made by the system is called the policy, π. The policy need not be deterministic; it may select actions from a probability distribution.

The system is aiming to find the policy which maximises the return from all states $\mathbf{x} \in \mathbf{X}$. Therefore, a *value function*, $V^{\pi}(\mathbf{x})$, which is a prediction of the return available from each state, can be defined for any policy π,

$$V^{\pi}(\mathbf{x}_t) = E\left\{\sum_{k=t}^{\infty} \gamma^{k-t} r_k\right\} \tag{1.2}$$

The policy, π^*, for which $V^{\pi^*}(\mathbf{x}) \geq V^{\pi}(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{X}$ is called the *optimal policy*, and finding π^* is the ultimate aim of a reinforcement learning control system.

For any state $\mathbf{x}_i \in \mathbf{X}$, equation 1.2 can be rewritten in terms of the value function predictions of states that can be reached by the next state transition,

$$V^{\pi}(\mathbf{x}_i) = \sum_{\mathbf{x}_j \in \mathbf{X}} P(\mathbf{x}_j|\mathbf{x}_i, \pi)\left[r(\mathbf{x}_i, \mathbf{x}_j) + \gamma V^{\pi}(\mathbf{x}_j)\right] \tag{1.3}$$

for discrete Markovian state-spaces. This allows the value function to be learnt iteratively for any policy π. For continuous state-spaces, the equivalent is,

$$V^{\pi}(\mathbf{x}_i) = \int_{\mathbf{X}} p(\mathbf{x}|\mathbf{x}_i, \pi)\left[r(\mathbf{x}_i, \mathbf{x}) + \gamma V^{\pi}(\mathbf{x})\right] d\mathbf{x} \tag{1.4}$$

where $p(\mathbf{x}|\mathbf{x}_i, \pi)$ is the state-transition probability distribution. However, in the remainder of this introduction, only discrete Markovian state-spaces are considered.

1.3.4 Dynamic Programming

A necessary and sufficient condition for a value function to be optimal for each state $\mathbf{x}_i \in \mathbf{X}$ is that,

$$V^{\pi^*}(\mathbf{x}_i) = \max_{a \in A} \sum_{\mathbf{x}_j \in \mathbf{X}} P(\mathbf{x}_j|\mathbf{x}_i, a)\left[r(\mathbf{x}_i, \mathbf{x}_j) + \gamma V^{\pi^*}(\mathbf{x}_j)\right] \tag{1.5}$$

This is called *Bellman's Optimality Equation* (Bellman 1957). This equation forms the basis for reinforcement learning algorithms that make use of the principles of *dynamic programming* (Ross 1983, Bertsekas 1987), as it can be used to drive the learning of improved policies.

The reinforcement learning algorithms considered in this section are applicable to systems where the state transition probabilities are known i.e. the system has a world model. A world model allows the value function to be learnt *off-line*, as the system does not need to interact with its environment in order to collect information about transition probabilities or payoffs.

The basic principle is to use a type of dynamic programming algorithm called *value iteration*. This involves applying Bellman's Optimality Equation (equation 1.5) directly as an update rule to improve the current value function predictions,

$$V(\mathbf{x}_i) \leftarrow \max_{a \in A} \sum_{\mathbf{x}_j \in \mathbf{X}} P(\mathbf{x}_j|\mathbf{x}_i, a)\left[r(\mathbf{x}_i, \mathbf{x}_j) + \gamma V(\mathbf{x}_j)\right] \tag{1.6}$$

The above equation allows the value function predictions to be updated for each state, but only if the equation is applied at each $\mathbf{x}_i \in \mathbf{X}$.[2] Further, in order to converge, this equation has to be applied at each state repeatedly.

The optimal policy is therefore found from the optimal value function, rather than vice versa, by using the actions a which maximise the above equation in each state $\mathbf{x}_i$. These are called the *greedy actions* and taking them in each state is called the *greedy policy*. It should be noted that the optimal policy, π^*, may be represented by the greedy policy of the current value function without the value function having actually converged to the *optimal* value function. In other words, the actions that currently have the highest predictions of return associated with them may be optimal, even though the predictions are not. However, there is currently no way of determining whether the optimal policy has been found prematurely from a non-optimal value function.

The update rule can be applied to states in any order, and is guaranteed to converge towards the optimal value function as long as all states are visited repeatedly and an optimal policy does actually exist (Bertsekas 1987, Bertsekas and Tsiksiklis 1989). One algorithm to propagate information is therefore to synchronously update the value function estimates at every state. However, for convergence the order of updates does not matter and so they can be performed asynchronously at all states $\mathbf{x}_i \in \mathbf{X}$ one after another (a Gauss-Seidel sweep). This can result in faster convergence because the current update may benefit from information propagated by previous updates. This can be seen by considering equation 1.6; if the states $\mathbf{x}_j$ that have high probabilities of being reached from state $\mathbf{x}_i$ have just been updated, then this will improve the information gained by applying this equation.

Unfortunately, dynamic programming methods can be very computationally expensive, as information may take many passes to propagate back to states that require long action sequences to reach the goal states. Consequently, in large state-spaces the number of updates required for convergence can become impractical.

Barto et al. (1993) introduced the idea of *real-time dynamic programming*, where the only regions learnt about are those that are actually visited by the system during its normal operation. Instead of updating the value function for every state in $\mathbf{X}$, the states to be updated are selected by performing *trials*. In this method, the system performs an update at state $\mathbf{x}_t$ and then performs the greedy action to arrive in a new state $\mathbf{x}_{t+1}$. This can greatly reduce the number of updates required to reach a usable policy. However in order to *guarantee* convergence the system must still repeatedly visit all the states occasionally. If it does not, it is possible for the optimal policy to be missed if it involves sequences of actions that are never tested. This problem is true of all forms of real-time reinforcement learning but must be traded against faster learning times, or tractability, which may make full searches impractical.

In this thesis, two methods are examined for speeding up convergence. The first is to use *temporal difference* methods, which are described in outline in section 1.3.8 and examined in much greater detail in chapter 2. The second is to use some form of generalising function approximator to represent $V(\mathbf{x})$, as for many systems the optimal value function is a smooth function of $\mathbf{x}$ and thus for states close in state-space the values $V(\mathbf{x})$ are close too. This issue is examined in chapter 3, where methods are presented for using *neural networks* for reinforcement learning.

[2] Note that the update equation 1.6 is only suitable for discrete state-spaces. By considering equation 1.4 it can be seen that the equivalent continuous state-space update would involve integrating across a probability distribution, which could make each update very computationally expensive.

1.3.5 Learning without a Prior World Model

If a model of the environment is not available *a-priori*, then there are two options:

- Learn one from experience.
- Use methods which do not require one.

In both cases a new concept is introduced — that of *exploration*. In order to learn a world model, the system must try out different actions in each state to build up a picture of the state-transitions that can occur. On the other hand, if a model is not being learnt, then the system must explore in order to update its value function successfully.

Learning a World Model

If a world model is not known in advance, then it can be learnt by trials on the environment. Learning a world model can either be treated as a separate task (*system identification*), or can be performed simultaneously with learning the value function (as in *adaptive real-time dynamic programming* (Barto et al. 1993)). Once a world model has been learnt, it can also be used to perform value function updates off-line (Sutton 1990, Peng and Williams 1993) or for planning ahead (Thrun and Möller 1992).

Learning a model from experience is straight-forward in a Markovian domain. The basic method is to keep counters of the individual state transitions that occur and hence calculate the transition probabilities using,

$$P(\mathbf{x}_j|\mathbf{x}_i, a) = \frac{n(\mathbf{x}_i, a, \mathbf{x}_j)}{n(\mathbf{x}_i, a)} \tag{1.7}$$

where $n(\mathbf{x}_i, a)$ is the count of the number of times the action a has been used in state $\mathbf{x}_i$, and $n(\mathbf{x}_i, a, \mathbf{x}_j)$ is the count of the number of times performing this action has led to a transition from state $\mathbf{x}_i$ to state $\mathbf{x}_j$. If there are any prior estimates of the values of the probabilities, they can be encoded by initialising the counters in the appropriate proportions, which may help accelerate convergence.

However, learning world models in more complex environments (especially continuous state-spaces) may not be so easy, at least not to a useful accuracy. If an inaccurate model is used, then the value function learnt from it will not be optimal and hence nor will the resulting greedy policy. The solution is to use value function updating methods that do not require a world model. This is because predicting a scalar expected return in a complex environment is relatively easy compared with trying to predict the probability distribution across the next state vector values. It is this type of reinforcement learning method that is examined throughout the remainder of this thesis.

Alternatives to Learning a World Model

If a model of the environment is not available, and the system cannot learn one, then the value function updates must be made based purely on experience i.e. they must be performed *on-line* by interacting with the environment. More specifically, on each visit to a state, only one action can be performed, and hence information can only be learnt from the outcome of that action. Therefore, it is very important to use methods that make maximum use of the information gathered in order to reduce the number of trials that need to be performed.

There are two main classes of method available:

- **Adaptive Heuristic Critic** methods, which keep track of the current policy and value function separately.
- **Q-Learning** methods which learn a different form of value function which also defines the policy.

These methods are examined in the following sections.

1.3.6 Adaptive Heuristic Critic

The *Adaptive Heuristic Critic* (AHC) is actually a form of dynamic programming method called *policy iteration*. With policy iteration, value functions and policies are learnt iteratively from one another by repeating the following two phases,

1. Learn a value function for the current fixed policy.
2. Learn the greedy policy with respect to the current fixed value function.

Repeatedly performing both phases to completion is likely to be computationally expensive even for small problems, but it is possible for a phase to be performed for a fixed number of updates before switching to the other (Puterman and Shin 1978). The limiting case for policy iteration is to update the value function and policy simultaneously, which results in the Adaptive Heuristic Critic class of methods.

The original AHC system (Barto, Sutton and Anderson 1983, Sutton 1984) consists of two elements:

- **ASE** The *Associative Search Element* chooses actions from a stochastic policy.
- **ACE** The *Adaptive Critic Element* learns the value function.

These two elements are now more generally called the *actor* and the *critic* (thus AHC systems are often called *Actor-Critic* methods (Williams and Baird 1993a)). The basic operation of these systems is for the probability distribution used by the actor to select actions to be updated based on internal payoffs generated by the critic.

Because there is no world model available, the value function must be learnt using a different incremental update equation from that of equation 1.6, namely,

$$V(\mathbf{x}_t) \leftarrow V(\mathbf{x}_t) + \alpha \left[r_t + \gamma V(\mathbf{x}_{t+1}) - V(\mathbf{x}_t) \right] \tag{1.8}$$

where α is a learning rate parameter. This is necessary as the only way the prediction at state $\mathbf{x}_t$ can be updated is by performing an action and arriving at a state $\mathbf{x}_{t+1}$.[3]

Effectively, with each visit to a state $\mathbf{x}_i$, the value $V(\mathbf{x}_i)$ is updated by *sampling* from the possible state-transitions that may occur and so α acts as a first-order filter on the values seen. If the action taken each time the state is visited is fixed, then the next states $\mathbf{x}_j$ will be seen in proportion to the state-transition probabilities $P(\mathbf{x}_j|\mathbf{x}_i, a)$ and so the expected prediction $E\{V(\mathbf{x}_i)\}$ will converge.

The critic uses the error between successive predictions made by the value function to provide a measure of the quality of the action, a_t, that was performed,

$$\varepsilon_t = r_t + \gamma V(\mathbf{x}_{t+1}) - V(\mathbf{x}_t) \tag{1.9}$$

[3] The use of t as a subscript is to emphasise that these updates are performed for the state $\mathbf{x}_t, \mathbf{x}_{t+1}, \ldots$ in the order in which they are visited during a trial.

Hence, if the result of the selected action was better than predicted by $V(\mathbf{x}_t)$, then ε_t will be positive and can be used as a positive reinforcement to the action (and vice versa if it is negative). This value can be used as an immediate payoff in order to judge how the actor should be altered to improve the policy.

The actor uses the internal reinforcement, ε_t, to update the probability of the action, a_t, being selected in future. The exact manner in which this is done depends on the form of the actor. As an illustration, it can be performed for the case of discrete actions by summing the internal payoffs received over time,

$$W(\mathbf{x}_t, a_t) \leftarrow W(\mathbf{x}_t, a_t) + \varepsilon_t \tag{1.10}$$

These weighting values, $W(\mathbf{x}, a)$, can then be used as the basis on which the actor selects actions in the future, with the actor favouring the actions with higher weightings. Thus, actions which lead to states from which the expected return is improving will gain weighting and be selected with a higher probability in the future.

The advantage of AHC methods is that the actions selected by the actor can be real-valued, i.e. the actor can produce a continuous range of action values, rather than selecting from a discrete set A. This topic is investigated in chapter 5.

1.3.7 Q-Learning

In *Q-learning* (Watkins 1989), an alternative form of value function is learnt, called the Q-function. Here the expected return is learnt with respect to both the state and action,

$$Q^{\pi}(\mathbf{x}_i, a) = \sum_{\mathbf{x}_j \in \mathbf{X}} P(\mathbf{x}_j | \mathbf{x}_i, a) \left[r(\mathbf{x}_i, \mathbf{x}_j) + \gamma V^{\pi}(\mathbf{x}_j) \right] \tag{1.11}$$

The value $Q^{\pi}(\mathbf{x}_i, a)$ is called the *action value.* If the Q-function has been learnt accurately, then the value function can be related to it using,

$$V^{\pi}(\mathbf{x}) = \max_{a \in A} Q^{\pi}(\mathbf{x}, a) \tag{1.12}$$

The Q-function can be learnt when the state-transition probabilities are not known, in a similar way to the incremental value function update equation 1.8. The updates can be performed during trials using,

$$Q(\mathbf{x}_t, a_t) \leftarrow Q(\mathbf{x}_t, a_t) + \alpha \left[r_t + \gamma V(\mathbf{x}_{t+1}) - Q(\mathbf{x}_t, a_t) \right] \tag{1.13}$$

which by substituting equation 1.12, can be written entirely in terms of Q-function predictions,

$$Q(\mathbf{x}_t, a_t) \leftarrow Q(\mathbf{x}_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in A} Q(\mathbf{x}_{t+1}, a) - Q(\mathbf{x}_t, a_t) \right] \tag{1.14}$$

This is called the *one-step Q-learning* algorithm.

When the Q-function has been learnt, the policy can be determined simply by taking the action with the highest action value, $Q(\mathbf{x}, a)$, in each state, as this predicts the greatest future return. However, in the course of learning the Q-function, the system must perform actions other than suggested by the greedy policy in case the current Q-function predictions are wrong. The exploration policy used is critical in determining the rate of convergence of the algorithm, and though Q-learning has been proved to converge for discrete state-space Markovian problems (Watkins and Dayan 1992, Jaakkola, Jordan and Singh 1993), this is only on the condition that the exploration policy has a finite probability of visiting all states repeatedly.

1.3.8 Temporal Difference Learning

Temporal difference learning (Sutton 1988) is another incremental learning method that can be used to learn value function predictions. The algorithm is described in detail in the next chapter, but here a brief overview is given.

To explain the concept behind temporal difference learning (TD-learning), consider a problem where a sequence of predictions, $P_t, P_{t+1}, \ldots$, is being made of the expected value of a random variable r_T at a future time T. At this time, the predictions P_t for all $t < T$ could be improved by making changes of,

$$\Delta P_t = \alpha(r_T - P_t) \tag{1.15}$$

where α is a learning rate parameter. The above equation can be expanded in terms of the temporal difference errors between successive predictions i.e.

$$\begin{aligned} \Delta P_t &= \alpha\left[(P_{t+1} - P_t) + (P_{t+2} - P_{t+1}) + \cdots + (P_{T-1} - P_{T-2}) + (r_T - P_{T-1})\right] \\ &= \alpha \sum_{k=t}^{T-1} (P_{k+1} - P_k) \end{aligned} \tag{1.16}$$

where $P_T = r_T$. This means that at time step t, each prediction P_k for $k \leq t$ could be updated using the current TD-error, $(P_{t+1} - P_t)$. This idea forms the basis of temporal difference learning algorithms, as it allows the current TD-error to be used at each time step to update all previous predictions, and so removes the necessity to wait until time T before updating each prediction by applying equation 1.15.

In fact, Sutton introduced an entire family of temporal difference algorithms called TD(λ) where λ is a weighting on the importance of future TD-errors to the current prediction, such that,

$$\Delta P_t = \alpha \sum_{k=t}^{T-1} (P_{k+1} - P_k)\lambda^{t-k} \tag{1.17}$$

Therefore, equation 1.16 is called a TD(1) algorithm since it is equivalent to $\lambda = 1$. At the other end of the scale, if $\lambda = 0$ then each update ΔP_t is only based on the next temporal difference error, $(P_{t+1} - P_t)$. For this reason, one-step Q-learning (equation 1.14) and the incremental value function update (equation 1.8) are regarded as TD(0) algorithms, as they involve updates based only on the next TD-error. Potentially, therefore, the convergence rates of these methods can be improved by using temporal difference algorithms with $\lambda > 0$. The original AHC architecture of Barto et al. (1983) used this kind of algorithm for updating the ASE and ACE, and in the next chapter alternatives for performing Q-function updates with $\lambda > 0$ are discussed.

1.3.9 Limitations of Discrete State-Spaces

In this chapter, all of the algorithms have been discussed in relation to finite-state Markovian environments and hence it has been assumed that the information gathered is stored explicitly at each state as it is collected. This implies the use of a discrete storage method, such as a lookup-table, where each state vector, $\mathbf{x}_i \in \mathbf{X}$, is used to select a value, $V(\mathbf{x}_i)$, which is stored independently of all others. The number of entries required in the table is therefore equal to $|\mathbf{X}|$, which for even a low dimensional state vector $\mathbf{x}$ can be large. In the case of Q-learning, the number of independent values that must be stored to represent the function $Q(\mathbf{x}, a)$ is equal to $|\mathbf{X}||A|$, which is even larger. Furthermore, each of these values must be learnt, which requires multiple applications of the update rule, and hence the number of updates (or trials in the case of real-time methods) required becomes huge.

The problem is that in the above discussions, it has been assumed that there is absolutely no link between states in the state-space other than the transition probabilities. A factor that has not been examined is that states that are 'close' in the state-space (i.e. their state vectors $\mathbf{x}$ are similar) may require similar policies to be followed to lead to success and so have very similar predictions of future payoffs. This is where *generalisation* can help make seemingly intractable problems tractable, simply by exploiting the fact that experience gained by the system in one part of the state-space may be equally relevant to neighbouring regions. This becomes critical if reinforcement learning algorithms are to be applied to continuous state-space problems. In such cases the number of *discrete* states in $\mathbf{X}$ is infinite and so the system is unlikely to revisit exactly the same point in the state-space more than once.

1.4 Overview of the Thesis

Much of the work done in the reinforcement learning literature uses low dimensional discrete state-spaces. This is because reinforcement learning algorithms require extensive repeated searches of the state-space in order to propagate information about the payoffs available and so smaller state-spaces can be examined more easily. From a theoretical point of view, the only proofs of convergence available for reinforcement learning algorithms are based on information being stored explicitly at each state or using a linear weighting of the state vector. However, it is desirable to extend reinforcement learning algorithms to work efficiently in high dimensional continuous state-spaces, which requires that each piece of information learnt by the system is used to its maximum effect. Two factors are involved; the *update rule* and the *function approximation* used to generalise information between similar states. Consideration of these issues forms a major part of this thesis.

Over this chapter, a variety of reinforcement learning methods have been discussed, with a view to presenting the evolution of update rules that can be used without requiring a world model. These methods are well suited to continuous state-spaces, where learning an accurate world model may be a difficult and time-consuming task. Hence, the remainder of this thesis concentrates on reinforcement learning algorithms that can be used without the need to learn an explicit model of the environment.

The overall aim, therefore, is to examine reinforcement learning methods that can be applied to solving tasks in high dimensional continuous state-spaces, and provide robust, efficient convergence.

The remainder of the thesis is structured as follows,

Chapter 2: Watkins presented a method for combining Q-learning with TD(λ) to speed up convergence of the Q-function. In this chapter, a variety of alternative Q-learning update rules are presented and compared to see if faster convergence is possible. This includes novel methods called Modified Q-Learning and Summation Q-Learning, as well as Q(λ) (Peng and Williams 1994). The performance of the update rules is then compared empirically using the discrete state-space Race Track problem (Barto et al. 1993).

Chapter 3: One choice for a general function approximator that will work with continuous state inputs is the *multi-layer perceptron* (MLP) or *back-propagation neural network*. Although the use of neural networks in reinforcement problems has been examined before (Lin 1992, Sutton 1988, Anderson 1993, Thrun 1994, Tesauro 1992, Boyan 1992), the use of *on-line* training methods for performing Q-learning updates with $\lambda > 0$ has not been examined previously. These allow temporal difference methods to be applied during the trial as each reinforcement signal becomes available, rather than waiting until the end of the trial as has been required by previous connectionist Q-learning methods.

Chapter 4: The MLP training algorithms are empirically tested on a navigation problem where a simulated mobile robot is trained to guide itself to a goal position in a 2D environment. The robot must find its way to a goal position while avoiding obstacles, but only receives payoffs at the end of each trial, when the outcome is known (the only information available to it during a trial are sensor readings and information it has learnt from previous trials). In order to ensure the control policy learnt is as generally applicable as possible, the robot is trained on a sequence of randomly generated environments, with each used for only a single trial.

Chapter 5: The Robot Problem considered in chapter 4 involves continuous state-space inputs, but the control actions are selected from a discrete set. Therefore, in this chapter, stochastic hill-climbing AHC methods are examined as a technique for providing real-valued actions. However, as a single continuous function approximator may not be able to learn to represent the optimal policy accurately (especially if it contains discontinuities), a hybrid system called Q-AHC is introduced, which seeks to combine real-valued AHC learning with Q-learning.

Chapter 6: Finally, the conclusions of this thesis are given, along with considerations of possible future research.

Chapter 2

Alternative Q-Learning Update Rules

The standard one-step Q-learning algorithm as introduced by Watkins (1989) was presented in the last chapter. This has been shown to converge (Watkins and Dayan 1992, Jaakkola et al. 1993) for a system operating in fixed Markovian environment. However, these proofs give no indication as to the convergence *rate*. In fact, they require that every state is visited infinitely often, which means that convergence to a particular accuracy could be infinitely slow. In practice, therefore, methods are needed that accelerate the convergence rate of the system so that useful policies can be learnt within a reasonable time.

One method of increasing Q-learning convergence rates is to use *temporal difference* methods with $\lambda > 0$, which were briefly introduced in the last chapter (section 1.3.8). Temporal difference methods allow accelerated learning when no model is available, whilst preserving the on-line updating property of one-step reinforcement learning methods. This on-line feature is explored further in the next chapter, when on-line updating of neural networks is examined.

In the first part of this chapter, the TD-learning algorithm is derived for a general cumulative payoff prediction problem. This results in easier interpretation of a range TD-learning algorithms, and gives a clearer insight into the role played by each of the parameters used by the method. In particular, it shows that the TD-learning parameter λ can be considered constant during trials, in that it does not need to be adjusted in order to implement learning rules such as TD(1/n) (Sutton and Singh 1994) or the original method of combining Q-learning and TD(λ) suggested by Watkins (1989).

A number of methods for updating a Q-function using TD(λ) techniques are then examined, including the standard method introduced by Watkins and also the more recent Q(λ) method introduced by Peng and Williams (1994). In addition, several novel methods are introduced, including Modified Q-Learning and Summation Q-Learning. In the final section of this chapter, the performance of these Q-learning methods is compared empirically on the Race Track problem (Barto et al. 1993), which is one of the largest discrete Markovian control problems so far studied in the reinforcement learning literature.

2.1 General Temporal Difference Learning

In section 1.3.8 the basic concepts behind TD-learning (Sutton 1988) were introduced. In this section, the method is considered in greater detail, by deriving the TD-learning equations for a general prediction problem and examining some of the issues surrounding its application to reinforcement learning tasks. This will be useful when considering the application of this method to Q-learning update rules in the remainder of the chapter.

Consider a problem where the system is trying to learn a sequence of predictions, $P_t, P_{t+1}, \ldots$, such that eventually,

$$P_t = E\left\{\sum_{k=t}^{\infty} \gamma_t^{(k-t)} c_k\right\} \tag{2.1}$$

for all t. The term $\gamma_t^{(n)}$ is defined as follows,

$$\gamma_t^{(n)} = \begin{cases} \prod_{k=t+1}^{t+n} \gamma_k & n > 0 \\ 1 & n = 0 \end{cases} \tag{2.2}$$

where $0 \leq \gamma_t \leq 1$. The right hand part of equation 2.1 represents a *general discounted return*. The discounted return usually used in reinforcement learning problems is the special case where γ_t has a fixed value γ for all t, and $c_t = r_t$.

The prediction P_t can be updated according to,

$$\Delta P_t = \alpha_t \left[\sum_{k=t}^{\infty} \gamma_t^{(k-t)} c_k - P_t\right] \tag{2.3}$$

where α_t is a learning constant and is used so that the prediction will converge towards the expected value as required (equation 2.1). Equation 2.3 can be expanded in terms of the temporal differences between successive predictions in a similar manner to the example given in the introduction (section 1.3.8),

$$\begin{aligned} \Delta P_t &= \alpha_t \left[(c_t + \gamma_{t+1} P_{t+1} - P_t) + \gamma_{t+1}(c_{t+1} + \gamma_{t+2} P_{t+2} - P_{t+1}) + \cdots\right] \\ &= \alpha_t \sum_{k=t}^{\infty} (c_k + \gamma_{k+1} P_{k+1} - P_k) \gamma_t^{(k-t)} \end{aligned} \tag{2.4}$$

Taking things a step further, the predictions P_t could be generated by a function approximator P, which is parametrised by a vector of internal values $\mathbf{w}$. Assuming these values could be updated by a gradient ascent step utilising the vector of gradients $\nabla_{\mathbf{w}} P_t$ (which is made up from the partial derivatives $\partial P_t / \partial w_t$) then,

$$\Delta \mathbf{w}_t = \eta_t \left[\sum_{k=t}^{\infty} (c_k + \gamma_{k+1} P_{k+1} - P_k) \gamma_t^{(k-t)}\right] \nabla_{\mathbf{w}} P_t \tag{2.5}$$

where η_t is a learning rate parameter, which includes α_t. The overall change to the parameters $\mathbf{w}$ is the summation of the individual $\mathbf{w}_t$ over time, which can be rearranged as follows,

$$\begin{aligned} \Delta \mathbf{w} = \sum_{t=0}^{\infty} \Delta \mathbf{w}_t &= \sum_{t=0}^{\infty} \eta_t \left[\sum_{k=t}^{\infty} (c_k + \gamma_{k+1} P_{k+1} - P_k) \gamma_t^{(k-t)}\right] \nabla_{\mathbf{w}} P_t \\ &= \sum_{t=0}^{\infty} (c_t + \gamma_{t+1} P_{t+1} - P_t) \sum_{k=0}^{t} \eta_k \gamma_k^{(t-k)} \nabla_{\mathbf{w}} P_k \end{aligned} \tag{2.6}$$

Thus, a general temporal difference update equation can be extracted which can be used to update the parameters $\mathbf{w}$ at each time step t according to the current TD-error between predictions, i.e.

$$\Delta \mathbf{w}_t = (c_t + \gamma_{t+1} P_{t+1} - P_t) \sum_{k=0}^{t} \eta_k \gamma_k^{(t-k)} \nabla_{\mathbf{w}} P_k \tag{2.7}$$

The summation at the end of the equation has the property that it can be incrementally updated at each time step t as well. If a parameter vector $\mathbf{e}$ is introduced to store these summation terms (one element per element of $\mathbf{w}$), then it can be updated according to,

$$\begin{aligned} \mathbf{e}_t &= \sum_{k=0}^{t} \eta_k \gamma_k^{(t-k)} \nabla_{\mathbf{w}} P_k \\ &= \gamma_t \mathbf{e}_{t-1} + \eta_t \nabla_{\mathbf{w}} P_t \end{aligned} \tag{2.8}$$

and therefore equation 2.7 becomes simply,

$$\Delta \mathbf{w}_t = (c_t + \gamma_{t+1} P_{t+1} - P_t) \mathbf{e}_t \tag{2.9}$$

The values $\mathbf{e}$ are referred to as the *eligibilities* of the parameters $\mathbf{w}$, as they determine how large a change will occur in response to the current TD-error. This mechanism will be used extensively in this thesis for on-line updating of neural networks (see chapter 3).

In fact, when Sutton introduced the TD-learning class of algorithms, he included an extra parameter $0 \leq \lambda \leq 1$ which can be incorporated in the eligibility mechanism and results in the TD(λ) family of algorithms. Thus equation 2.8 becomes,

$$\mathbf{e}_t = (\gamma_t \lambda) \mathbf{e}_{t-1} + \eta_t \nabla_{\mathbf{w}} P_t \tag{2.10}$$

The purpose of the λ term is to adjust the weighting of future temporal difference errors as seen by a particular prediction P_t. This may be helpful if the future errors have a high variance, as a lower value of λ will reduce the effect of these errors, but at the cost of increased bias in the prediction (it will be biased towards the value of predictions occurring closer in time). This is known as a *bias-variance trade-off*, and is important to reinforcement systems which change their policy over time, since a changing policy will result in changing average returns being seen by the system. Thus a future prediction of return P_{t+T} may not have much relevance to the current prediction P_t if T is large, since the sequence of actions that led to that region of the state-space may not occur again as the policy changes.

Equations 2.9 and 2.10 represent the TD-learning update equations for a system predicting a generalised return using a parametrised function approximator. This presentation of the equations differs slightly from the usual forms, which assume a fixed learning rate $\eta_t = \eta$ and thus leave the learning rate at the start of the weight update in equation 2.9. However, the above general derivation allows for the training parameter η_t to be different at each state $\mathbf{x}_t$, which has resulted in the learning rate η_t being incorporated in the eligibility trace. In the Race Track problem presented at the end of this chapter, the learning rate *is* different at each time step, as it is a function of the number of visits that have been made to the current state, and so this difference is important. However, when presenting the Q-function updating rules in section 2.2, a constant η is assumed for clarity.

2.1.1 Truncated Returns

Watkins (1989) showed that using temporal difference updates with a constant λ results in an overall update at each state equivalent to taking the weighted sum of *truncated returns*, e.g. for the general discounted return (equation 2.1), the truncated return is,

$$c_t^{(n)} = c_t + \gamma_t^{(1)} c_{t+1} + \gamma_t^{(2)} c_{t+2} + \cdots + \gamma_t^{(n-1)} c_{t+n-1} + \gamma_t^{(n)} P_{t+n} \tag{2.11}$$

where the prediction P_{t+n} is used to estimate the remainder of the sequence. The overall update produced by a sequence of TD-errors is,

$$\begin{aligned} &(c_t + \gamma_{t+1} P_{t+1} - P_t) + \gamma_{t+1}\lambda(c_{t+1} + \gamma_{t+2} P_{t+2} - P_{t+1}) + \cdots \\ &\quad = (1-\lambda)\left[(c_t + \gamma_t^{(1)} P_{t+1}) + \lambda(c_t + \gamma_t^{(1)} c_{t+1} + \gamma_t^{(2)} P_{t+2}) + \cdots\right] - P_t \\ &\quad \equiv (1-\lambda)\left[c_t^{(1)} + \lambda c_t^{(2)} + \cdots\right] - P_t \end{aligned} \tag{2.12}$$

which is a weighted sum of truncated returns. This result helps clarify the use of the λ parameter as a method for adjusting the importance of estimates of the return made over longer sequences. It is essential to note, however, that it is only applicable where a constant λ value is used.[1] Despite this, it is common when using TD methods to consider varying the value of λ at each time step; for instance for standard Q-learning (see section 2.2.1) or methods such as TD(1/n) (section 2.1.2), even though this renders the interpretation of the updates as weighted sums of truncated returns invalid.

However, equation 2.12 does hold for arbitrary values of γ_t and thus it will be shown over the remainder of this chapter that using a constant λ value is not a problem; it is the value of γ_t that should be adjusted and not λ at all. Although this has no practical effect on the type of updates used, it does allow a clearer understanding of how they are derived and what the parameters and updates represent.

Finite Trial Length

The summation of equation 2.12 assumes time $t \to \infty$, but most reinforcement systems are stopped after reaching a goal state and hence only perform the summation for a finite number of steps. This does not effect the interpretation of the summation, which turns out to be equivalent to remaining in the final state forever, receiving immediate payoffs c_t of zero. For example, if the system reaches the goal at time step $t+1$ then,

$$\begin{aligned} (c_t + \gamma_{t+1} P_{t+1} - P_t) &= (1-\lambda)\left[(c_t + \gamma_t^{(1)} P_{t+1}) + \lambda(c_t + \gamma_t^{(1)} 0 + \gamma_t^{(2)} P_{t+1}) + \cdots\right] - P_t \\ &= (1-\lambda)\left[c_t^{(1)} + \lambda c_t^{(2)} + \cdots\right] - P_t \end{aligned} \tag{2.13}$$

where it should be remembered that P_{t+n} for $n > 1$ is equal to P_{t+1}, as the system is assumed to remain in the final state forever.

2.1.2 Value Function Updates

Over the previous sections, a temporal difference algorithm for a general cumulative payoff prediction problem has been derived and some of its properties examined. However, the term TD(λ) is generally associated with the specific case of learning a value function, where

[1] If $\lambda = 1$ then the sequence of TD-errors is equivalent to $\sum_{k=t}^{\infty} \gamma_t^{(k-t)} c_k - P_t \equiv c_t^{(\infty)} - P_t$.

$P_t = V_t$, $c_t = r_t$, $\gamma_t = \gamma$, and $V = V(\mathbf{x})$ is the value function prediction of returns. The parameters of the function representing V can therefore be updated using a TD-algorithm,

$$\begin{aligned}\Delta \mathbf{w}_t &= (r_t + \gamma V_{t+1} - V_t)\mathbf{e}_t & (2.14)\\ \mathbf{e}_t &= (\gamma\lambda)\mathbf{e}_{t-1} + \eta_t \nabla_{\mathbf{w}} V_t & (2.15)\end{aligned}$$

The above allows the value function prediction V_t to be updated at each time step using the TD-error and eligibility trace mechanism.

Convergence proofs (Sutton 1988, Dayan 1992, Jaakkola et al. 1993) have shown that this form of temporal difference algorithm will guarantee the predictions, V, to converge to the expected return for any value of $0 \leq \lambda \leq 1$.[2] This is under the conditions that the predictions are made in a Markovian environment with a fixed policy and transition probabilities, and the function approximator used to store the predictions is simply a linear weighting of the input vector, i.e. $\mathbf{w}.\mathbf{x}$. However, these proofs do not provide indications as to the convergence rate (the convergence is asymptotic, so could be infinitely slow to reach the required accuracy) and so the choice of values for η_t and λ must be made with care.

The TD(1/n) Algorithm

TD(1/n) is a method suggested in Sutton and Singh (1994) when considering the optimum values for the parameters η_t and λ. The paper concentrates on predicting the value function, V, in fixed Markovian domains where the system is not trying to learn a policy (thus the sequences of states and payoffs seen are entirely controlled by the state transition probabilities).

In this environment, a prediction will converge to the expected value if the returns it sees are averaged over all trials. This can be achieved by keeping count of how many times the state has been visited and then updating its prediction according to,

$$\Delta V_t = \frac{1}{n_t}\left[r_t + \gamma V_{t+1} - V_t\right] \qquad (2.16)$$

where n_t is the number of times the state has been visited, including the current visit at time t. By then considering the change in V_{t+1}, a temporal difference algorithm can be constructed where,

$$\begin{aligned}\Delta \mathbf{w}_t &= \left[r_t + \gamma V_{t+1} - V_t\right]\mathbf{e}_t & (2.17)\\ \mathbf{e}_t &= (\gamma\frac{1}{n_t})\mathbf{w}_t + \frac{1}{n_t}\nabla_{\mathbf{w}} V_t & (2.18)\end{aligned}$$

This algorithm only makes sense for a lookup table representation, as $\mathbf{w}_t = V$ in the above equation. Hence the value of $\partial V_t/\partial w_t$ is 1 for the current state and zero for all others. By comparing the above algorithm with the general TD equation 2.9, it can be seen that $\eta_t \equiv 1/n_t$. Also, comparison with the eligibility update equation 2.10 suggests letting $\gamma_t = \gamma$ and therefore $\lambda = 1/n_t$. The latter gives rise to the name of the algorithm as TD(1/n). However, this means that λ is not a constant and thus the truncated return interpretation (equation 2.12) cannot be used.

[2] In fact, they will converge to the expected return only if the learning rate η_t is reduced over time. If η_t is not reduced, then the *expected value* of the prediction will converge to this value.

However, an alternative way of looking at the above algorithm is to let $\gamma_t = \gamma / n_t$ and $\lambda = 1$. Then letting $P_t = V_t$, it can be seen that,

$$r_t + \gamma V_{t+1} \equiv r_t + \frac{n_{t+1} - 1}{n_{t+1}} \gamma V_{t+1} + \gamma_{t+1} P_{t+1} \tag{2.19}$$

and thus $c_t = r_t + (n_{t+1} - 1)\gamma V_{t+1}/n_{t+1}$. Therefore the interpretation of section 2.1.1 still applies, with the sequence of TD-errors equal to $c_t^{(\infty)} - P_t$ (because $\lambda = 1$). This leaves the question of what the return $c_t^{(\infty)}$ for this form of updates actually represents,

$$\begin{aligned} c_t^{(\infty)} &= c_t + \gamma_{t+1} c_{t+1} + \gamma_{t+1}\gamma_{t+2} c_{t+2} + \cdots \\ &= r_t + \frac{n_{t+1} - 1}{n_{t+1}} \gamma V_{t+1} + \frac{1}{n_{t+1}} \gamma r_{t+1} + \frac{1}{n_{t+1}} \frac{n_{t+2} - 1}{n_{t+2}} \gamma^2 V_{t+2} + \cdots \\ &= r_t + \frac{\gamma}{n_{t+1}} \left[(n_{t+1} - 1) V_{t+1} + r_{t+1} + \frac{\gamma}{n_{t+2}} \left[(n_{t+2} - 1) V_{t+2} + \cdots \right] \right] \end{aligned} \tag{2.20}$$

This is an expansion of the expected discounted return and shows how the current value function predictions, V, will gain importance as the number of state visits, n, go up.

The overall aim of this section has been to demonstrate as an example that the TD(1/n) algorithm can be viewed using the general TD framework described in section 2.1. By doing this, it has been shown that λ can be considered to remain constant with a value of 1 and so is effectively not used in this algorithm. This helps clarify the role of λ as the weighting parameter for the summation of truncated returns. For the fixed environment considered for TD(1/n), setting λ less than 1 would not useful, but in problems where the policy, and thus the returns seen, change, this helps avoid early biasing of predictions and therefore action choices.

2.2 Combining Q-Learning and TD(λ)

One-step Q-learning makes minimal use of the information received by the system, by only updating a single prediction for a single state-action pair at each time-step. TD(λ) methods offer a way of allowing multiple predictions to be updated at each step and hence speeding up convergence.

Firstly, consider the one-step Q-learning algorithm applied to a general function approximator, such that each prediction $Q(\mathbf{x}_t, a_t)$ is made by a function using a set of internal parameters $\mathbf{w}_t$ to make the prediction. In this case, equation 1.14 is applied to update the parameters according to,

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t \right] \nabla_{\mathbf{w}} Q_t \tag{2.21}$$

where Q_t is used as a notational shorthand for $Q(\mathbf{x}_t, a_t)$ and η is a constant learning rate parameter. $\nabla_{\mathbf{w}} Q_t$ is a vector of the partial derivatives $\partial Q_t / \partial w_t$, which will be referred to as the *output gradients*.

2.2.1 Standard Q-Learning

In order to speed up learning, Watkins (1989) suggests combining Q-learning with temporal difference methods using $\lambda > 0$. In this formulation, the current update error is used to adjust not only the current estimate, Q_t, but also that of previous states, by keeping a weighted sum of earlier output gradients,

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t \right] \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{\mathbf{w}} Q_k \tag{2.22}$$

The one-step Q-learning equation is therefore a special instance of this equation where $\lambda = 0$. To distinguish the algorithm represented by equation 2.22 from the methods presented over the next sections, this will be referred to as *standard* Q-learning.

An important point about equation 2.22 is that it is not a correct TD-learning algorithm unless the *greedy* policy is followed at all times, i.e. the temporal difference errors will not add up correctly,

$$\sum_{k=t}^{\infty} \gamma^{k-t} \left[r_k + \gamma \max_{a \in A} Q_{k+1} - Q_k \right] \neq \sum_{k=t}^{\infty} \gamma^{k-t} r_k - Q_t \tag{2.23}$$

unless the action corresponding to $\max_{a \in A} Q(\mathbf{x}_t, a_t)$ is performed at every time step. Watkins recognised this and suggested setting $\lambda = 0$ whenever non-greedy actions are performed (as is necessary for exploration; see section 1.3.7).

However, by comparing the standard Q-learning equations with the general temporal difference update equations presented in section 2.1, this update algorithm will be seen to follow directly by substitution into equations 2.9 and 2.10, with the proviso that it is γ_t that is set to zero and not λ as suggested by Watkins.

This can be seen by letting $P_t = Q_t$. Then the values of c_t and γ_t depend on whether $P_{t+1} \equiv \max_{a \in A} Q_t$ or not, which is down to whether the system performs the greedy action or not. If the greedy action is performed then $c_t = r_t$ and $\gamma_t = \gamma$. However, if it is not, then the TD-error is equivalent to,

$$r_t + \gamma \max_{a \in A} Q_{t+1} + 0.P_{t+1} - P_t \tag{2.24}$$

which implies that $c_t = r_t + \gamma \max_{a \in A} Q_{t+1}$ and $\gamma_{t+1} = 0$. Using these values has exactly the same effect as zeroing λ, but means that the sum of truncated returns interpretation of equation 2.12 can be seen to still apply. In fact, by considering equation 2.13 with these values of c_t and γ_t, it can be seen that the effect of the zero γ_{t+1} is the same as if the trial has ended in the state $\mathbf{x}_{t+1}$. Clearly, this will introduce bias into the returns seen by the system and thus in the next section an alternative Q-learning update rule is presented which avoids this problem.

2.2.2 Modified Q-Learning

The question is whether $\max_{a \in A} Q(\mathbf{x}, a)$ really provides the best estimate of the return of the state $\mathbf{x}$. In the early stages of learning, the Q-function values of actions that have not been explored is likely to be completely wrong, and even in the latter stages, the maximum value is more likely to be an over-estimation of the true return available (as argued in Thrun and Schwartz (1993)). Further, the standard update rule for Q-learning combined with TD(λ) methods requires γ_t to be zero for every step that a non-greedy action is taken. As from the above arguments the greedy action could in fact be incorrect (especially in the early stages of learning), zeroing the effect of subsequent predictions on those prior to a non-greedy action is likely to be more of a hindrance than a help in converging on the required predictions. Furthermore, as the system converges to a solution, greedy actions will be used more to exploit the policy learnt by the system, so the greedy returns will be seen anyway. Therefore, a new update algorithm is suggested here, based more strongly on TD(λ) for value function updates (section 2.14), called Modified Q-Learning.[3]

The proposed alternative update rule is,

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma Q_{t+1} - Q_t \right] \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \nabla_{\mathbf{w}} Q_k \tag{2.25}$$

This differs from standard Q-learning in the use of the Q_{t+1} associated with the action selected, rather than the greedy $\max_{a \in A} Q_{t+1}$ used in Q-learning.[4] This ensures that the temporal difference errors will add up correctly, regardless of whether greedy actions are taken or not, without introducing zero γ_t terms. This can be seen from the general TD-learning equations derived in section 2.1, as $P_t = Q_t$, $c_t = r_t$ and $\gamma_t = \gamma$ at all time steps. If greedy actions *are* taken, however, then this equation is exactly equivalent to standard Q-learning, and so, in the limit when exploration has ceased and the greedy policy is being followed, the updates will be the same as for standard Q-learning (equation 2.22).

Modified Q-Learning therefore samples from the distribution of possible future returns given the current exploration policy, rather than just the greedy policy as for normal Q-learning. Therefore, the Q-function will converge to,

$$Q(\mathbf{x}_t, a_t) \leftarrow E \left\{ r_t + \gamma \sum_{a \in A} P(a|\mathbf{x}_{t+1}) Q(\mathbf{x}_{t+1}, a) \right\} \tag{2.26}$$

which is the expected return given the probabilities, $P(a|\mathbf{x}_t)$, of actions being selected. Consequently, at any point during training, the Q-function should give an estimation of the expected returns that are available for the current exploration policy. As it is normal to reduce the amount of exploration as training proceeds, eventually the greedy action will be taken at each step, and so the Q-function should converge to the optimal values.

Can this algorithm be *guaranteed* to converge in a Markovian environment, as TD(λ)[5] and one-step Q-learning can? The proof of Jaakkola et al. (1993) relies on the max operator, which has been discarded in Modified Q-Learning. On the other hand, at each step, the value seen depends on the transition probability multiplied by the probability of selecting the next action i.e. $P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t) P(a_{t+1}|\mathbf{x}_{t+1})$. The overall effect is equivalent to a transition probability $P(\mathbf{x}_{t+1}|\mathbf{x}_t)$ as seen by a TD(λ) process, which is known to converge if these values are *constant* (Dayan 1992, Jaakkola et al. 1993). So, clearly, if the policy, and thus $P(a|\mathbf{x})$, is constant then Modified Q-Learning will converge to the expected return given the current policy. Any restrictions that exist for convergence to be guaranteed when the policy is changing are related to the way in which the action probabilities $P(a|\mathbf{x})$ change over time. Whether the proofs based on stochastic approximation theory (Jaakkola et al. 1993, Tsitsiklis 1994) can be modified to provide these bounds is an open question.

[3] Though Rich Sutton suggests *SARSA*, as you need to know State-Action-Reward-State-Action before performing an update (Singh and Sutton 1994).

[4] Wilson (1994) noted the similarities between Q-learning and the *bucket-brigade* classifier system (Holland 1986). Using this interpretation, the bucket-brigade algorithm is equivalent a TD(0) form of Modified Q-Learning.

[5] In the specific form defined in section 2.1.2.

2.2.3 Summation Q-Learning

Considering equation 2.26, it is clear that another novel update would be to use expected return given the current action probabilities i.e. $\sum_{a\in A} P(a|\mathbf{x}_{t+1})Q(\mathbf{x}_{t+1}, a)$, instead of the maximum value, $\max_{a\in A} Q(\mathbf{x}_{t+1}, a)$, as in the standard update rule, or the selected action value, $Q(\mathbf{x}_{t+1}, a_{t+1})$, as used in Modified Q-Learning. This rule, which will be called Summation Q-Learning, has attractions as it takes the probability of selecting actions into account directly, rather than indirectly as for Modified Q-Learning or not at all for standard Q-learning.

However, simply summing the temporal difference errors over time will lead to similar problems as for standard Q-learning, in that they will not add up correctly. The solution can be found by considering the general TD-learning algorithm of section 2.1 and what the predictions P_t and P_{t+1} actually represent at each time step. If a_{t+1} is the action selected to be performed at time step $t+1$ and $P_t = Q_t$, then the temporal difference error at each time step will be equal to,

$$r_t + \gamma \sum_{a\neq a_{t+1}} P(a|\mathbf{x}_{t+1})Q_{t+1} + \gamma P(a_{t+1}|\mathbf{x}_{t+1})P_{t+1} - P_t \tag{2.27}$$

Thus, it can be seen that $c_t = r_t + \sum_{a\neq a_{t+1}} P(a|\mathbf{x}_{t+1})Q_{t+1}$ and $\gamma_t = \gamma P(a_{t+1}|\mathbf{x}_{t+1})$. In other words, in order for the temporal differences sum correctly, it is necessary to include the probability of the selected action, $P(a_{t+1}, \mathbf{x}_{t+1})$, into the eligibility trace along with γ, leading to an overall update algorithm of,

$$\Delta\mathbf{w}_t = \eta\left[r_t + \gamma \sum_{a\in A} P(a|\mathbf{x}_{t+1})Q_{t+1} - Q_t\right]\mathbf{e}_t \tag{2.28}$$

$$\mathbf{e}_t = (\gamma\lambda)P(a_t|\mathbf{x}_t)\mathbf{e}_{t-1} + \nabla_{\mathbf{w}}Q_t \tag{2.29}$$

However, experiments have shown that this update rule actually performs much worse than standard Q-learning and Modified Q-Learning, despite appearing to be the most theoretically sound. This is due to the fact that the probability term in the eligibility trace actually results in reducing the size of updates to earlier states. Also, the summation of action values weighted by probabilities ends up giving too much weighting to poor estimates and thus suffers from the problem of bias that occurs with standard Q-learning.

2.2.4 Q(λ)

Peng and Williams (1994) presented another method of combining Q-learning and TD(λ), called Q(λ). This is based on performing a standard one-step Q-learning update to improve the current prediction Q_t and then using the temporal differences between successive *greedy* predictions to update it from there on, regardless of whether greedy actions are performed or not. This means that the eligibilities do not need to be zeroed, but requires that two different error terms be calculated at each step. Peng presented the algorithm for discrete state-space systems, whilst here it is extended for use with a general function approximator.

At each time step, an update is made according to the one-step Q-learning equation 2.21. Then a *second* update is made using,

$$\Delta\mathbf{w}_t = \eta\left[r_t + \gamma \max_{a\in A} Q_{t+1} - \max_{a\in A} Q_t\right]\sum_{k=0}^{t-1}(\gamma\lambda)^{t-k}\nabla_{\mathbf{w}}Q_k \tag{2.30}$$

Note the summation is only up to step $t-1$. If a continuous state-space function approximator is being updated, both changes will affect the same weights and so result in an overall update of,

$$\Delta \mathbf{w}_t = \eta \left[(r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t) \nabla_{\mathbf{w}} Q_t + (r_t + \gamma \max_{a \in A} Q_{t+1} - \max_{a \in A} Q_t) \mathbf{e}_{t-1} \right] \tag{2.31}$$

where,

$$\mathbf{e}_t = (\lambda\gamma)\mathbf{e}_{t-1} + \nabla_{\mathbf{w}} Q_t \tag{2.32}$$

This algorithm does not fit the general TD-learning framework presented in section 2.1, because a prediction $P_t = Q_t$ does not appear in equation 2.30 unless it corresponds with the greedy action. However, the algorithm can still be interpreted as a weighted sum of truncated returns,

$$\begin{aligned} & (r_t + \gamma \max_{a \in A} Q_{t+1} - Q_t) + \gamma\lambda(r_{t+1} + \gamma \max_{a \in A} Q_{t+2} - \max_{a \in A} Q_{t+1}) + \cdots \\ = \; & (1-\lambda) \left[(r_t + \gamma \max_{a \in A} Q_{t+1}) + \lambda(r_t + \gamma r_{t+1} + \gamma^2 \max_{a \in A} Q_{t+2}) + \cdots \right] - Q_t \\ = \; & (1-\lambda) \left[r_t^{(1)} + \lambda r_t^{(2)} + \cdots \right] - Q_t \end{aligned} \tag{2.33}$$

where $r_t^{(n)}$ is therefore equal to,

$$r_t^{(n)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_{a \in A} Q_{t+n} \tag{2.34}$$

The truncated returns summed by Modified Q-Learning can be found by substitution into equation 2.11 and turn out to be equal to,

$$r_t^{(n)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n Q_{t+n} \tag{2.35}$$

Therefore, Q(λ) and Modified Q-Learning are very similar, in that they both sum the truncated return estimates of cumulative discounted payoffs, regardless of whether greedy or non-greedy actions are performed (i.e. the return seen is for the current policy). The only difference is with the value used to estimate the remainder of the return. Modified Q-Learning uses the Q_t estimates, which should be good estimates of this return, as that is what they are being updated to represent. Q(λ) uses the biased greedy estimates, $\max_{a \in A} Q_t$, which are estimates of what the predictions should eventually represent. The difference is subtle and in the experiments presented in section 2.3.2 the difference in performance between the algorithms is also small.

2.2.5 Alternative Summation Update Rule

The thinking behind Q(λ) suggests another possibility. In section 2.2.3, it was suggested that a summation of the Q-function values weighted by the probability of their being chosen might provide a good update rule. However, it was then shown that this requires that the eligibility traces fade off in proportion to the probability $P(a_t|\mathbf{x}_t)$ of the chosen action at each step, which is equivalent to using low λ values. However, this problem can be avoided by using two updates at each time step as in Q(λ) i.e. by performing an immediate update of,

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \sum_{a \in A} P(a|\mathbf{x}_{t+1}) Q_{t+1} - Q_t \right] \nabla_{\mathbf{w}} Q_k \tag{2.36}$$

and updating the predictions at all previously visited states using,

$$\Delta \mathbf{w}_t = \eta \left[r_t + \gamma \sum_{a \in A} P(a|\mathbf{x}_{t+1}) Q_{t+1} - \sum_{a \in A} P(a|\mathbf{x}_t) Q_t \right] \sum_{k=0}^{t-1} (\gamma\lambda)^{t-k} \nabla_{\mathbf{w}} Q_k \qquad (2.37)$$

Again, these two updates should be summed together to give the overall update $\Delta \mathbf{w}_t$ as for the Q(λ) update in equation 2.31. The resulting new update rule will be referred to as Summation Q(λ).

This means that the eligibility trace fades in proportion to ($\lambda\gamma$) as for equation 2.32, rather than discounting it further using the action probability, $P(a_t|\mathbf{x}_t)$, as for Summation Q-Learning (equation 2.29). However, it does result in the most computationally expensive update of those presented here, as there is the requirement to calculate both the summation across actions and the two TD-error terms at each time step.

2.2.6 Theoretically Unsound Update Rules

The previous sections have presented a variety of methods for combining TD(λ) methods and Q-learning in an attempt to produce faster convergence when learning a Q-function. It was discussed in section 2.2.1 that, when performing standard Q-learning updates, not zeroing the eligibilities when non-greedy actions are performed means that the temporal difference errors do not add up correctly. To avoid this it is necessary to zero γ_t when non-greedy actions are performed.

However, the question is whether temporal difference errors failing to sum correctly is actually a problem. If it is not then standard Q-learning with non-zeroed γ_t, and the summation update rule of section 2.2.3 ignoring the action probability, $P(a_t|\mathbf{x}_t)$, in the eligibility trace, become viable update rules.[6] To distinguish these two algorithms from the others, they will be referred to as Fixed Q-learning and Fixed Summation Q-Learning respectively.

It is certainly possible to construct conditions under which using these update rules would result in undesirable, and perhaps even unstable, update sequences. For example, consider a system learning using Fixed Q-learning updates in the situation where in each state the maximum Q-function prediction is equal to its optimum value of Q^*. All other actions predict a value of q where $q < Q^*$. Also $\lambda = 1$ and there are no payoffs until the end of the trial ($r_t = 0$). Each time the system performs the greedy action, the TD-error will be zero and so no changes in prediction will occur. However, the eligibilities will be not be zero, but equal to the summation of error gradients. If a non-greedy action is performed, the TD-error will be $\gamma Q^* - q$ i.e. an indication that the last prediction was too low. This will update the action value for the state-action pair that has a value of q, but, due to the non-zero eligibilities, the greedy predictions at the previously visited states will change in response too. Therefore, these states will now predict values slightly over Q^*, which is not what is required at all. This will happen each time a non-greedy action is performed and so these predictions will continue to grow as a result.

This effect may be kept in check, as the state-action pairs which contain an over-prediction could be corrected back in a later trial. The danger, however, is that these unwanted changes could lead to instability.

Despite this, in the experiments presented later in this chapter, it is found that these types of update rule *can* perform better than their more theoretically sound counterparts

[6] This latter method was also suggested in Sathiya Keerthi and Ravindran (1994) when discussing Modified Q-Learning as originally presented in Rummery and Niranjan (1994).

(for instance, Q-learning with fixed γ_t outperforms standard Q-learning with γ_t zeroed for non-greedy action choices). This is because the effect that they overcome — unnecessarily cautious masking of the effects of exploratory actions and thus increased bias — is more important than the occasional poor updates they introduce.

2.3 The Race Track Problem

Having introduced a variety of update rules for combining Q-learning with TD(λ) methods, here results are presented of experiments to provide empirical evidence about the relative merits of the different update rules.

The Race Track problem used here is exactly as presented by Barto et al. (1993) in their report on Real-Time Dynamic Programming, which included a comparison of RTDP with one-step Q-learning. This problem was chosen as it is one of the largest discrete state-space control problems thus far considered in the reinforcement learning literature. Hence, given the desire to investigate methods suitable for larger problems, this task provides a good test to compare the relative performance of the different update algorithms.

2.3.1 The Environment

The 'race tracks' are laid out on 2D grids as shown in Figs. 2.1 and 2.4. Hence, each track is a discrete Markovian environment, where the aim is for a robot to guide itself from the start line to the finish line in the least number of steps possible. The robot state is defined in terms of (p_x, p_y, v_x, v_y) i.e. its position and velocity (all integer values) in the x and y planes. At each step, the robot can select an acceleration (a_x, a_y) choosing from $\{-1, 0, +1\}$ in both axes. It therefore has 9 possible combinations and thus actions to choose from. However, there is a 0.1 probability that the acceleration it selects is ignored and $(0, 0)$ is used instead.

The robot receives a payoff of -1 for each step it makes[7] and thus the only reward for reaching the goal is that no more costs are incurred. If the robot moves off the track, it is simply placed at a random point on the starting line and the trial continues. The two tracks and the learning parameters are exactly as used by Barto et al. This includes a learning rate that reduces with the number of visits to a state-action pair and a Boltzmann exploration strategy with exponentially decreasing temperature parameter (see Appendix A for details).

A lookup table representation was used to store the Q-function values. This means that the parameter vector $\mathbf{w}_t$ used in the update algorithms is simply a vector of all action values, $Q(\mathbf{x}, a)$, with one entry for each state-action pair. Hence $\partial Q_t / \partial w_t$ is 1 for the Q-function entry corresponding to current state-action pair and zero for all others. The eligibility traces are implemented as a buffer of the most recently visited states, which maintains only states with eligibilities greater than a certain threshold (in this work, 0.1 was used). The buffer technique was used as to implement the eligibilities as one per state-action pair, and then update them all at each time step, would be impractical (this is difficulty of using a lookup table representation).

Real-Time Dynamic Programming (RTDP) was also implemented to provide a performance comparison. In this method, the value function $V(\mathbf{x}_t)$ is learnt and is updated

[7] Barto et al. (1993) used a $+1$ costs and selected actions to *minimise* the expected future cost. Here negative payoffs are used to achieve the same effect.

when a state is visited by performing the following dynamic programming update,

$$V(\mathbf{x}_t) = \max_{a \in A} \sum_{\mathbf{x} \in \mathbf{X}} P(\mathbf{x}|a, \mathbf{x}_t) \left[r_t + \gamma V(\mathbf{x}) \right] \tag{2.38}$$

where $P(\mathbf{x}|a, \mathbf{x}_t)$ is the probability of reaching a state $\mathbf{x}$ from $\mathbf{x}_t$ given that action a is performed. It is therefore necessary to have access to a full world model[8] to perform the RTDP updates.

2.3.2 Results

The two race tracks used for testing are shown at the top of Figs. 2.1 and 2.4. The training curves for one-step Q-learning (equation 2.21) and RTDP (equation 2.38) are also shown, which represent the two extremes in convergence rates of the methods studied here. Each method was repeated 25 times on the same problem with different random number seeds and the results averaged to give the training curves reproduced here. The results are shown in terms of the average number of steps required to reach the finish line per epoch, where an epoch consisted of 20 trials. The lines representing one standard deviation either side of the curves are included for reference and to show that the problem has been reproduced exactly as defined in Barto et al. (1993).

In this section, the results of applying the different update rules discussed in this chapter on the Race Track problem are presented. The methods under test are:

- **Standard Q-learning** (equation 2.22) with the eligibilities zeroed whenever a non-greedy action is performed.
- **Modified Q-learning** (equation 2.25).
- **Summation Q-learning** (equation 2.28).
- **Q(λ)** (equation 2.31).
- **Summation Q(λ)** (equations 2.36 and 2.37).
- **Fixed Standard Q-learning** where the eligibilities are not zeroed (section 2.2.6).
- **Fixed Summation Q-learning** where $P(a|\mathbf{x})$ is not used in the eligibilities (section 2.2.6).

Again, each training curve is the average over 25 runs of the algorithm with differing initial random seeds.

The first results are shown in Fig. 2.2 and 2.3 for the small track using two different values of λ. As can be seen, the performance of all the methods improves with the higher value of λ, which shows that in this problem the long term payoffs are important to solving the task (which is reasonable as the overall goal of the system is to reach the finish line and so stop accumulating negative payoffs). The performance of standard Q-learning is better than the simple one-step updates, but is actually worse than the Fixed Q-learning using the constant $\gamma_t = \gamma$. The Summation Q-Learning method performs even worse to start with, but catches up the standard Q-learning method by the end of the run for both values of λ.

[8] Or, in the case of Adaptive-RTDP, to learn one (Barto et al. 1993).

On the lower graphs, the performance of Q(λ), Fixed Summation, Summation Q(λ), and Modified Q-Learning can be seen to be almost identical at both values of λ. In fact, at a λ value of 0.75, these methods manage to learn at almost the same rate as Real-Time Dynamic Programming, even though RTDP has the advantage of a full model of the environment transition probabilities.

The second set of graphs in Fig. 2.5 and 2.6 show the performance of the methods on the large race track. The ranking of the different update rules the same as on the small track. Q(λ) performs noticeably worse than the other methods on the lower graph of Fig. 2.5 when $\lambda = 0.5$. At the higher λ value, virtually all of the methods appear to be able to converge at the same kind of rate as can be achieved using RTDP updates. The one exception is Summation Q-Learning, which does barely better than when using $\lambda = 0.5$.

Finally, the performance of the different choices of TD-error are considered by using the one-step TD(0) versions of the algorithms. There are only 3 different algorithms tested; standard Q-learning, Modified Q-Learning and Summation Q-Learning. The other algorithms differ from these algorithms only in the way that the eligibility traces are handled, but when $\lambda = 0$ these differences disappear. Thus Q(λ) and Fixed Q-learning become equivalent to standard Q-learning, whilst Summation Q(λ) and Fixed Summation Q-Learning become equivalent to Summation Q-Learning.

Fig. 2.7 shows the relative performance of the 3 different choices of TD-error. Modified Q-Learning updates perform the best, especially on the large race track. Summation Q-Learning starts by improving its policy at the same rate as standard Q-learning, but gradually pulls ahead towards the end of the runs, and in fact more or less catches up Modified Q-Learning for the small race track test. So, it appears from this task that the less biased the TD-error used, the better the performance of the update method. In other words, it is better to pass back genuine information as it is accumulated over the course of the trial, rather than rely on intermediate predictions that may not be based on any information at all (i.e. they may be simply initial settings).

2.3.3 Discussion of Results

The results consistently demonstrated the Modified, Summation Q(λ) and Fixed Summation Q-Learning rules provided the fastest convergence of the update rules considered. Q(λ) was equally fast on most of the problems, apart from on the large track when $\lambda = 0.5$ was used (Fig. 2.5). Of the other methods, Fixed Q-learning was the best, followed by standard Q-learning and finally Summation Q-Learning.

Of the fastest methods, Modified Q-Learning has the advantage of being the least computationally expensive and easiest to implement. Summation Q(λ) is at the other end of the scale in terms of computational expense, requiring the calculation of two TD-error terms and a summation across all actions to be performed at every time step. So although it performs as well as Modified Q-Learning, it does not offer any advantages. A similar argument applies to Q(λ) and Fixed Summation Q-Learning. In addition, the two rules with 'fixed' γ_t fall into the category of being theoretically unsound, and so whilst they work well on this problem, there could be situations in which they could lead to unstable updates. Overall, therefore, Modified Q-Learning offers the most efficient Q-function update rule on the basis of these experiments.

Real Time Dynamic Programming provided faster convergence than any of the Q-learning methods. However, RTDP has the advantage of a world model, which it requires

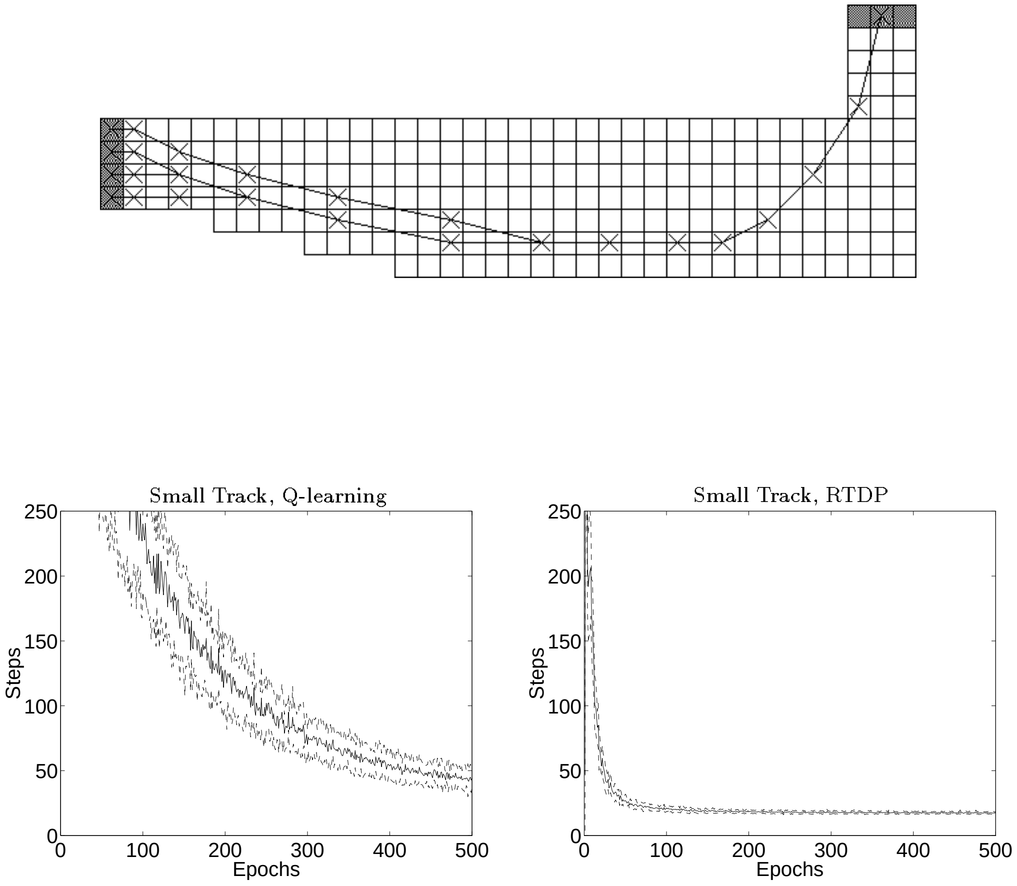
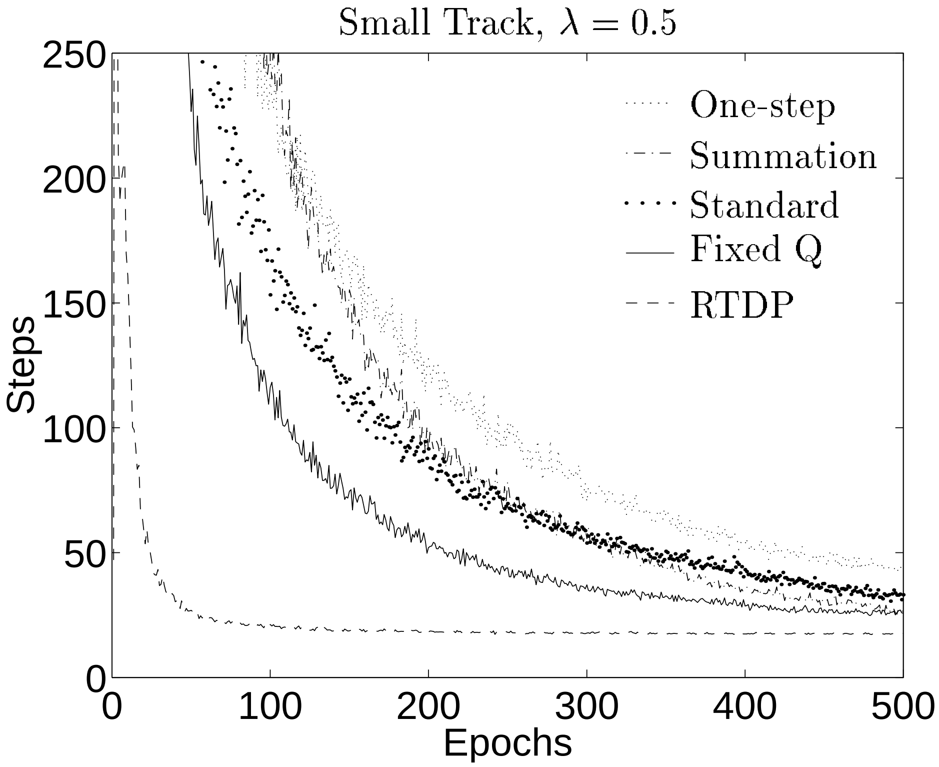


Figure 2.1: *Top:* The small race track used for testing. The start line is on the left and the finish line is on the right (the shaded squares). The lines show a typical trajectory achieved by the robot after training. *Bottom:* Graphs for one-step Q-learning and Real-Time Dynamic Programming. The dashed lines mark one standard deviation either side of the mean as measured over 25 runs.



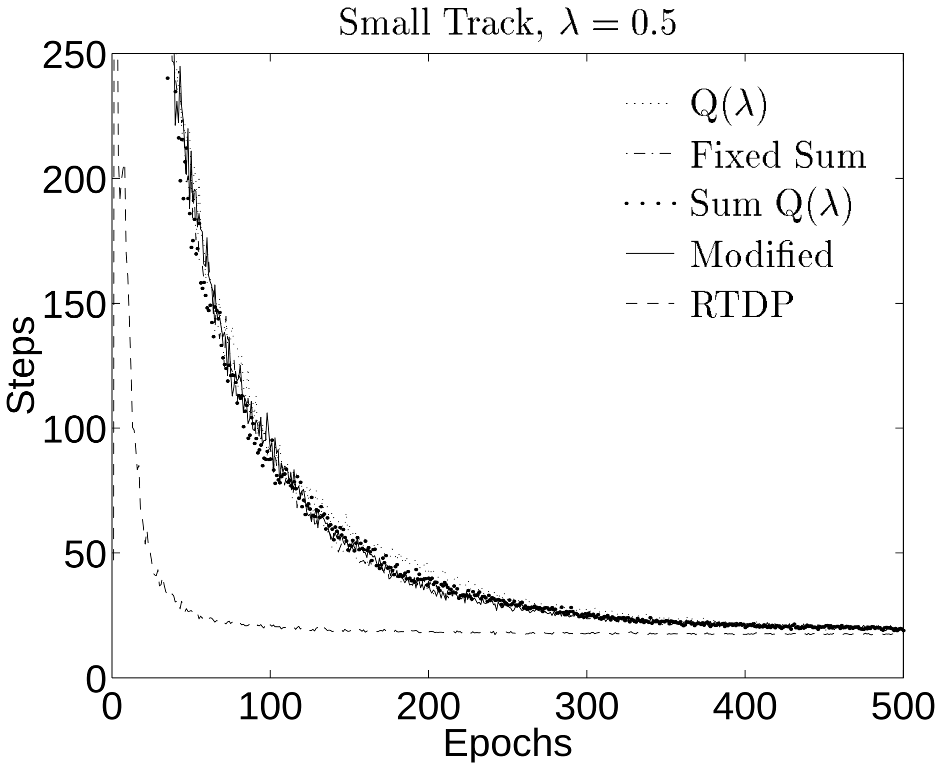
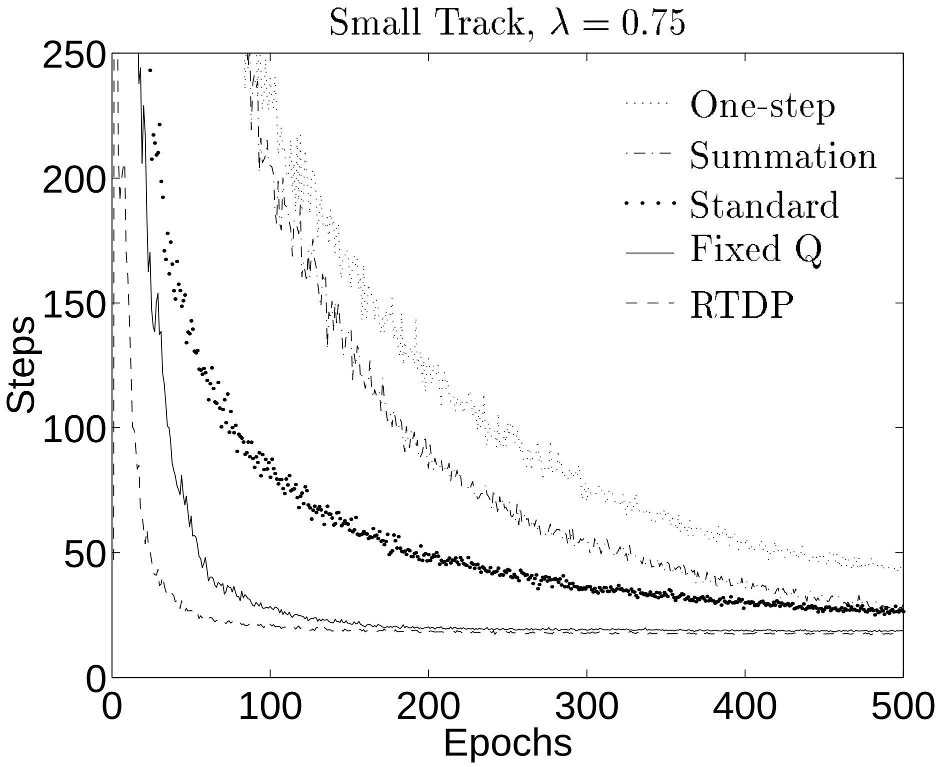


Figure 2.2: Small race track tests for $\lambda = 0.5$. Graphs show the relative performance of the different update rules across epochs. Each epoch consists of 20 trials and each curve is the average over 25 runs. The one-step Q-learning and RTDP lines are included for reference.



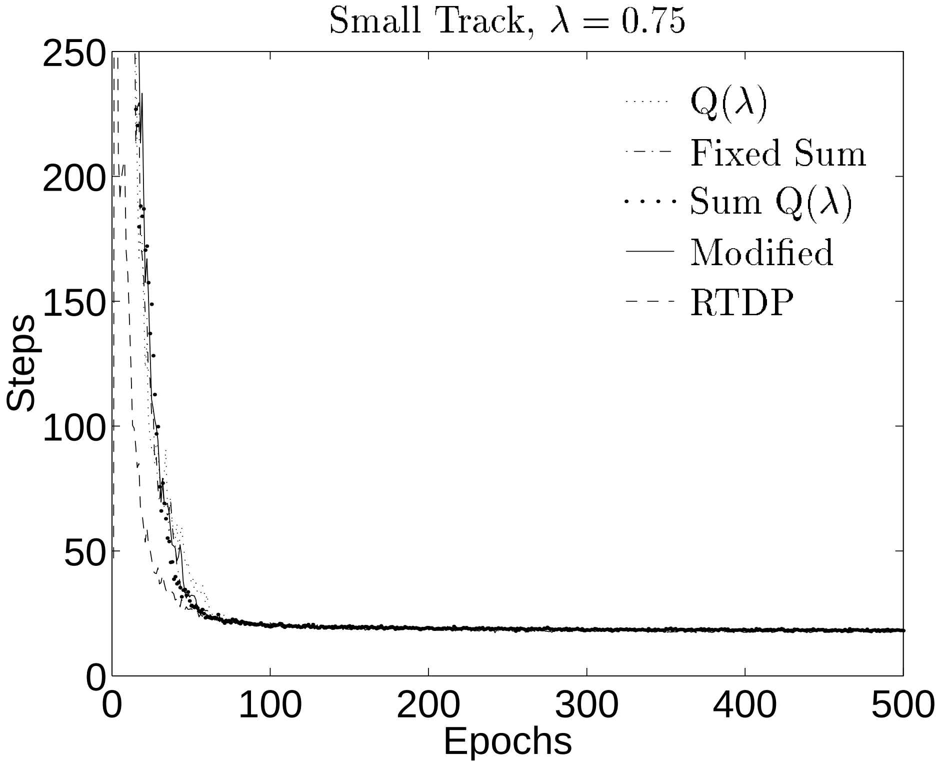


Figure 2.3: Small race track tests for $\lambda = 0.75$. Graphs show the relative performance of the different update rules across epochs. Each epoch consists of 20 trials and each curve is the average over 25 runs. The one-step Q-learning and RTDP lines are included for reference.

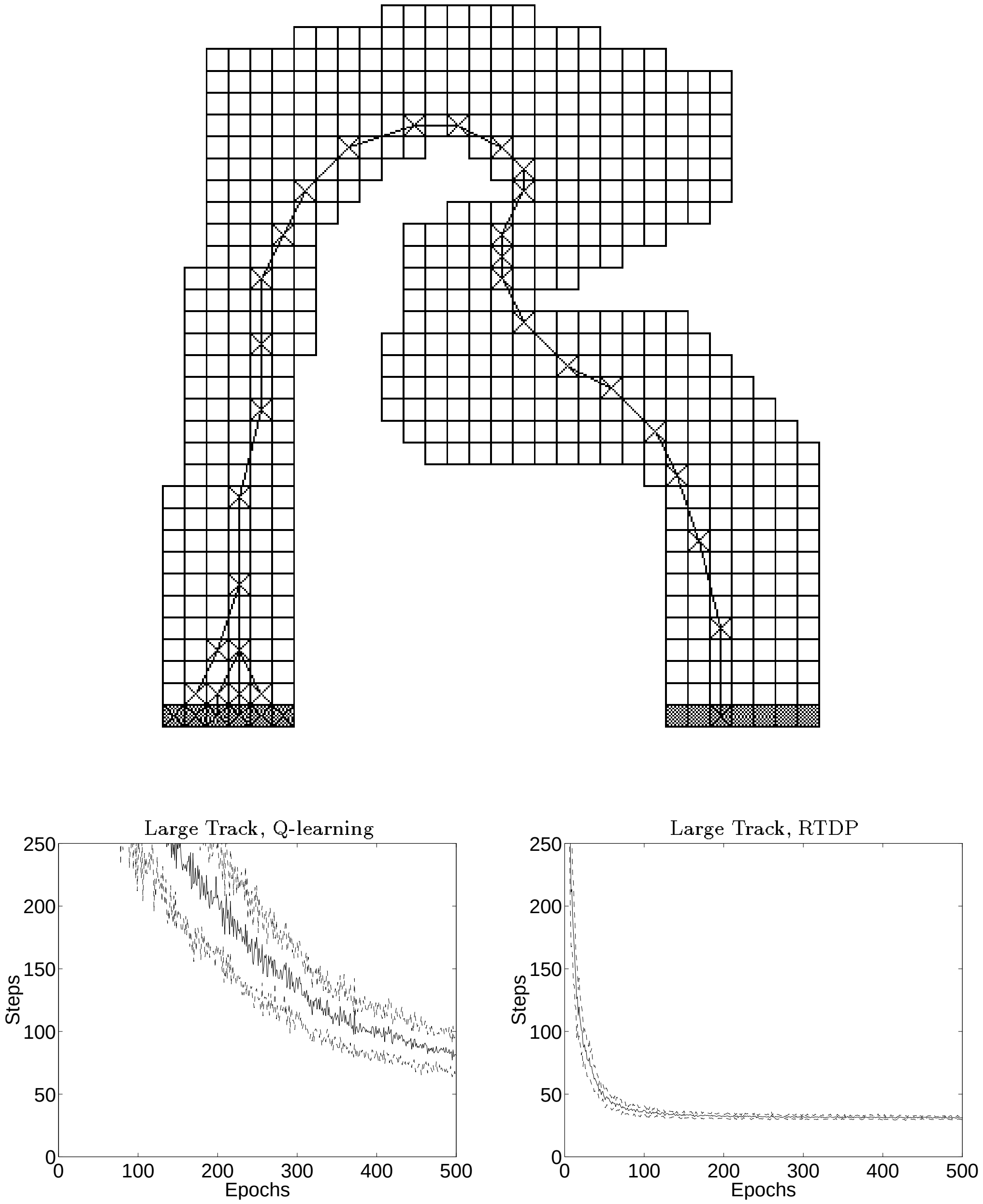
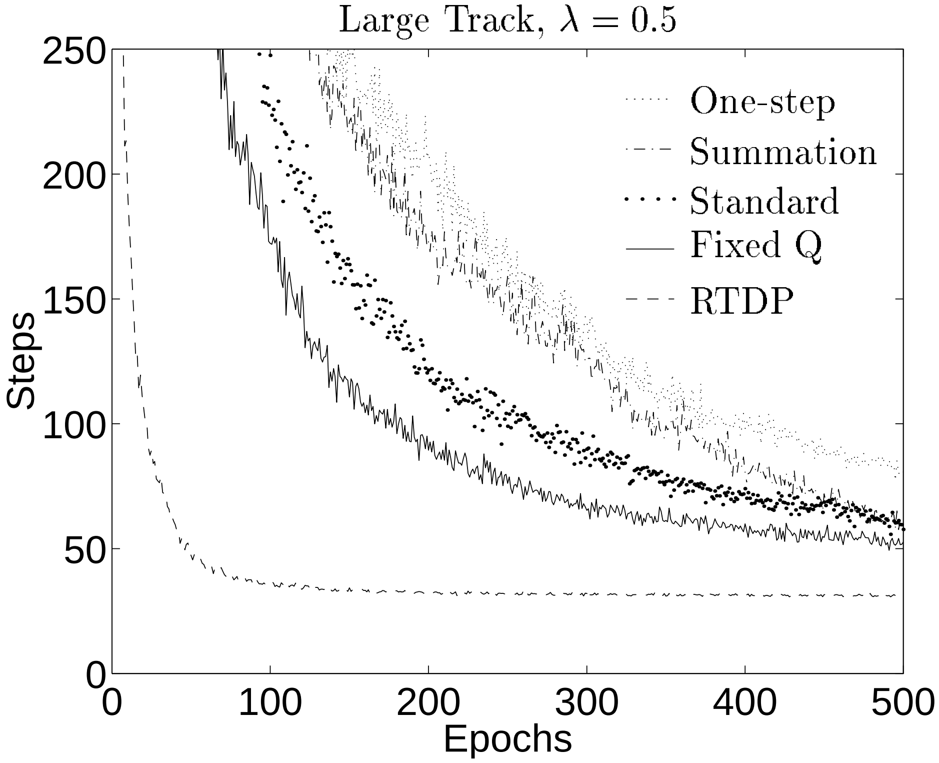


Figure 2.4: *Top:* The large race track used for testing. The start line is on the left and the finish line is on the right (the shaded squares). The lines show a typical trajectory achieved by the robot after training. *Bottom:* Graphs for one-step Q-learning and Real-Time Dynamic Programming. The dashed lines mark one standard deviation either side of the mean as measured over 25 runs.



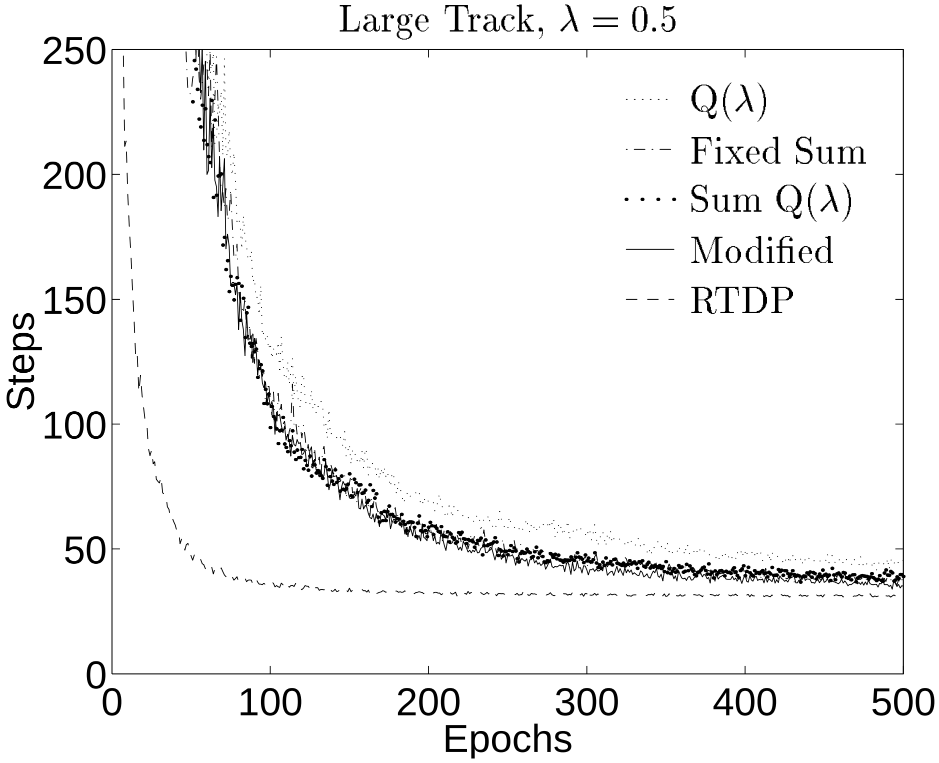
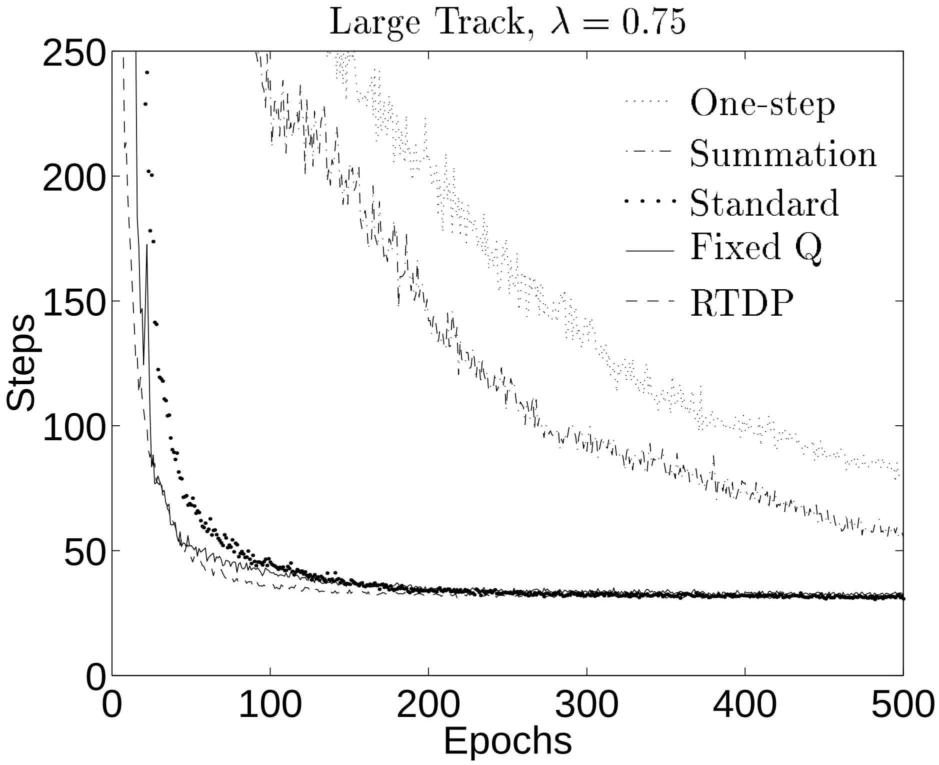


Figure 2.5: Large race track tests for $\lambda = 0.5$. Graphs show the relative performance of the different update rules across epochs. Each epoch consists of 20 trials and each curve is the average over 25 runs. The one-step Q-learning and RTDP lines are included for reference.



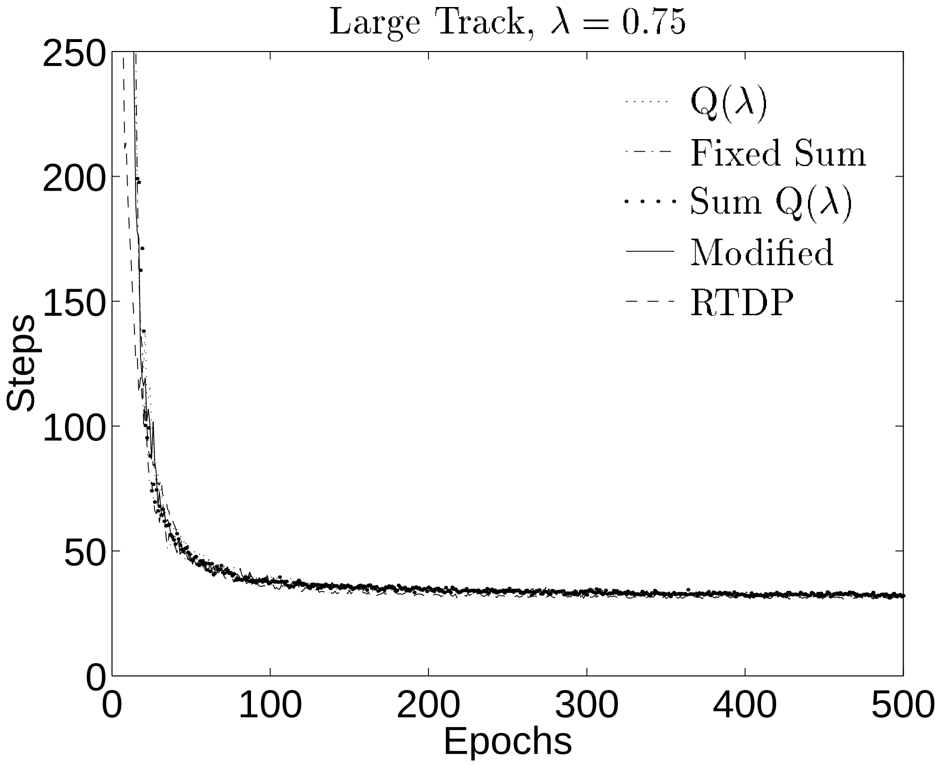
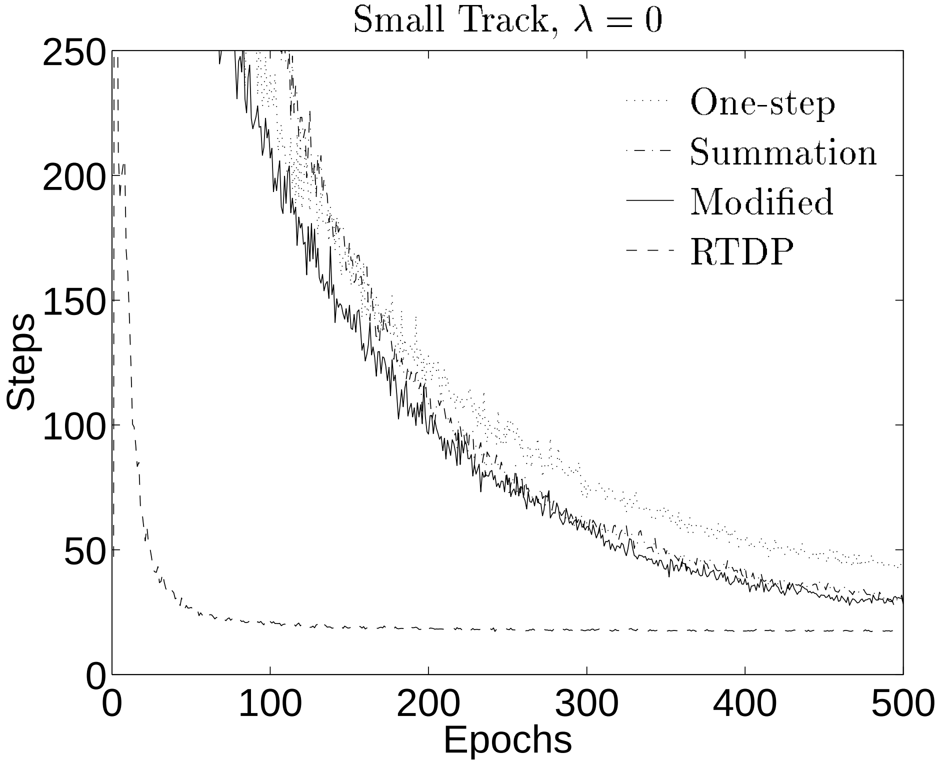


Figure 2.6: Large race track tests for $\lambda = 0.75$. Graphs show the relative performance of the different update rules across epochs. Each epoch consists of 20 trials and each curve is the average over 25 runs. The one-step Q-learning and RTDP lines are included for reference.



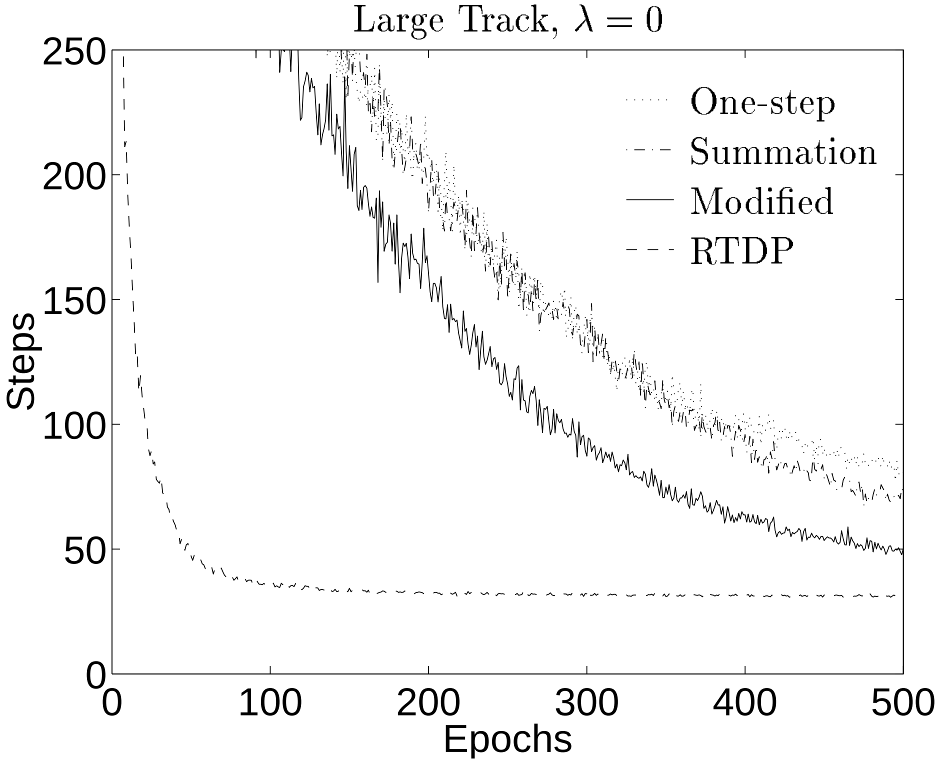


Figure 2.7: Results for one-step ($\lambda = 0$) learning using Q-learning, Summation Q-Learning, and Modified Q-Learning updates. The RTDP curve is shown for comparison.

in order to operate. Performing each RTDP update requires calculating the outcomes of performing all of the actions in the state, including all of the alternatives caused by the probabilistic nature of the state transitions. In the Race Track problem, this means that the computational expense is actually greater than the Q-learning methods, despite the fact that the combined Q-learning and TD(λ) methods require several updates to be performed at each time step due to the discrete buffer of eligibilities traces. Given the small improvement RTDP brings even in the Race Track problem where the it has access to a *perfect* model of the environment, it suggests that Q-learning methods are of more practical use for tasks where the environment is harder to model.

2.3.4 What Makes an Effective Update Rule?

On this problem, the best Q-function update rules are Modified Q-Learning, Q(λ) and Summation Q(λ), which all perform similarly. What they have in common is that they all use a constant $\gamma_t = \gamma$ which ensures that the eligibilities are never zeroed and so future TD-errors are seen by previously visited states. The actual update made at each step, whether it is based on Q_t, $\max Q_t$ or $\sum_a P(a|\mathbf{x}_t)Q_t$, is not so critical. However, as the results for $\lambda = 0$ show (Fig. 2.7), the least biased estimate, Q_t, performed the best by providing the most new information.

In finite length trials, the most important state and payoff is the final one, as this 'grounds' the overall TD-error that is seen by the system and thus contains the most information of any of the updates. This can be most clearly understood by considering early trials. At this time, the predictions at each state will just be random initial values and not represent good estimates of the return available. The immediate payoffs will allow the system to move the predictions to the right levels relative to one another, but it is only the final state that will provide the absolute indiction of the return available.

It is therefore clear why the update rules that result in the most states receiving this final information do better than methods such as Summation Q-Learning, which uses low eligibility values, or standard Q-learning, which restricts which states see this information by reducing the eligibilities to zero every so often. It also makes it clearer why high values of λ provide faster convergence in this task than low values.

2.3.5 Eligibility Traces in Lookup Tables

In order to implement the eligibility traces in a lookup table, there are several options. One is to maintain one eligibility at every state $\mathbf{x} \in \mathbf{X}$, then to update them *all* at every time step according to equation 2.10 and all the predictions according to equation 2.9. An alternative is to maintain a buffer of the most recently visited states and only update those. This latter option is the most commonly used, as if the product $(\gamma_t\lambda)$ is less than 1, then the eligibilities will decay exponentially towards zero. Thus, after only a few time steps they will usually be small enough to be removed from the update list without introducing much prediction bias. This was the method used in the experiments.

A recent method (Cichosz 1995) provides an alternative and potentially computationally cheaper method of performing these updates, by keeping track of the n-step truncated return and using this to update the prediction made at time $t - n$ only. The disadvantage is that this method does not allow for the case where γ_t varies with time (as is required by standard Q-learning and Summation Q-Learning). However, it could be used with Modified Q-Learning updates without a problem.

2.4 Summary

A number of alternative Q-learning update rules have been introduced which seek to provide the fastest convergence when temporal difference updates with $\lambda > 0$ are made. Of those presented, Modified Q-Learning, Q(λ), and Summation Q(λ) have been found to provide the fastest convergence. However, of these methods, Modified Q-Learning has the advantage of being the simplest and thus least computationally expensive, and so on this basis represents the most successful update rule presented in this chapter.

In addition, one of the most important aspects of this chapter is that standard Q-learning, the method suggested by Watkins (1989) and used extensively since, does not provide the best convergence rates. In fact, in this chapter, a whole host of alternative update rules have been suggested that empirically outperform it. This may be a quirk of the Race Track problem, but in chapter 4 it is shown to be outperformed on the Robot Problem and in Tham (1994) it is shown to be outperformed on a multi-linked robot arm problem. Certainly, each task is similar, in that each is a control task requiring the system to achieve certain goals, but it is this type of task for which the reinforcement learning methods presented in this thesis are primarily intended.

Chapter 3

Connectionist Reinforcement Learning

One of the fundamental decisions for learning policies or value functions is how to store the information that is gathered. This is especially true when continuous state-space problems are considered. The ideal system must preserve all the salient information that has been learnt, whilst generalising this information to other states in order to reduce the learning time. These two aims conflict, however, as generalising can lead to previously learnt information being lost, and no generalisation to very slow convergence times, whilst explicitly storing every piece of information gathered can lead to huge storage requirements.

In *supervised* learning tasks such as pattern classification, it is usual to use a fixed training set of data, which is presented to the function approximator repeatedly until it has learnt the input-output mapping to the required accuracy. This introduces the possibility of *over-fitting*, which occurs if the function approximator learns atypical characteristics of the training set thus reducing its ability to generalise to new data. In reinforcement learning, this problem does not occur, as the training data is generated continuously by actual experiences and usually only used once. This means that noisy, unusual features are not presented to the function approximator repeatedly and so it will not over-fit by learning them. On the other hand, characteristics of regions of the state-space that are not visited very often may be 'forgotten', precisely because they are not presented often enough to the function approximator.

Another factor with reinforcement learning systems is that the required input-output mapping is not known at the start of training. For example, the function approximator may accurately predict low payoffs for the current policy, but the reinforcement system will be using this information to change its policy in order to increase the payoffs, which will change the required predictions. This means that early training data *should* eventually be forgotten by the function approximator, as it will no longer be an accurate reflection of what the system knows.

This chapter starts by briefly reviewing some of the function approximation techniques available that are suitable for use in reinforcement learning. It is suggested that neural networks, in particular *multi-layer perceptrons* (MLPs), provide one of the most promising general purpose methods currently available and so details of this type of function approximator are discussed in the second section. Finally, methods for performing updates of MLPs when using temporal difference learning algorithms are presented, which are useful for implementing the Q-learning update rules discussed in the previous chapter. The two methods examined are *backward-replay* (Lin 1992) and *on-line*[1] updating.

3.1 Function Approximation Techniques

The requirements for a function approximator to be useful for reinforcement learning systems operating in high-dimensional continuous state-spaces are:

- An on-line learning algorithm that can learn from individual training examples, as opposed to requiring batches of data to converge.
- Good scaling with the dimension of the input vector.
- Ability to provide continuous outputs in response to continuous inputs.
- Generalisation of data to provide faster convergence of predictions for all points in continuous state-spaces.

It is important to bear in mind that reinforcement methods are an iterative technique. Although the system is trying to converge to an optimal value function and thus policy, the output values of this function are not known in advance. This rules out learning methods which base the update of the function approximator on calculations performed across a fixed training data set.

MLPs fulfil all of the requirements set out above, especially when utilising the on-line training algorithms introduced later in this chapter (section 3.3). However, other types of function approximator have been used for reinforcement learning and some of these are discussed below.

3.1.1 Lookup Tables

A lookup table was used in the Race Track experiments at the end of the last chapter in order to store the Q-function values, and also to store a count of the number of times that state-action pair had been visited (used in the calculation of the learning rate, η_t). When dealing with an environment that can be broken into discrete states, such as described by the Markovian formalisation, the lookup table provides the most obvious method. The main disadvantage of this method is its huge storage requirements when the state-space becomes large, and the fact that there is no generalisation between states and thus each one must be visited repeatedly for the system to converge.

In a continuous state-space, a lookup table can still be used by partitioning the state-space into separate regions and then associating each block of states with a lookup table entry. This acts as a crude form of generalisation and its success will depend on how well the function being learnt can be represented by a quantised state-space. Regions in which the gradients $\partial V(\mathbf{x})/\partial x$ are high will not be well represented unless the space has been divided up into small regions. Therefore, for greater accuracy, larger lookup tables are required, which require more storage and so more updates to train. On the other hand, using smaller lookup tables may speed convergence, but the function may not be stored well enough to be useful for determining the optimum policy.

[1] On-line in this context means a method that allows the function approximator to be updated at each time step, and so enables the system to operate continuously without requiring a separate learning phase at the end of each trial.

Lookup tables have been used extensively in the literature e.g. (Watkins 1989, Barto et al. 1993, Sutton 1990), but always for problems with a low dimensional input due to their poor scaling characteristics (as is discussed in section 3.1.4).

3.1.2 CMAC

CMAC (Albus 1981) stands for *Cerebellar Model Articulation Controller*, and is a compromise between a pure lookup table and a continuous function approximator. It is effectively a set of lookup tables, with different quantisation boundaries used by each. The usual method is to use the same quantisation resolution for each table, but with the boundaries of each table offset from one another. Other options are possible, such as having different quantisation resolutions for each lookup table.

The final output of the CMAC is found by summing together the values found for the current state from each of the lookup tables. This gives the generalisation advantage of a coarse resolution lookup table, whilst still giving a fine resolution for individual function values. Hence CMACs can be used to provide generalisation in discrete state-spaces as well as continuous state-spaces and so can be used in discrete problems to try to speed up learning (Watkins 1989).

CMACs have been used successfully for reinforcement learning in some quite complex continuous state-space problems, including a robot manipulator task (Tham and Prager 1992).

3.1.3 Radial Basis Functions

Certain radial basis function (RBF) networks are closely related to CMACs and lookup tables, where instead of simply storing a discrete table of values, a lattice of functions such as Gaussians or quadratics is used. Each point in state-space receives a contribution from each of these functions and these contributions are then summed to provide the final output of the function approximator.

3.1.4 The Curse of Dimensionality

RBF lattices, CMACs, and lookup table techniques can all be incrementally updated with on-line data and so are suitable for reinforcement learning problems. However, they all suffer from the same problem — the so called 'curse of dimensionality' — in that they do not scale well to high dimensional input spaces. Consider a system with I inputs. It might be decided that in order to provide the resolution needed to represent the function mapping accurately, N basis functions in each dimension are needed. Therefore N^I basis functions would be required to represent the function across the whole state-space. It can be seen that the number of individual basis functions (or entries in a lookup table or CMAC) rises exponentially with the dimension of the input vector.

The Race Track problem (section 2.3) in the last chapter demonstrated how large the storage requirements could be even for a simple learning task. The function approximator used was a lookup table with one entry per state-action pair. Due to the fact that the large race track occupied a grid of 30×33 and that the robot could theoretically accelerate to a velocity between ± 8 in each direction, 286,110 states had to be allocated. With 9 actions available per state, this meant that 2,574,990 separate Q-function values had to be stored. Thus it can be seen why a method such as one-step Q-learning, without using TD(λ) methods or generalisation, can be expected to take a very long time to converge.

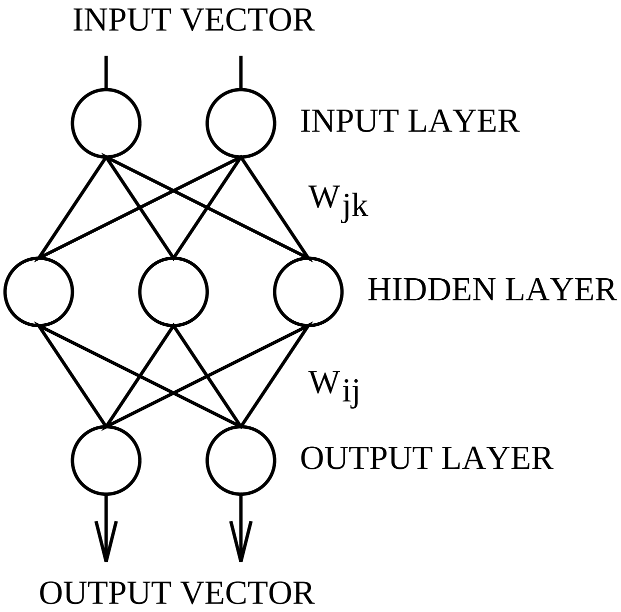


Figure 3.1: A neural network with 2 inputs, 3 hidden nodes, and 2 output nodes.

3.2 Neural Networks

An alternative to the function approximation methods so far described is *neural networks*. Neural networks are made up from interconnected units (often called perceptrons) which each combine their inputs using a simple function to arrive at an output value. This is then fed as an input to other units, or provides an output of the network as a whole (see Fig. 3.1).

These *connectionist*systems have the potential to be implemented in parallel hardware, with the individual units working simultaneously. However, currently neural networks are usually implemented on serial computers, which require that each unit is dealt with sequentially, but this does not detract from their potential power for high speed parallel computation.

One advantage over the function approximators discussed in the last section comes from the fact that the size of the network is determined by the number of perceptrons in the network, which is independent of the dimension of the input vector. So, if N perceptrons are considered necessary for the function to be approximated well, and the input vector has I components, then the network size is proportional to $I \times N$ (roughly speaking; it depends on the exact architecture of the network).

Neural networks can be used to approximate complex functions and provide generalisation. The disadvantage is that the generalisation can lead the networks to lose information previously learnt, especially for regions of the state-space that are not visited and updated regularly. It has been suggested that this problem could be dealt with by splitting the state-space into regions with a different neural network to learn the function in each region (Jacobs, Jordan and Barto 1991, Jordan and Jacobs 1992). Gating networks are then used to select which networks to use in each region, or even provide a weighted sum of the outputs of the networks. In chapter 5 a similar idea is explored when the Q-AHC architecture is presented which combines real-valued AHC learning elements with a Q-learning action selector. This selector therefore performs a similar function to the gating network mentioned above.

Neural networks therefore offer a powerful method for function approximation and hence in this thesis are examined in the context of storing information learnt by reinforcement learning algorithms. The following sections give a more detailed overview of their architecture.

3.2.1 Neural Network Architecture

The term *neural network* has come to represent a whole variety of architectures for function approximators, with the loose definition that they are all constructed from sets of perceptrons which share inputs and outputs. The specific neural networks considered in this thesis are the popular *multi-layer perceptron* (MLP) or *back-propagation network* type (Rumelhart et al. 1986).

Each perceptron has multiple inputs o_j and a single output o_i which is calculated from the inputs as follows,

$$o_i = f(\sigma_i) \tag{3.1}$$

where $f(.)$ is a *sigmoid* function, and,

$$\sigma_i = \sum_j w_{ij} o_j + b_i \tag{3.2}$$

where w_{ij} are the *weights* and b_i is the *bias weight.* The inputs o_j could be external inputs to the neural network or outputs from units in the layer above, depending on the position of this unit within the network. The sigmoid function is a non-linear S-curve, which can be defined in several ways, for example by using a hyperbolic tangent function. In this work, however, the following sigmoid function is used throughout,

$$f(\sigma) = \frac{1}{1 + \exp^{-\sigma}} \tag{3.3}$$

This gives an output approaching 1 if σ is large and positive, and 0 if it is large and negative. Thus a sigmoidal unit splits its input space by a hyperplane defined by the weight settings. The output approaches 1 on one side and 0 on the other, with a smooth transition in between.

The units are arranged in layers as shown in Fig. 3.1, with the output from each unit in one layer being used as an input for each unit in the layer below. So, the first layer extracts features from the network inputs, the second extracts features from the outputs of the first, and so on down through the layers until the network outputs are reached. More complex interconnections are possible, including direct links from the inputs to the output units (bypassing the hidden layer units), or feeding back outputs to previous layers to give *recurrent* networks (Werbos 1990, Williams and Zipser 1989).

The exact function performed by the network is dependent on the current weight values at each unit, and it is by changing these values that the network can be *trained* to produce a particular input-output mapping. The basic method is to compare the actual output of the network with the required output and adjust the weights in order to reduce the error. In order for the weights to converge to a good approximation of the function, repeated updates must be performed. One such method is *back-propagation*, a simple gradient descent rule that is discussed in section 3.2.7.

Despite the fact that this thesis concentrates on only one type of neural network, the multi-layer perceptron, with a specific architecture, there still remain a number of issues pertaining to its use, which are briefly explored in the following sections.

3.2.2 Layers

It has been shown by several authors (Hornik, Stinchcombe and White 1989, Cybenko 1989, Funahashi 1989), that one hidden layer is all that is required for an MLP to approximate an arbitrary function. However, the number of hidden units actually needed in this hidden layer could be huge. For this reason, for some problems, two or more hidden layers are a better option.

Unfortunately, increased numbers of layers create a problem for training algorithms like back-propagation due to the manner in which the errors become attenuated as they are passed back through the network. Also, as is discussed in the next section, there is currently no ideal method to automatically determine the number of hidden units required in a layer, so more hidden layers introduce more parameters to experiment with in order to get the best results.

3.2.3 Hidden Units

As mentioned above, a single hidden layer network is potentially all that is needed to approximate any input-output mapping, but this is dependent on the number of hidden units used. The ideal is to use as few as possible, not only to reduce computation time on serial computers, but also to improve the generalisation abilities of the network. This is because the more units, and hence weights, that are available, the more degrees of freedom the network has to approximate a given function. For networks trained using a fixed training set, this can lead to solutions that work well for the training data, but perform badly on more general test data. However, as discussed earlier, this problem is not so important for reinforcement learning systems, which have access to a continuous stream of new training examples. In experiments, it was found that 'oversized' networks did not effect the final performance of the system, but did result in longer training times (both in terms of processing and number of updates required for convergence).

Rather than fix the number of units in advance, methods of adjusting the number of hidden units automatically (Reed 1993, Hassibi and Stork 1993, Lee, Song and Kim 1991) have been suggested, which involve trying to remove units which perform no useful function. Ultimately, the requirement is for more sophisticated learning algorithms that can be applied to arbitrarily large networks, and which only utilise as many units of the network as are required to fit the function being learnt. Steps in this direction have been made by considering Bayesian methods to select from the state-space of possible network weights on the basis of probability distributions (Mackay 1991, Neal 1995). However, these methods have relied on second order calculations based on data gathered from fixed training sets, which are not suitable for use with on-line reinforcement methods.

3.2.4 Choice of Perceptron Function

For MLPs with several layers of units, each using the same non-linear function, sigmoidal units are the type that have been studied extensively (and thus are used in this work). However, it should be noted that alternative functions could be used.

For example, consider a function to be approximated which is zero everywhere apart from a 'bubble' of points of value 1. To define this region using the hyperplanes produced by sigmoidal units would require surrounding it with planes. For this case, a more localised function would be appropriate, such as a Gaussian. For this reason, various alternatives have been suggested, for example RBF units (discussed in the context of a fixed lattice in section 3.1.3). However, in these cases, it is usual for only one layer of units to have the specialised function — the output units simply perform a weighted sum of the outputs from the functions.

3.2.5 Input Representation

The choice of representation of the input vector **x** to the network is obviously very important, as it defines one half of the input-output mapping that is to be learnt. Usually, the more useful pre-processing that is done, the less complex the function mapping that needs to be learnt.

Hence, in the Robot Problem in the next chapter, the state-space is defined by a set of continuous variables, but they are not simply presented to the network directly. This is because a real-valued input may require very different responses over different ranges, which will take several sigmoids to model if the network must base this information on one input. Therefore, it was chosen to spread each real-value across a number of inputs using a *coarse coding* technique. This means that each input is more sensitive to the value when it is a particular range and thus the network can more easily model discontinuities in the function mapping to be approximated. The exact coarse coding method used is described in Appendix A.

3.2.6 Training Algorithms

The method used to select the weight changes within the network is critical to its training speed and also to the quality of the final solution arrived at. As yet, there is no clear optimal algorithm, and many different methods have been proposed (for an overview, see Jervis and Fitzgerald (1993)). These fall into two main categories:

- Batch or off-line
- On-line

In the first category, updates to the weights are only made after a number of input-output pairs have been presented to the network and the errors have been seen. A calculation is made based on this set of input data to try to find an optimal change in weights to reduce the overall observed error in the output. This is usually used where a fixed set of training data is available, in which case the weights are only updated after the whole set has been presented to the network.

With on-line learning, the weights are updated after each individual input-output pair is presented. This is more suitable for systems where data is arriving continuously, and the desire is to learn from the data and then discard it. Therefore, this is the style of learning required in reinforcement learning systems, where there is no *fixed* training set, as the value functions are evolving as the system learns more about its environment. This rules out many of the more sophisticated training methods (e.g. Moller (1993), Mackay (1991)) which rely on estimations made about the global error surface with respect to the training set.

Furthermore, in reinforcement learning problems, where the desire is to create autonomous systems that can operate in real-time, it is preferable that the training method for the networks should not interfere with their inherent parallelism. For this reason, *local* adaptive techniques are desirable which update the weights of each perceptron separately, rather than requiring global calculations based on the entire state of the network (not least because of the computational expense involved). A comparison of some of the local methods available can be found in Riedmiller (1994).

3.2.7 Back-Propagation

In the following work, the back-propagation algorithm (Rumelhart et al. 1986) was used due to the fact that it is a local update rule that can be applied on-line, and also because it could be accelerated by combining it with TD(λ) methods (section 2.1), as is shown in the next section.

The algorithm is a gradient descent rule. Thus it seeks to minimise the output error by moving the weights in the direction indicated by the partial derivatives $\partial E/\partial w_{ij}$. E is the sum squared error,

$$E = \frac{1}{2}\sum_i (t_i - o_i)^2 \tag{3.4}$$

where o_i is the actual output of unit i and t_i is the required (target) output. So for units in the output layer,

$$\frac{\partial E}{\partial w_{ij}} = (t_i - o_i)f'(\sigma_i)o_j \tag{3.5}$$

where $f'(.)$ is the first derivative of the sigmoid function with respect to σ_i. The weight w_{ij} can then be updated using,

$$w_{ij} \leftarrow w_{ij} + \eta\frac{\partial E}{\partial w_{ij}} \tag{3.6}$$

where η is the learning rate parameter.

For units in the first hidden layer the partial derivative is,

$$\frac{\partial E}{\partial w_{jk}} = \left[\sum_i (t_i - o_i)f'(\sigma_i)w_{ij}\right] f'(\sigma_j)o_k \tag{3.7}$$

which includes the summation of gradients $\partial E/\partial o_j$ for each of the output units, and hence the term back-propagation, as gradient information is propagated through the network from output to input. This can be repeated for as many layers as the MLP possesses.

3.2.8 Momentum Term

A simple method to accelerate the convergence rate of learning for MLPs is to use a *momentum term*. Instead of using only the current error gradients in order to calculate the amount of change that should be made to each network weight, a weighted sum of the most recent gradients may give a better indication of how to change the weights. Thus the weight updating equation becomes,

$$m_{ij} \leftarrow \alpha m_{ij} + \eta\frac{\partial E}{\partial w_{ij}} \tag{3.8}$$

$$w_{ij} \leftarrow w_{ij} + m_{ij} \tag{3.9}$$

where $0 \leq \alpha \leq 1$ is the weighting parameter which controls how much the gradients are averaged over time. In practice, quite high values of α seem to give the best convergence times.

In the next section, temporal difference methods using MLPs are presented, where the weight change at each time step is calculated using the eligibility trace mechanism. This will be seen to result in an update mechanism very similar to the momentum term, except that it uses a weighted sum of the output gradients $\partial o_i/\partial w_{ij}$ rather than the error gradients.

3.3 Connectionist Reinforcement Learning

In the preceding sections, function approximation techniques that are suitable for reinforcement learning have been discussed. They all have the quality that they can be incrementally updated in response to each new piece of data, which is a prerequisite for on-line reinforcement learning methods. Of the methods presented, MLPs have the advantage of being continuous function approximators that scale well to high-dimensional inputs. In addition, they generalise the information learnt and can be implemented in parallel hardware.

In fact, MLPs have already been used to learn the value function in a very large discrete state-space problem — that of playing back-gammon — and were able to learn to play to grand-master level (Tesauro 1992). This required a very large input vector to represent the current state of play and so approximating the function using lookup table or CMAC methods would have been impractical.

In the remainder of this chapter, methods are examined for applying Q-learning updates utilising MLPs as function approximators. Unlike previous work in this area (Lin 1993b), the algorithms presented here can be applied *on-line* by the learning system. This results in a reinforcement learning system which fulfils the following goals:

- On-line training for autonomous systems.
- No world model is required.
- Efficient scaling to high-dimensional inputs.
- Generalisation for large and continuous state-spaces.
- Potential for implementation using parallel hardware.

The next sections introduce the general on-line temporal difference update algorithms and the remainder the specific Q-learning algorithms.

3.3.1 General On-Line Learning

In chapter 2, a temporal difference prediction problem was discussed, where a sequence of predictions P_t were made of a generalised return. This assumed that a single prediction was made at each time step, which could then be updated using the temporal difference algorithm. In this section, the same general temporal difference problem is again considered, but with the further extension that at each time step a *set* of predictions $\mathbf{P}_t$ are made. Although the following discussion assumes that the predictions are made by an MLP with multiple outputs, it is valid for any function approximation technique where multiple predictions are made at each time step.

```
1.  Reset all eligibilities, $\mathbf{e}_0 = 0$.
2.  $t = 0$.
3.  Produce a set of predictions $\mathbf{P}_t$.
4.  Select one prediction to use, $P_t$.
5.  If $t > 0$,
        $\mathbf{w}_t = \mathbf{w}_{t-1} + (c_{t-1} + \gamma_t P_t - P_{t-1})\mathbf{e}_{t-1}$
5b. Recalculate selected output $P_t$.
6.  Calculate $\nabla_{\mathbf{w}} P_t$ w.r.t. the selected output $P_t$ only.
7.  $\mathbf{e}_t = \gamma_t \lambda \mathbf{e}_{t-1} + \eta_t \nabla_{\mathbf{w}} P_t$
8.  If trial has not ended, $t \leftarrow t + 1$ and go to step 3.
```

Figure 3.2: The on-line update algorithm for the general TD-algorithm, for the case where a set of predictions are made, $\mathbf{P}_t$. Step 5b is shown for the corrected output algorithm.

The basic algorithm for applying TD-learning techniques to train an MLP can be found in Sutton (1989). Each weight in the network maintains its own eligibility traces,

$$\mathbf{e}_{ij} \leftarrow (\lambda\gamma_t)\mathbf{e}_{ij} + \eta_t \frac{\partial \mathbf{P}_t}{\partial w_{ij}} \tag{3.10}$$

to keep track of a weighted sum of previous output gradients. $\partial \mathbf{P}_t / \partial w_{ij}$ represents the vector of output gradients, with one element per network output, so each weight w_{ij} has a vector of eligibilities associated with it. This ensures that the error in each output affects only the weights that directly caused it, rather than those that have built up a high eligibility due to their contribution to other outputs. Hence the algorithm is suitable for the general case of a multiple output network which has different prediction errors at each of its outputs.

This algorithm has not been widely used, with Tesauro's TD-Gammon work being one of the few applications. This is because for algorithms that require more than one output, it appears that multiple eligibilities are required per weight, with the resulting performance and storage hit this would incur compared to normal back-propagation.

However, in this work, the systems considered are those where all the outputs of the network are predictions of the expected return available (e.g. the action values in Q-learning). At each time step, only one of the outputs is selected for use as the current prediction P_t and so this will be the only one which is updated by future TD-errors. Therefore the output gradient $\nabla_{\mathbf{w}} P_t$ is only calculated with respect to the output that produced this prediction.

The important result of this is that each weight need only maintain a *single* eligibility trace e_{ij} per weight w_{ij}, as only one output gradient needs to be stored per time step. This is regardless of the number of outputs of the network, so long as only one is selected for updating per time step (see Appendix B). This fact leads to the on-line update sequence shown in Fig. 3.2.

Eligibility and Momentum Terms

Writing the update for a single weight in terms of the network output o_i that produced the selected prediction, gives,

$$e_{ij} \leftarrow (\lambda\gamma_t)e_{ij} + \eta_t \frac{\partial o_i}{\partial w_{ij}} \tag{3.11}$$

$$w_{ij} \leftarrow w_{ij} + Ee_{ij} \tag{3.12}$$

where E is the current TD-error, which is equivalent to $(t_i - o_i)$ for the particular output i that produced the prediction P_t. This form of the equations can be compared with those presented for the momentum term (section 3.2.8). It can be seen that the difference is that the eligibility keeps a weighted sum of the output gradients, whilst the momentum term keeps a weighted sum of the *error* gradients. This difference changes the characteristics of the two algorithms, as it means that the momentum term operates to smooth the change each output error will make, whilst the eligibility controls the magnitude of the change each output error will make.

Both algorithms can in fact be implemented on a computer using a single pair of update functions; one to update the eligibility/momentum term, z_{ij}, the other to update the weight, w_{ij},

$$z_{ij} \leftarrow \beta_t z_{ij} + \eta_t \frac{\partial o_i}{\partial w_{ij}} Z_i \tag{3.13}$$

$$w_{ij} \leftarrow w_{ij} + Wz_{ij} \tag{3.14}$$

where the settings of β_t, Z_i and W control the operation of the functions. Comparing the above equations with those in section 3.2.8, it can be seen that to perform momentum term updates $\beta_t = \alpha$, $Z_i = (t_i - o_i)$, and $W = 1$. For eligibility updates, $\beta_t = (\lambda\gamma_t)$, $Z_i = 1$ for the selected prediction output i and 0 for all others, and W equals the TD-error.

3.3.2 Corrected Output Gradients

The whole idea behind the methods presented in this section is that they should be applied on-line for updating MLPs. This means that when the network weights are updated in stage 5 of the algorithm (see Fig. 3.2), the output of the network is altered slightly from the value P_t that will be used to calculate the TD-error at the next time step $t+1$. Consequently, the system should really get the new ('corrected') value of P_t to calculate the TD-error, and also calculate the output gradients $\nabla_{\mathbf{w}} P_t$ with respect to this new prediction of return. However, to do this incurs the computational expense of a second pass through the network which produced the selected prediction P_t. The extra stage is shown as stage 5b in Fig. 3.2.

The convergence proof of Jaakkola et al. (1993) showed that TD-learning was still guaranteed to converge when on-line updates are made, as, in the limit, the effects of using the uncorrected outputs are not significant. Although this result applies to the case where a linear function approximator is used, it is conceivable that the same effect would also occur when other types of function approximator are used. In the next chapter, experiments are performed to test this hypothesis for on-line MLP updating.

```
1. Reset all eligibilities, $\mathbf{e}_0 = 0$
2. $t = 0$
3. Select action, $a_t$
4. If $t > 0$,
   $\mathbf{w}_t = \mathbf{w}_{t-1} + \eta(r_{t-1} + \gamma Q_t - Q_{t-1})\mathbf{e}_{t-1}$
5. Calculate $\nabla_{\mathbf{w}} Q_t$ w.r.t. selected action $a_t$ only.
6. $\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \nabla_{\mathbf{w}} Q_t$
7. Perform action $a_t$ and receive payoff $r_t$
8. If trial has not ended, $t \leftarrow t + 1$ and go to stage 3.
```

Figure 3.3: The on-line Q-learning update algorithm. The update in stage 5 is shown for Modified Q-Learning.

3.3.3 Connectionist Q-Learning

In order to represent a Q-function using neural networks, either a single network with $|A|$ outputs can be used, or $|A|$ separate networks, each with a single output. The algorithms presented in the next section can be applied to either architecture.

Lin (1992) used the one-step Q-learning equation 2.21 to update the MLP weights $\mathbf{w}_t$, calculating $\nabla_{\mathbf{w}} Q_t$ using back-propagation. In Lin (1993b), he went on to introduce a connectionist method for performing standard Q-learning using temporal difference updates with $\lambda > 0$. However, Lin applies the standard Q-learning equation 2.22 using the following algorithm,[2]

$$\Delta\mathbf{w}_t = \eta(Q'_t - Q_t)\nabla_{\mathbf{w}} Q_t \tag{3.15}$$

where,

$$Q'_t = r_t + \gamma\left[(1-\lambda)\max_{a\in A} Q_{t+1} + \lambda Q'_{t+1}\right] \tag{3.16}$$

From equation 3.16 it can be seen that each Q'_t depends recursively on future Q'_t values, which means that updating can only occur at the end of each trial. Until then, all state-action pairs must be stored and then presented in a temporally *backward* order to propagate the prediction errors correctly. This is called *backward-replay.*

To implement standard Q-learning, it is necessary to use a value of $\lambda = 0$ whenever the non-greedy action has been performed (this is the equivalent of zeroing the eligibility traces when on-line updates are being made). It is worth noting, however, that Lin's backward-replay equations (3.15, 3.16) with a constant λ are in fact an implementation of Q(λ) (section 2.2.4) and not of standard Q-learning with non-zeroed eligibilities (the algorithm referred to as Fixed Q-learning in the previous chapter; see section 2.2.6).

To implement Modified Q-Learning using backward-replay, the max operator is dropped from equation 3.16 and a constant λ is used.

On-line Q-Learning

In this section, the algorithms are presented required to apply Q-learning updates using the on-line connectionist learning equations introduced in section 3.3.1.

[2] This is also the algorithm used by Thrun in his work on connectionist Q-learning e.g. (Thrun 1994).

```
1. Reset all eligibilities, $\mathbf{e}_0 = 0$.
2. $t = 0$.
3. Select action, $a_t$.
4. If $t > 0$,
   $\mathbf{w}_t = \mathbf{w}_{t-1} + \eta\,[(r_{t-1} + \gamma \max_{a \in A} Q_t - Q_{t-1})\,\nabla_{\mathbf{w}} Q_{t-1} +$
   $(r_{t-1} + \gamma \max_{a \in A} Q_t - \max_{a \in A} Q_{t-1})\,\mathbf{e}_{t-1}]$
5. $\mathbf{e}_t = \gamma\lambda\,(\mathbf{e}_{t-1} + \nabla_{\mathbf{w}} Q_{t-1})$
6. Calculate $\nabla_{\mathbf{w}} Q_t$ w.r.t. selected action $a_t$ only.
7. Perform action $a_t$ and receive payoff $r_t$.
8. If trial has not ended, $t \leftarrow t + 1$ and go to stage 3.
```

Figure 3.4: The on-line update algorithm for Q(λ).

The full on-line Q-learning algorithm is shown in Fig. 3.3 (note that it is not shown for the case of corrected outputs discussed in section 3.3.2). It involves back-propagating to compute the output gradients $\nabla_{\mathbf{w}} Q_t$ for the action chosen, and hence updating the weight eligibilities, before the action is actually performed. At the next time step, all of the weights are updated according to the current TD-error multiplied by their eligibilities. Therefore, the only storage requirements are for the MLP weights and eligibilities, and the last selected action value, Q_t, and payoff, r_t.

The important point is that $\nabla_{\mathbf{w}} Q_t$ is only calculated with respect to the output producing the action value for the selected action a_t. Hence, if the Q-function is being represented by $|A|$ separate single output networks, then $\nabla_{\mathbf{w}} Q_t$ is zero for all weights in networks other than the one associated with predictions of return for action a_t. So the eligibilities for these networks are just updated according to,

$$\mathbf{e}_t = (\gamma\lambda)\mathbf{e}_{t-1} \tag{3.17}$$

In other words, the eligibility of the entire network decays for steps when the action associated with it is not selected. However, at every time step, *all* the networks are updated in stage 4 of Fig. 3.3 according to the current TD-error — it is the eligibilities which determine how much the individual weights will be affected.

On-line Connectionist Q(λ)

To apply on-line Q(λ) updates requires a slightly modified algorithm, which results in the update sequence shown in Fig. 3.4. Note the change in order of stages 5 and 6. This actually has an important consequence, as it means that the current network output gradients $\nabla_{\mathbf{w}} Q_t$ must be stored between time steps. The reason for this is that the gradients from the previous step, $\nabla_{\mathbf{w}} Q_{t-1}$, are required to be separate from the eligibilities $\mathbf{e}_{t-1}$ when the update is performed in stage 4. In the update sequence for standard and Modified Q-Learning (Fig. 3.3), they are combined in stage 6 of the previous time step to produce the current settings for the eligibilities and so only the eligibilities need to be stored between time steps.

The on-line Q(λ) algorithm is therefore not only slightly more computationally expensive than standard Q-learning and Modified Q-Learning (due to the complexity of the update in stage 4), but also requires storage for the output gradients, $\nabla_{\mathbf{w}} Q_t$, as well as the network weights and eligibilities. This represents a 50% increase in storage requirements per network.

3.4 Summary

A number of different function approximation methods have been discussed in the context of their usefulness for reinforcement learning. It has been suggested that multi-layer perceptron neural networks provide the range of features required for reinforcement learning systems to operate in high-dimensional continuous state-spaces, and some of the issues surrounding their use have been examined.

In particular, a method for applying on-line temporal difference updates has been examined. It has been shown that in the case where multiple predictions are made at each time step and only one is selected for updating, then only a single eligibility trace per weight is required. Based on this, Q-Learning algorithms have been presented to use MLPs to store the Q-function which can be updated on-line during trials. This removes the necessity to store state-action pairs until the end of trials and then perform an off-line learning phase, as is required by the backward-replay method suggested by Lin (1993c).

Chapter 4

The Robot Problem

In chapter 2 a number of different Q-function update rules were introduced and their performance compared on a discrete Markovian problem. In chapter 3 methods for using MLP neural networks as function approximators were presented. In this chapter, these methods are used to provide a reinforcement learning solution to a robot navigation task that takes place in a large continuous state-space.

There are three aims in this chapter:

- to compare the on-line temporal difference learning algorithm for training neural networks with the backward-replay method.
- to provide a further comparison of the convergence rates and robustness to learning parameter choices of standard Q-learning, Q(λ) and Modified Q-Learning updates.
- to demonstrate that MLP networks used with reinforcement learning methods can produce high quality solutions to difficult tasks.

This final point is important, as there is little point in worrying about the details of reinforcement learning algorithms if they cannot be used effectively for dealing with difficult engineering problems. Robot navigation tasks are a popular topic in the AI and control literature and many solutions have been proposed using methods developed in these fields (Kant and Zucker 1986, Khatib 1986, Barraquand and Latcombe 1991, Agre and Chapman 1987, Schoppers 1987, Ram and Santamaria 1993, Zhu 1991). However, it is shown here that a reinforcement learning system can train a controller which can deal with general obstacle layouts, new environments, and moving goal positions, without the need for the designer to do anything more than provide a set of sparse payoffs to define the problem.

4.1 Mobile Robot Navigation

The problem of navigating a mobile robot in a 2D environment has been tackled by many different methods in recent years. Much of this work has concentrated on *path planning*, where a fully known state space is analysed to find a suitable path. Such methods include calculating potential fields that repel the robot from obstacles and attract it towards the goal (Khatib 1986, Barraquand and Latcombe 1991) and discretising the state space in order to perform rapid searches for obstacle free paths. The problem with these methods is that they require complete prior knowledge of the obstacle configuration in order to calculate a path, and hence recalculation is necessary if this configuration is changed in any way. Also, the paths generated take no account of the robot dynamics, so a second calculation is required to work out the control actions needed to keep the robot on the pre-planned path. All these calculations must occur before the robot makes a single movement.

In contrast, *reactive* controllers use only the currently perceived situation to decide their next action. As each action is executed, the robot must re-evaluate the position it finds itself in and use this information to decide on its next action. This process is repeated until the robot reaches the goal. Knowledge based systems (Agre and Chapman 1987, Schoppers 1987) and Brooks' *subsumption* architecture (Brooks 1986) rely on previously defined rules and behaviours to decide on the appropriate actions at each step. This means that the designer must take into account the dynamics of the robot in deciding on the rules available to the system, and so the solutions must be tailored to the specific robot in question.

This is exactly the kind of task that can benefit from continuous state-space reinforcement learning techniques. The advantage of these methods is that the system learns to achieve its goal using whatever information and actions are available.[1] Consequently dynamic constraints and limitations in sensory input are automatically dealt with by the most effective policy learnt by the system.

Prescott and Mayhew (1992) present such a system, using an adaptive heuristic critic method (see section 1.3.6), but it is not goal based i.e. it avoids obstacles, but not in order to get to a target location. Millan and Torras (1992) demonstrate a robot controller that avoids obstacles to get to a goal, but their system receives potential field like payoffs from all objects at every time step (therefore requiring the positions of all objects to be known). In both cases, the systems are trained on a single obstacle layout.

Here the task of using reinforcement learning to train a robot with limited range sensor inputs (as for Prescott & Mayhew) on a goal based problem is considered, where the robot only receives a payoff at the very end of the trial. In addition, the start, goal and obstacle positions are changed after each trial, ensuring that the robot has to learn a generalised reactive policy in order to deal with the situations that it might encounter.

4.2 The Robot Environment

The robot navigation system described is a computer simulation. The robot has five range finding inputs which are spaced across the robots forward arc from $-30°$ to $+30°$ at $15°$ intervals (see Fig. 4.1) and give it accurate distance measurements to the nearest obstructions. It also always knows the distance and angle to the goal relative to its current position and orientation. The environment as seen by the robot is therefore a high-dimensional (seven separate real-valued inputs) continuous state-space.

The world the robot occupies is a square room, with randomly placed convex polygonal obstacles in it. The robot starts at a random position with a random orientation and has to reach the goal, which is also at a random position.

The robot moves by selecting from a discrete set of actions (see section 4.4). It does this until an action results in a collision with an obstacle, arriving at the goal, or a time-out (the robot only has a limited number of steps in which to reach the goal). The trial then ends and the robot receives a payoff based on its final position as described in section 4.4. The layout of the room is then completely randomised and the robot starts a new trial.

[1] As will be demonstrated in this chapter when a robot with an unreliable sensor is considered.

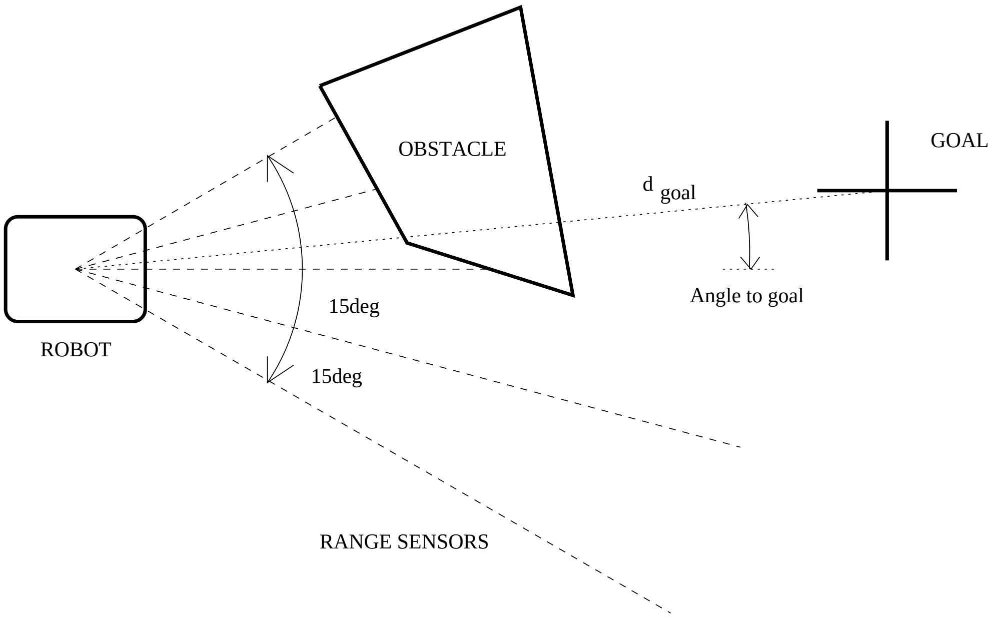


Figure 4.1: What the robot knows about its surroundings

Consequently, the only information that the robot has as to the quality of its actions is the final payoff it is given, and what it has learnt from previous trials.

4.3 Experimental Details

In the following results, the effects of applying the following three Q-learning update rules are compared:

- **Standard Q-learning** (section 2.2.1).
- **Q(λ)** (section 2.2.4).
- **Modified Q-Learning** (section 2.2.2).

These rules were discussed in chapter 2, where it was shown that standard Q-learning provided convergence rates that were lower than could be achieved using Modified Q-Learning or the Q(λ) method. The Summation Q-Learning family of update rules are not considered here, as they were shown in chapter 2 to be at best only as good as Modified Q-Learning, and to be more computationally expensive.

In addition, the effect is examined of applying these updates using both the backward-replay and on-line algorithm (section 3.3.3) presented in the last chapter.

The simulated robot was trained with 6 actions available to it: turn left 15°, turn right 15°, or keep the same heading, and either move forward a fixed distance d, or remain on the same spot. This meant it had one action — namely not moving or changing heading at all — that was never useful in achieving its objective of reaching the goal.

The robot received an immediate payoff of r_t of zero for every state except the final state reached at the end of the trial. The final payoff received depended on how the trial concluded:

Goal If the robot moved within a small fixed radius of the goal position, the payoff received was 1.

Crash The robot received a payoff based on its distance from the goal, d_{goal}, when it crashed i.e.

$$r_{\text{final}} = 0.5 \exp(-2d_{\text{goal}}/d_{\text{size}})$$

where d_{size} was the length of one wall of the square room.

Time-out If the trial timed-out (in the results presented in this section, this was after 200 steps), the robot received the same payoff as for a crash, but with a $+0.3$ bonus for not crashing.

It should be noted that the payoff for crashing was chosen in a fairly arbitrary way to give a higher payoff for ending up nearer to the goal and a maximum payoff of only 0.5. In fact, the whole payoff function was chosen based on intuitive arguments — no payoffs were given during the trial to avoid interfering with the policy found by the system and higher final payoffs were given for more desirable trial endings.

The discount factor, γ, was set to 0.99, which gives a higher weighting to actions that lead to the goal in the fewest steps.

In this work, as in Lin (1992), separate single-output MLP networks are used to predict the $Q(\mathbf{x}, a)$ of each action, rather than a single monolithic network with multiple outputs.[2] Thus, the Q-function was represented by 6 separate neural networks, one for each available action. Each network had 26 inputs, 3 hidden nodes, and a single output, and used sigmoidal activation functions of the form $f(\sigma) = 1/(1 + e^{-\sigma})$. This meant that the output of the networks was in the range $[0, 1]$ and is the reason why the end of trial payoff lies in this range. The network weights were initialised with random values in the range ± 0.1. The 26 inputs were due to coarse coding the 7 real valued sensor values using several input nodes per value (3 for each of the 5 range sensors, 5 for the distance to goal, and 6 for the angle to the goal; see Appendix A for exact details). The coding was chosen such that the range sensor inputs would only be sensitive to ranges of around half the width of the room. This was intentional and designed to simulate limited range sensor information.

A Boltzmann function was used to provide a probability distribution for exploration, where,

$$P(a|\mathbf{x}_t) = \frac{\exp^{-Q(\mathbf{x}_t, a)/T}}{\sum_{a \in A} \exp^{-Q(\mathbf{x}_t, a)/T}} \tag{4.1}$$

It is common to allow the exploration value T to start quite large and reduce as the number of trials goes by (as was done with the Race Track problem). Judging the rate of 'annealing' in this way is a matter of trial and error. In this research, it was found that a value of 0.05 for T in the early stages of learning, and of 0.01 by the end, provided the necessary trade-off between exploration and exploitation of the Q-function. In the results presented, the T value was reduced linearly between these two values over 20,000 trials.

At the start of each trial, the layout of the room was randomised (described in Appendix A). Thus, the robots saw a steady stream of new situations and so could not learn a policy which was useful only for a specific obstacle layout. This allowed a trained robot to cope with situations that it had never seen before, including dealing with moving goals and some concave obstacles. However, it should be noted that in the following results, the

[2] Separate networks avoid the weights of hidden units receiving conflicting demands from different outputs.

Training method	Successful robots (from 36)	Updates taken (millions)	Trial length (steps)
Standard	22	3.1	53.1
Q(λ)	24	2.1	38.9
Modified	26	1.3	33.6

Table 4.1: Backward-replay results. Summary of successful robots (those averaging greater than 0.98 average payoff over the last 5,000 training trials), from the 36 different λ and η combinations. Columns show number of updates made over the trials and the average number of steps required to find the goal by the end.

Training method	Successful robots (from 36)	Updates taken (millions)	Trial length (steps)
Standard	23	3.1	45.9
Q(λ)	26	2.1	36.1
Modified	28	1.1	30.5

Table 4.2: On-line update results. Summary of successful robots (those averaging greater 0.98 average payoff over the last 5,000 training trials), from the 36 different λ and η combinations using on-line updates.

sequence of rooms generated was the same for all robots to enable a sound comparison between them.

4.4 Results

There are several parameters that must be set during training. The values used for the discount factor γ and the exploration value T have been described in the previous section. These were chosen after a small amount of experimentation, but no attempt was made to find optimal values (e.g. a faster reduction in T can speed up convergence to a solution). This leaves the training rate η and the temporal difference parameter λ, which were found to have a significant effect on the ability of the neural networks to learn and the quality of the solutions achieved. Therefore, rather than performing multiple trials at fixed values[3] of η and λ the methods were tested over a range of values, so that an idea of the relative performance of the different methods, and their sensitivity to the training parameters, can be gained. The results are presented using contour plots for values of $\eta = \{0.1, 0.5, 1.0, 2.0, 3.0, 4.0\}$ and $\lambda = \{0.0, 0.25, 0.5, 0.75, 0.85, 1.0\}$, which show the average level of payoffs received by the systems over their final 5,000 trials. These plots are meant to provide a visual guide as to the quality of solutions produced by a particular method. The fact they are constructed by interpolating between the 36 data points should be remembered when interpreting them.

The first results compare the performance of the three update rules when the backward-

[3] However, such a test was performed to discover if different sequences of rooms and initial network weights had a significant effect when all else remained the same. It was found that the average payoff received over the final 5,000 trials of a run had a standard deviation of only 0.008 across 20 runs using the same value of η and λ.

replay method was used. This required the storage of all state-action pairs until the end of each trial (so, for a time-out, 200 steps needed to be recorded and then replayed to update the networks). The contour plots in Fig. 4.2 show the variation in average payoffs received by robots after being trained on 30,000 randomly generated rooms using backward-replay. Also shown is an example of the typical learning curves associated with each update rule for robots that learn to reach the goal consistently. It is worth remembering that the exploration factor T has reached its minimum value of 0.01 at 20,000 trials.

The contour plots show that the performance of the different update methods is very good over a wide range of training parameter values. The plots in Fig. 4.3 show the results when the on-line temporal difference updates are used instead. For this method of updating, it is only necessary to store the previous action value prediction in order to calculate the current TD-error. The updating of the networks is complete as soon as the final payoff has been received and used to provide the final weight update. As can be seen from the contour plots, the performance is very similar to that of the robot controllers produced by using the backward-replay method. In fact, if only the most successful robots are considered (those that average over 0.98 payoff across their final 5,000 trials[4]), then Tables 4.1 and 4.2 show that the on-line training method has resulted in a slightly higher number of successful controllers. In addition, the on-line systems reach the goal in fewer steps than used by the backward-replay systems.

The second point that can be seen from Tables 4.1 and 4.2 is that for both on-line and backward-replay methods, the Modified Q-Learning updates have resulted in the greatest number of successful robots, trained in the fewest number of updates, and requiring the lowest number of steps to reach the goal. Q(λ) comes second in all categories, with standard Q-learning just behind.

The number of updates taken in 30,000 trials varies considerably between the 3 different Q-learning methods. Fig. 4.4 shows the same graphs used in Fig. 4.3, but with the x-axis scale in terms of updates rather than number of trials. As can be seen, the Modified Q-Learning trained robots have converged to a solution in well under a million updates, compared to over 2 million for Q(λ) and 4 million for standard Q-learning.

To put this in perspective, the contour plots in Fig. 4.4 represents the average payoffs received by on-line Modified Q-Learning and Q(λ) trained robots after 1 million applications of the update rule. The contour plot for standard Q-learning is not shown as it is blank i.e. no combination of η and λ has led to a robot that averages a payoff over 0.95 (the lowest contour plotted) after 1 million updates. Meanwhile, Q(λ) updates have resulted in only 14 robots achieving average payoffs of over 0.95, which occur when the high values of λ of 0.75 and 0.85 are used.

The rate of convergence of Q(λ) as λ increases can be seen more clearly in Fig. 4.5, which also shows the graphs for the corresponding Modified Q-Learning systems. As can be seen, the change for Q(λ) is much more pronounced, with the curve for low values of λ being more similar in shape to those produced by standard Q-learning systems (e.g. the top right graph in Fig. 4.3), whilst the high values of λ result in curves that are more like Modified Q-Learning. This is no surprise, as Q(λ) with $\lambda = 0$ is exactly the same as one-step Q-learning, whilst at high values of λ, the sum of truncated returns will be increasingly similar to those seen by Modified Q-Learning (see equations 2.34 and 2.35).

Therefore, on the evidence of this section, it appears that on-line updates are at least as good as backward-replay ones, if not slightly better, and that Modified Q-Learning updates provide the best trained robot controllers in the fewest updates. However, the

[4] This level of payoff requires in the region of 96% of trials ending with the robot reaching the goal.

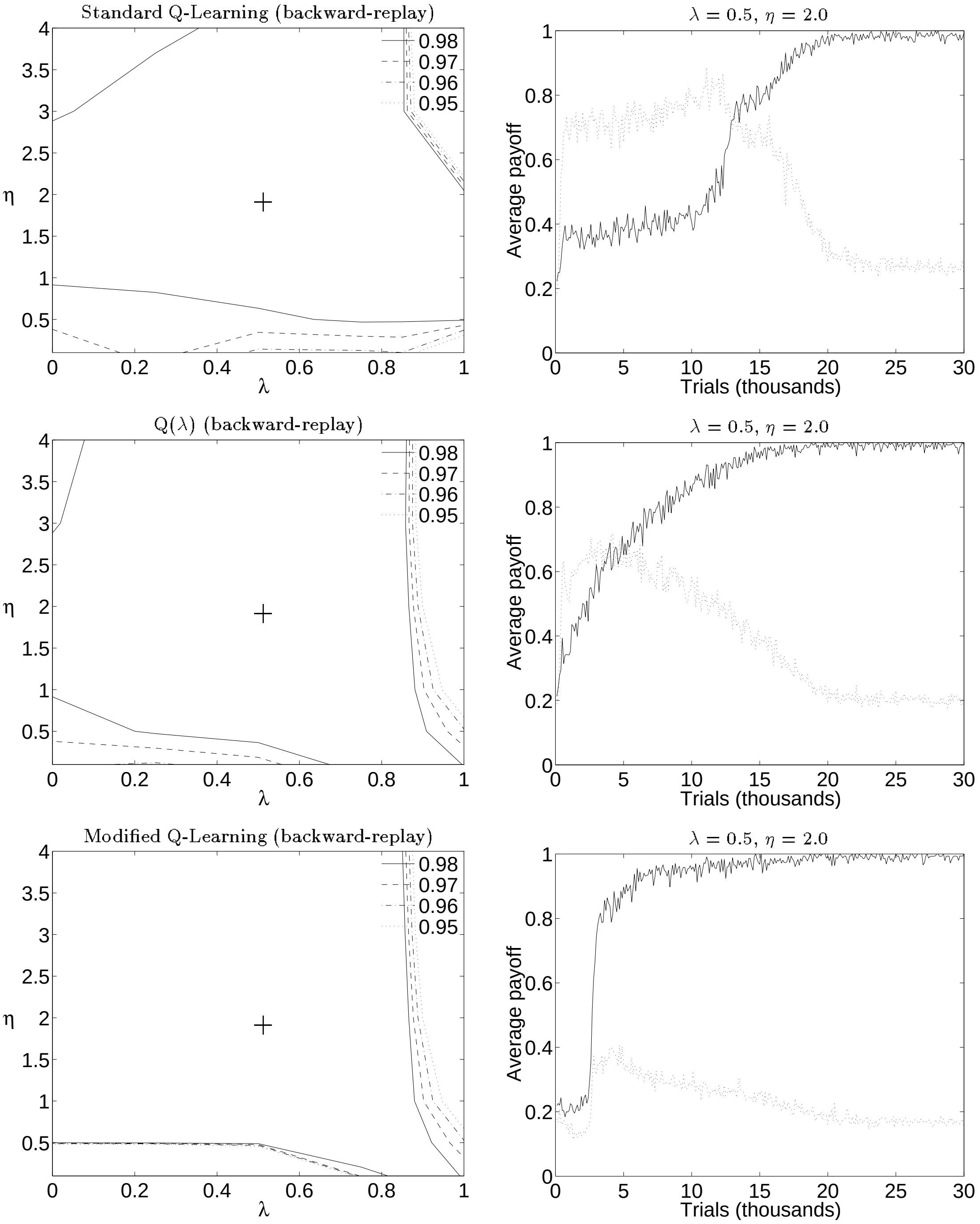


Figure 4.2: Backward-replay results. *Left:* Contour plots showing how the final payoff after 30,000 trials varies for each of the three update rules applied using different values of η and λ. *Right:* Sample training curves taken for each update rule, corresponding to the value of η and λ marked by a + on each contour plot. The dotted line is the normalised average number of steps taken in each trial (maximum trial length was 200 steps).

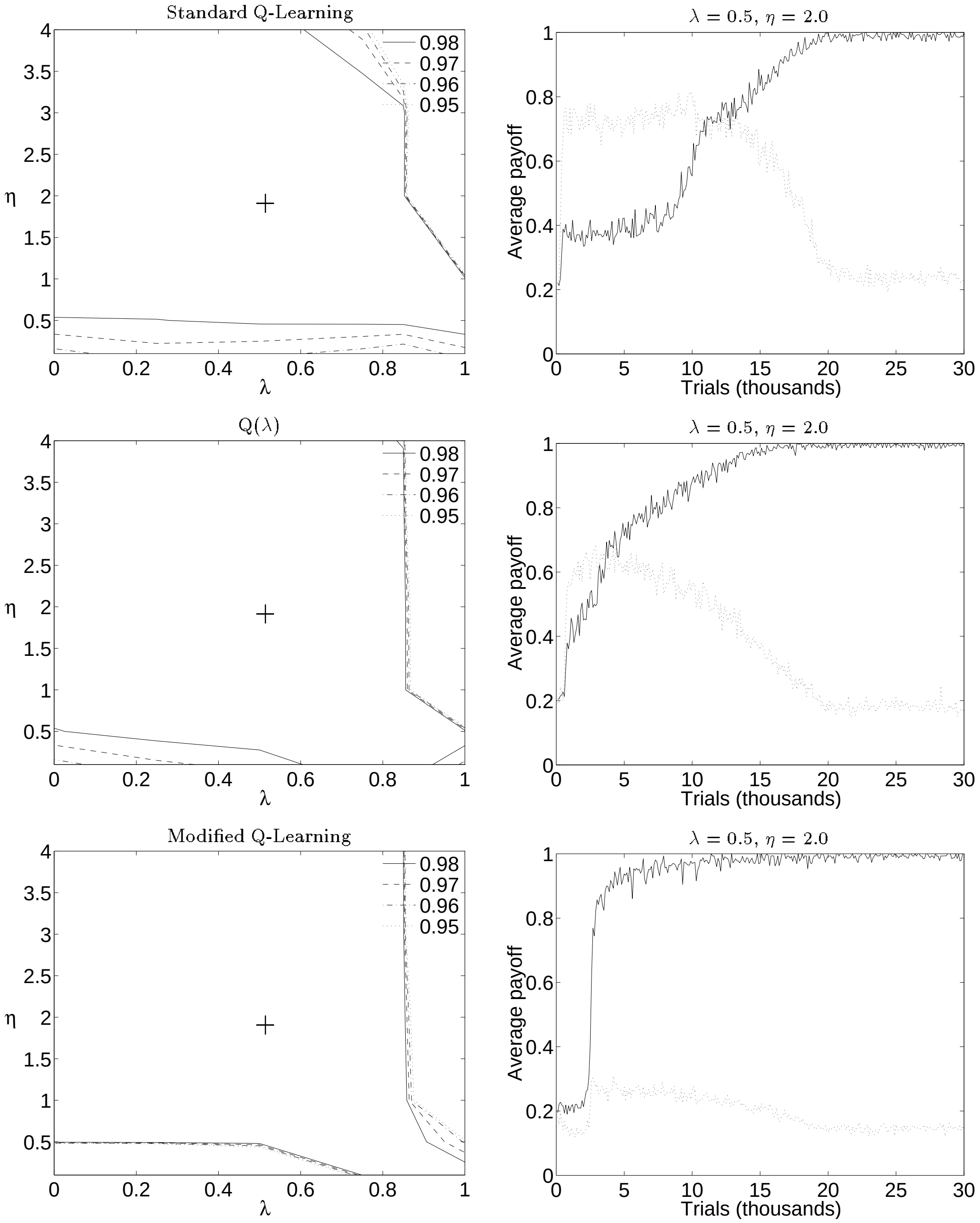


Figure 4.3: On-line results. *Left:* Contour plots showing how the final payoff after 30,000 trials varies for each of the three update rules applied using different values of η and λ. *Right:* Sample training curves taken for each update rule, corresponding to the value of η and λ marked by a $+$ on each contour plot. The dotted line is the normalised average number of steps taken in each trial (with 1.0 corresponding to 200 steps).

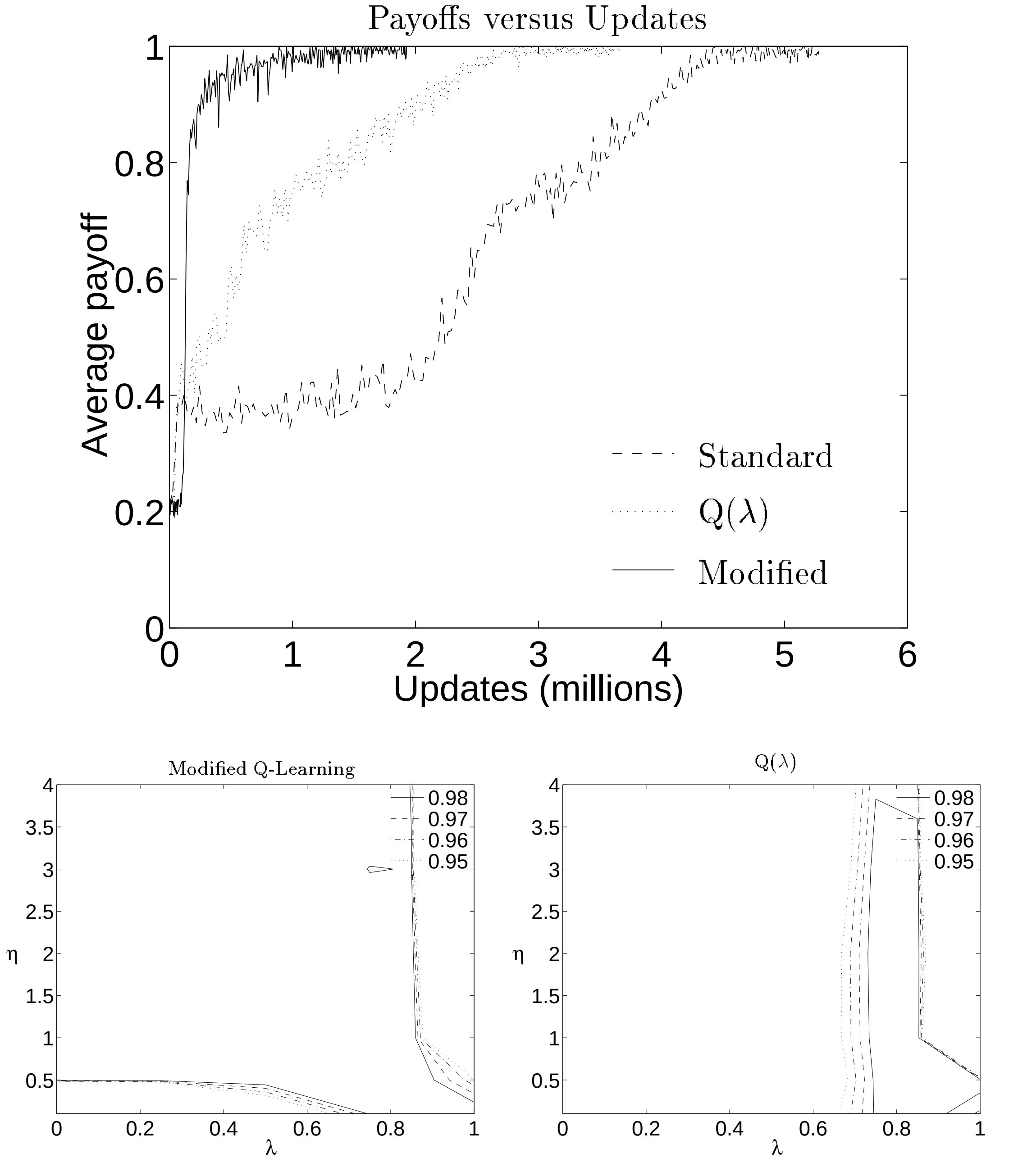


Figure 4.4: *Top:* The same three example graphs from Fig. 4.3, this time plotted against updates rather than trials. *Bottom:* The contour plots for on-line Modified Q-Learning and Q(λ) shown after one million updates.

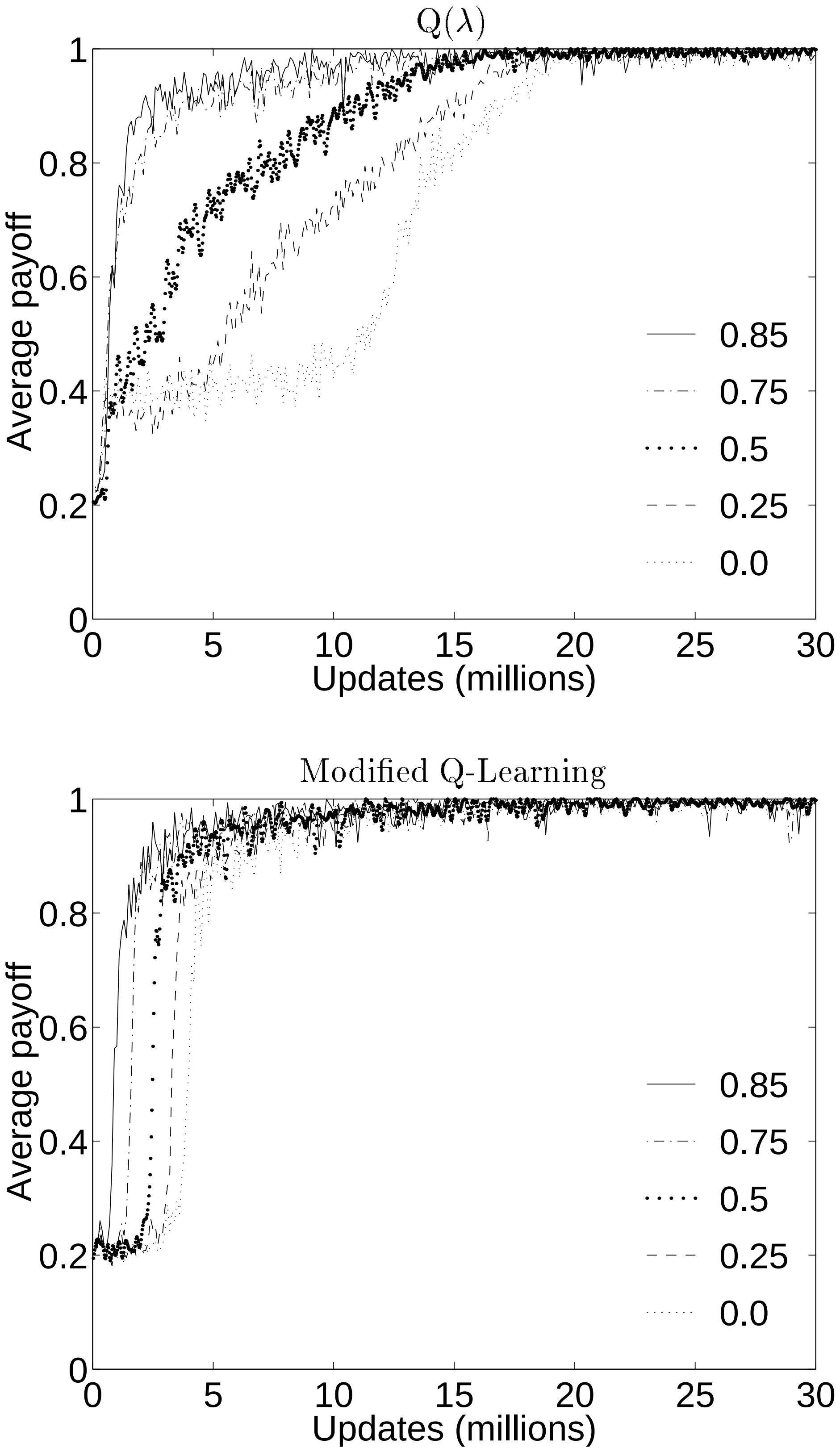


Figure 4.5: Graphs showing the learning curves for on-line Modified Q-Learning and Q(λ) for $\eta = 2.0$ using different values of λ.

Training method	Successful robots (from 36)	Updates taken (millions)	Trial length (steps)
Standard	0	-	-
Q(λ)	4	3.1	45.6
Modified	1	2.8	46.3

Table 4.3: Backward-replay results with damaged sensors. Summary of successful robots (those averaging greater than 0.98 average payoff over the last 5,000 training trials), from 36 different λ and η combinations.

Training method	Successful robots (from 36)	Updates taken (millions)	Trial length (steps)
Standard	7	4.9	49.7
Q(λ)	9	3.7	45.7
Modified	11	2.4	46.1

Table 4.4: On-line updates with damaged sensors. Summary of successful robots (those averaging greater than 0.98 average payoff over the last 5,000 training trials), from 36 different λ and η combinations.

difference in performance between the methods recorded in Tables 4.1 and 4.2 is quite small and so in the next section results are presented for a slightly modified and more difficult version of the robot problem.

4.4.1 Damaged Sensors

The results in the last section demonstrate that training a high quality reactive robot controller using reinforcement learning is quite straightforward and works for a wide range of training parameters. It also gives the impression that any of the Q-learning update rules will work equally well. However, earlier results presented in Rummery and Niranjan (1994) for this problem showed a wider variation in performance. This difference arises because in the previous work the coding for the state vector $\mathbf{x}_t$ was different to that used for the robots in the preceding section.

To be precise, in the last section, the coarse coded input for the relative angle of the goal to the robot came from the following,

$$\psi = \text{mod}(\phi - \theta) \tag{4.2}$$

where $0 \leq \phi \leq 2\pi$ is the absolute angle to the goal, $0 \leq \theta \leq 2\pi$ is the absolute facing of the robot, and the function mod(.) adjusts the calculated relative angle ψ to also lie in the range $[0, 2\pi]$. However, in the results presented in Rummery and Niranjan (1994), the mod(.) operation was not performed and thus it was possible for ψ to lie outside the range $[0, 2\pi]$ if $\theta > \phi$. This makes the state-action mapping for the Q-function significantly harder to learn, as a single relative angle, $0 \leq \psi^* \leq 2\pi$, can be represented by ψ as either ψ^* or $\psi^* - 2\pi$ depending on the absolute angles (which are unknown to the robot). As the coarse coding was designed assuming ψ would lie in range $[0, 2\pi]$, this means that the negative values lie outside of this region and are thus represented by very similar input vectors.

However, the results obtained with this 'damaged' sensor are interesting, as the robot controllers produced, whilst not as good in general as those achieved in the last section, still result in some robots capable of reaching the goal successfully 99% of the time (see section 4.4.3). Therefore, the results for this system are presented in this section, as this is a significantly harder reinforcement learning task and shows up the differences in performance of the learning rules more sharply than the results presented in the previous section. It also demonstrates the power of reinforcement learning to overcome such problems.

One of the first consequences of the damaged sensor is that convergence takes longer. Improvements to the policy still occur after the exploration value T has reached its minimum value of 0.01 after 20,000 trials. Thus, in the results presented, 50,000 trials were used to train each robot.

Fig. 4.6 is the equivalent of Fig. 4.2 for the case where backward-replay methods are used to perform the network updates. The difference is striking, and, as Table 4.3 shows, suggests that the incorrect goal angle input has virtually destroyed the ability of the system to converge to good solutions. The performance is very sensitive to the choice of learning parameters η and λ, with standard Q-learning producing no truly successful robots.

In contrast, the contour plots for on-line updating (Fig. 4.7) show far less sensitivity to the choice of learning parameters. As can be seen from Table 4.4, the number and quality of the successful robots is greatly increased. The best results are for Modified Q-Learning trained robots using on-line updates. The results are not of the same standard as for the correct angle input (Fig. 4.3), but as a comparison of Table 4.4 and Table 4.3 shows, they are significantly better than achieved by the backward-replay method on the damaged sensor task.

A close look at the policy learnt by the successful systems shows that the solution arrived at by the reinforcement learning system is to keep the goal slightly to the left of the current facing of the robot at all times i.e. in the region where $\theta < \phi$ and thus the relative angle ψ^* is positive and represented unambiguously by the calculated relative angle ψ (see the maps in Fig. 4.10).

4.4.2 Corrected Output Gradients

In section 3.3.2 of the previous chapter, the issue of the output gradients used by the on-line neural network updating method was discussed. As described, the difficulty is that the network producing the selected action value, Q_t, is updated by the current TD-error, and thus its prediction and output error gradients $\nabla_{\mathbf{w}} Q_t$ are altered. In order to 'correct' them to values reflecting the current weight settings, another forward pass is required.

However, it was questioned whether the correct output and gradients are really required; a forward pass through an MLP requires a lot of computation, even for a small network, and so introducing an extra one into the on-line updating sequence is undesirable. In the results presented so far, this correcting pass was not performed. To evaluate its effect results are presented in this section for on-line Modified Q-Learning where it has been used.

The contour plots are shown in Fig. 4.8 and Table 4.5 presents the results for the successful robots. There is quite an improvement in performance on the problem with the damaged sensor input.[5] The main area of improvement is in the region where the learning rate η is high, which is where the greatest changes between the outputs before

[5] Which would be even better if it were not for a particularly bad result at $\eta = 2.0, \lambda = 0.75$.

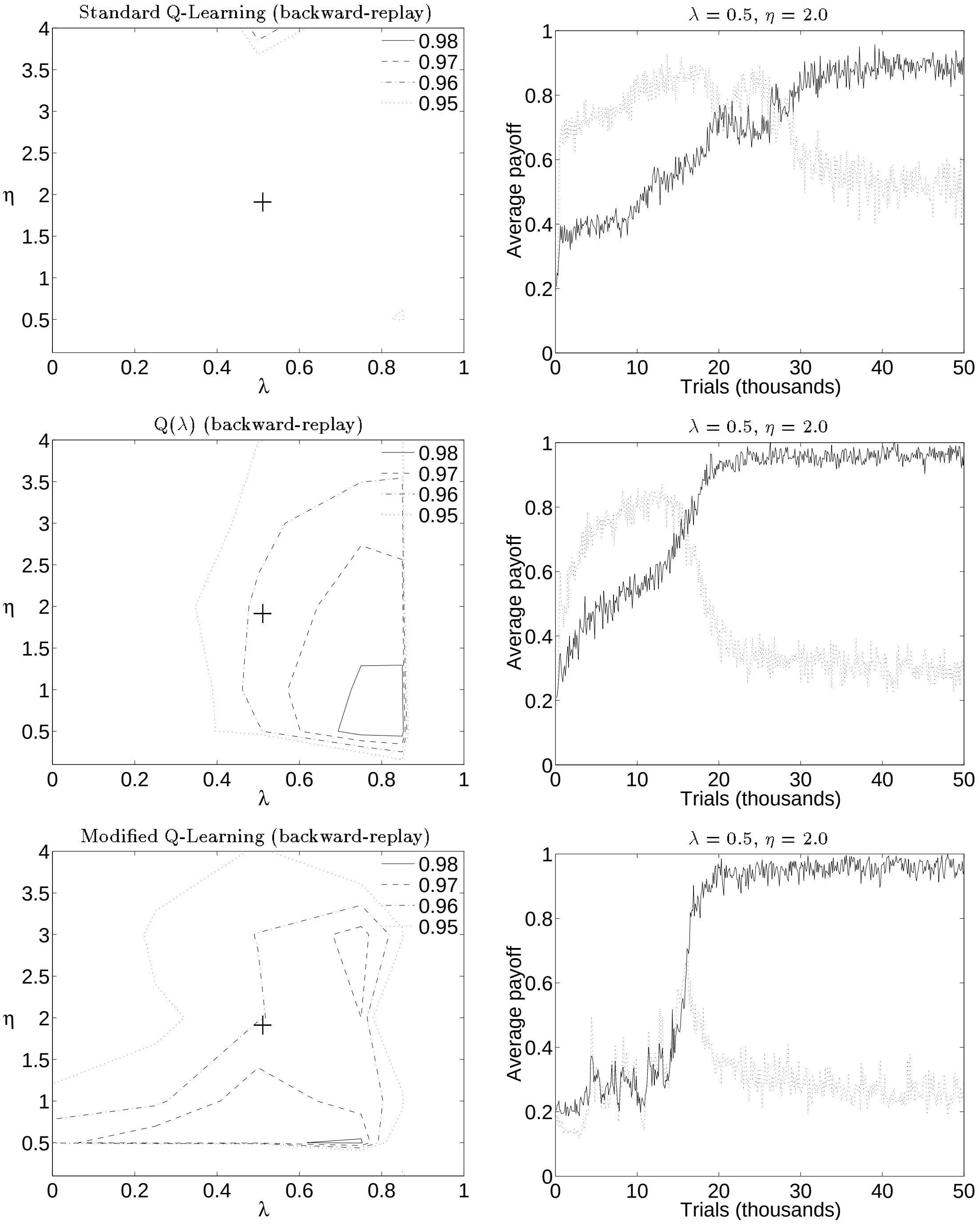


Figure 4.6: Backward-replay with damaged sensors. *Left:* Contour plots showing how the final payoff after 50,000 trials varies for each of the three update rules applied using different values of η and λ. *Right:* Sample training curves taken for each update rule, corresponding to the value of η and λ marked by a + on each contour plot. The dotted line is the normalised average number of steps taken in each trial (with 1.0 corresponding to 200 steps).

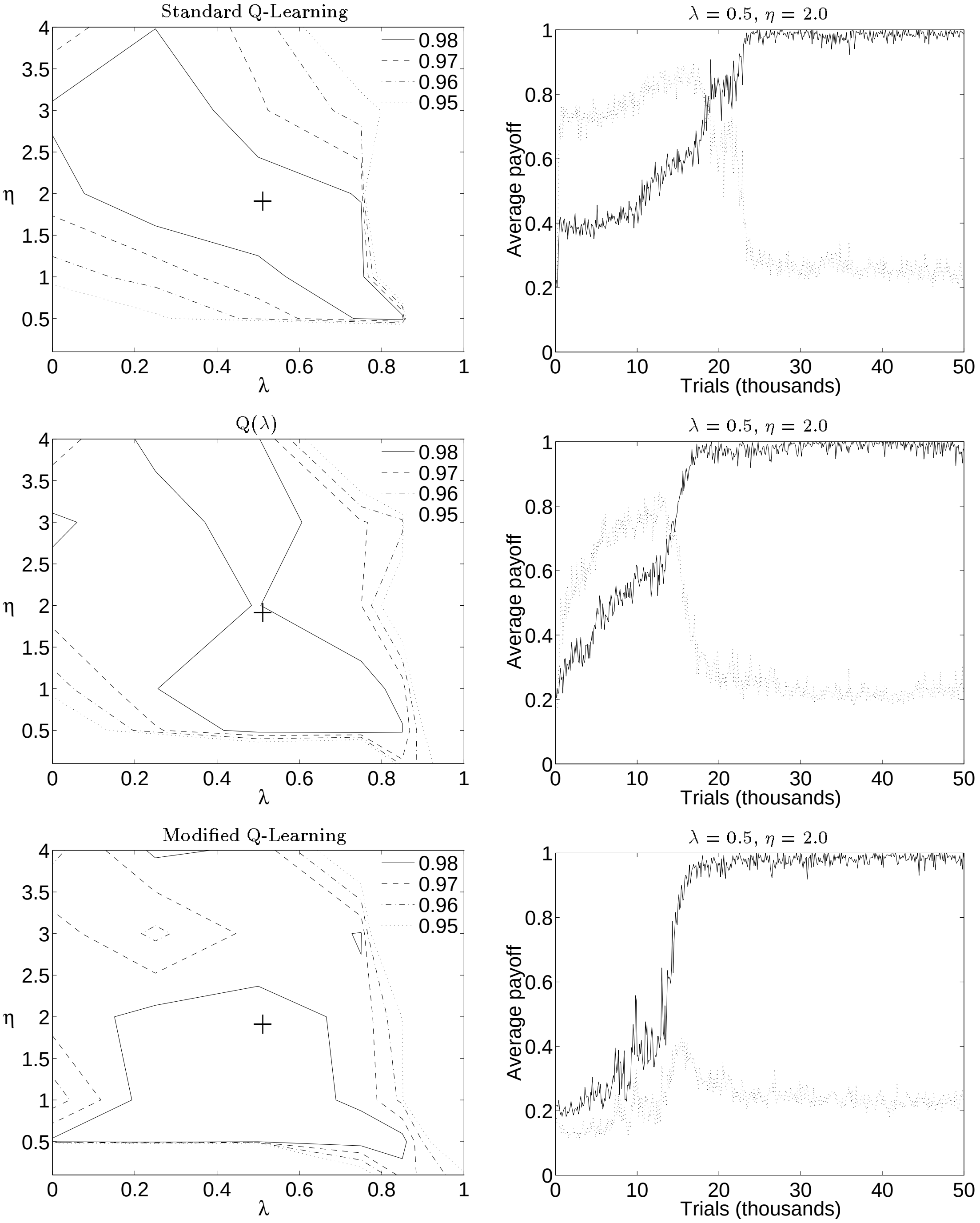


Figure 4.7: On-line updates with damaged sensors. *Left:* Contour plots showing how the final payoff after 50,000 trials varies for each of the three update rules applied *on-line* for different values of η and λ. *Right:* Sample training curves taken for each update rule, corresponding to the value of η and λ marked by a + on each contour plot. The dotted line is the normalised average number of steps taken in each trial (with 1.0 corresponding to 200 steps).

Goal angle sensor	Successful robots (from 36)	Updates taken (millions)	Trial length (steps)
Normal	27	1.2	31.1
Damaged	16	2.4	48.7

Table 4.5: Corrected gradient results. Summary of successful robots (those averaging greater than 0.98 average payoff over the last 5,000 training trials), from 36 different λ and η combinations using on-line Modified Q-Learning updates with 'corrected' output gradients.

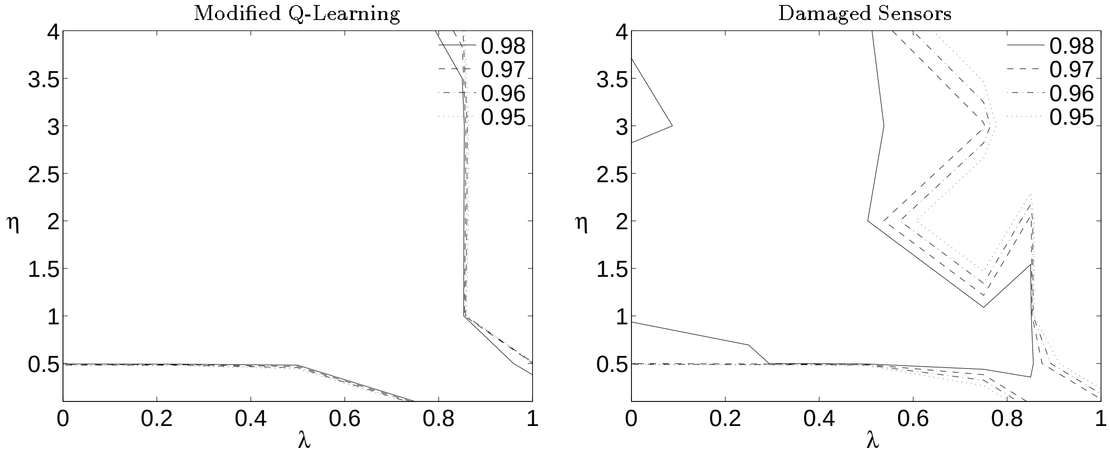


Figure 4.8: The contour plots for Modified Q-Learning when the corrected gradients are used. *Left:* The graph corresponding to the normal robot. *Right:* The graph corresponding to the robot with the damaged angle sensor.

and after updating will occur. This suggests that output correcting is mainly useful when high learning rates are being used.

The results for the robots learning with the correct input show no real differences when compared to those presented in Fig. 4.7 and Table 4.2, but then there was not much room for improvement.

From these results, it would seem that the correcting forward pass does provide some benefit, albeit at the expense of having to perform significantly extra computation at each time step. Whether the extra pass is worth doing depends on the size of the learning rate and how much improvement in performance is available.

4.4.3 Best Control Policy

In the last sections, it has been shown that the reinforcement learning methods using MLP networks can learn to achieve very nearly the maximum payoff for every trial. In fact, the level of average payoff received by the very best controllers translates into reaching the goal over 99% of the time. In this section, the quality of one such well trained controller is examined and compared with a system trained with the incorrect goal angle sensor values (section 4.4.1) to show the effect that this has and how the reinforcement learning system has learnt to overcome it.

The controller examined in this section comes from a robot that was trained on 30,000

Maximum trial length (steps)	Exploration value T	Average payoff	Goal	Crash	Timed-out
100	0.0	0.996	991	5	4
	0.01	0.988	972	12	16
200	0.0	0.996	991	5	4
	0.01	0.992	988	12	0
500	0.0	0.996	991	5	4
	0.01	0.992	988	12	0

Table 4.6: On-line Modified Q-Learning trained robot tested on 1000 randomly generated rooms, with varying maximum trial lengths, and with and without exploration.

Maximum trial length (steps)	Exploration value T	Average payoff	Goal	Crash	Timed-out
100	0.0	0.984	962	5	23
	0.01	0.981	960	9	21
200	0.0	0.991	980	5	15
	0.01	0.993	989	9	2
500	0.0	0.991	980	5	15
	0.01	0.993	990	9	1

Table 4.7: On-line Modified Q-Learning trained robot with damaged goal angle sensor tested on 1000 randomly generated rooms, with varying maximum trial lengths, and with and without exploration.

randomly generated rooms using on-line Modified Q-Learning with a training rate η of 2.0 and λ equal to 0.25. Fig. 4.9 shows a typical training room layout with the trajectory of the robot before and after training.

Table 4.6 summarises the performance of the trained robot on groups of 1000 trials when the maximum steps available per trial is altered, and also if a small amount of random exploration is available. These results show that the controller does not receive any benefit from performing occasional exploratory actions, which just result in increased numbers of crashes. It also shows that the final robot can either get to the goal, in which case less than 100 steps are required, or it cannot, in which case allowing extra steps before time-out is of no use whatsoever. Typically, this means that the robot has got caught in a loop, which is a consequence of its reactive controller which has no memory of whether a situation has already been visited during a trial (and thus the robot will choose the same action as it did last time).

In contrast, the robot with the damaged sensor *does* benefit from exploration, as can be seen in Table 4.7. In fact, the increased trial lengths allows more of the robots that are timing-out to reach the goal when exploration is allowed. With only 100 steps available to reach the goal position, the robot fails 2% of the time due to time-outs. This drops to 1.5% if 200 step trials are allowed, with no real improvement in performance for further increases in the number of steps allowed. However, comparing the final performance of this robot when using exploration with that of the robot with the correct angle sensor input (Table 4.6) reveals that their performance is almost identical in terms of allowing the robot to reach the goal. The difference is that the robot with the benefit of the correct angle sensor reaches the goal in an average of 25 steps, whilst the robot with the damaged sensor takes an average of 40 steps.

This can be seen in Fig. 4.10, which compares the trajectory chosen by the robot with the correct angle input with that of the robot with the damaged sensor on the same room. As can be seen, the robot with clear knowledge of the relative angle of the goal elects to turn around and so finds the gap between the obstacles. However, the robot with the damaged sensor ends up taking the long way round, as it has learnt to prefer to keep the goal on the left (where the angle inputs are correctly coded at all times).

4.4.4 New Environments

The last section showed the performance of a well trained robot when faced with room layouts of the kind which it experienced during training. As Fig. 4.9 and 4.10 show, these rooms have a fairly sparse layout of obstacles and often the robot can guide itself to the goal without encountering any intervening obstacles. However, the point of training the robot on randomised layouts is that it should learn a general policy. Therefore, it should be able to deal with more cluttered environments and unusual obstacle layouts than it has encountered during training.

In this section, this property is examined by seeing how the robot trained using on-line Modified Q-Learning (with the correct goal angle sensor) coped in a variety of more complex environments. The robot received no additional training to help deal with the new environments and had to rely on what it had learnt about avoiding obstacles from the training rooms.

Fig. 4.11 shows some hand constructed obstacle layouts, and the trajectories chosen by the robot when placed at a variety of different starting locations. The robot control policy copes with most of the situations with surprising ease, including finding its way into a concave enclosure (bottom right of Fig. 4.11). However, sometimes it does fail, as can be seen in the top right-hand room, in which one of the starting positions has resulted in the robot getting caught in a loop and failing to find the 'entrance' leading to the goal.

Some more rooms are shown in Fig. 4.12. The top two rooms involve many more obstacles than the training environments and a larger room size, yet the controller has no trouble dealing with these situations. The bottom two rooms involve concave structures, and whilst the robot copes with the bottom left example, the bottom right situation proves to be too difficult. Although it reaches the goal from certain starting positions, in others it simply gets caught in a loop in the concave corners of the obstacle.

Another test of the robot control policy shown in Fig. 4.13, where the obstacles are randomly generated circles. The 'circle world' generates rooms which are more cluttered than the training environment. Table 4.8 shows the performance of the system when presented with 1,000 randomly generated rooms of this type (the time-out occurs after 200 steps). The controller is able to cope with nearly as many of these types of room as with the normal training environments (compare with the results in Table 4.6). Also, again, a small amount of random exploration does not help the system.

Finally, in order to further demonstrate the robustness of the robot control policy, the case where the goal is moving is considered. This was achieved by using two robots, both using the same control policy. The *target* robot simply tries to get to a goal position as before. The *catcher* robot, however, has the target robot position as its goal and so must catch this robot in order to get its reward. If the target robot reaches its goal, it is

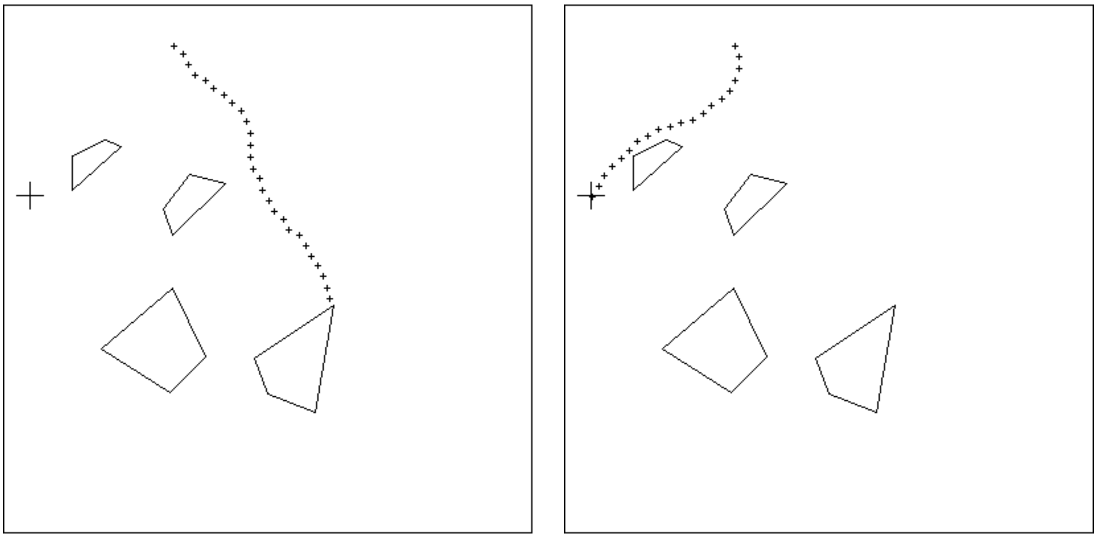


Figure 4.9: A robot shown in a typical training environment, before and after training using on-line Modified Q-Learning with $\lambda = 0.25$ and $\eta = 2.0$. The large cross marks the goal position. *Left:* Before training. *Right:* After training on 30,000 randomly generated rooms.

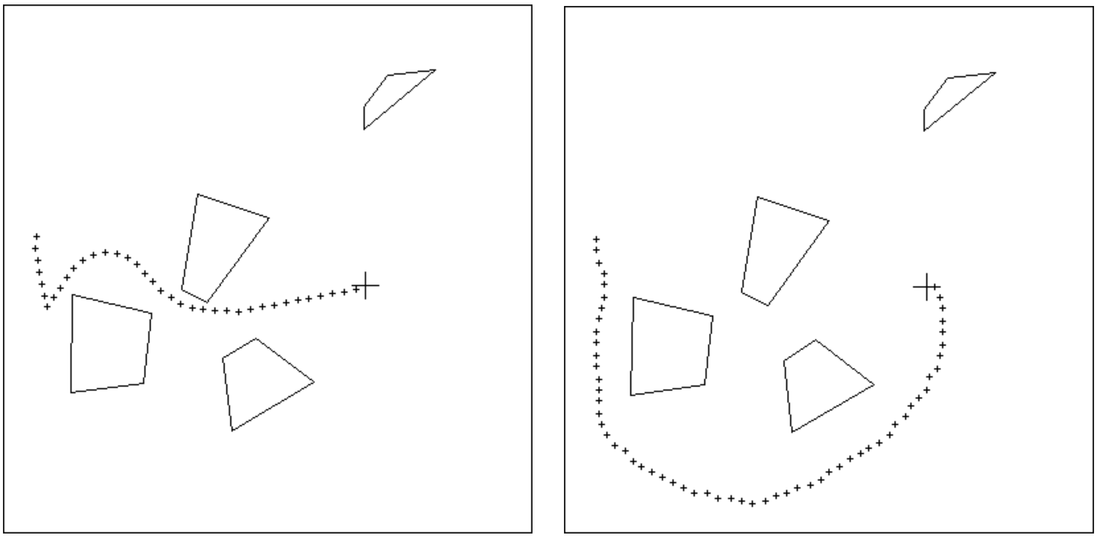


Figure 4.10: A comparison of the trajectory chosen by a robots with and without damaged sensors. Both were trained using on-line Modified Q-Learning with $\lambda = 0.25$ and $\eta = 2.0$. *Left:* The robot trained on the correct relative goal angle input. *Right:* One trained on the damaged input.

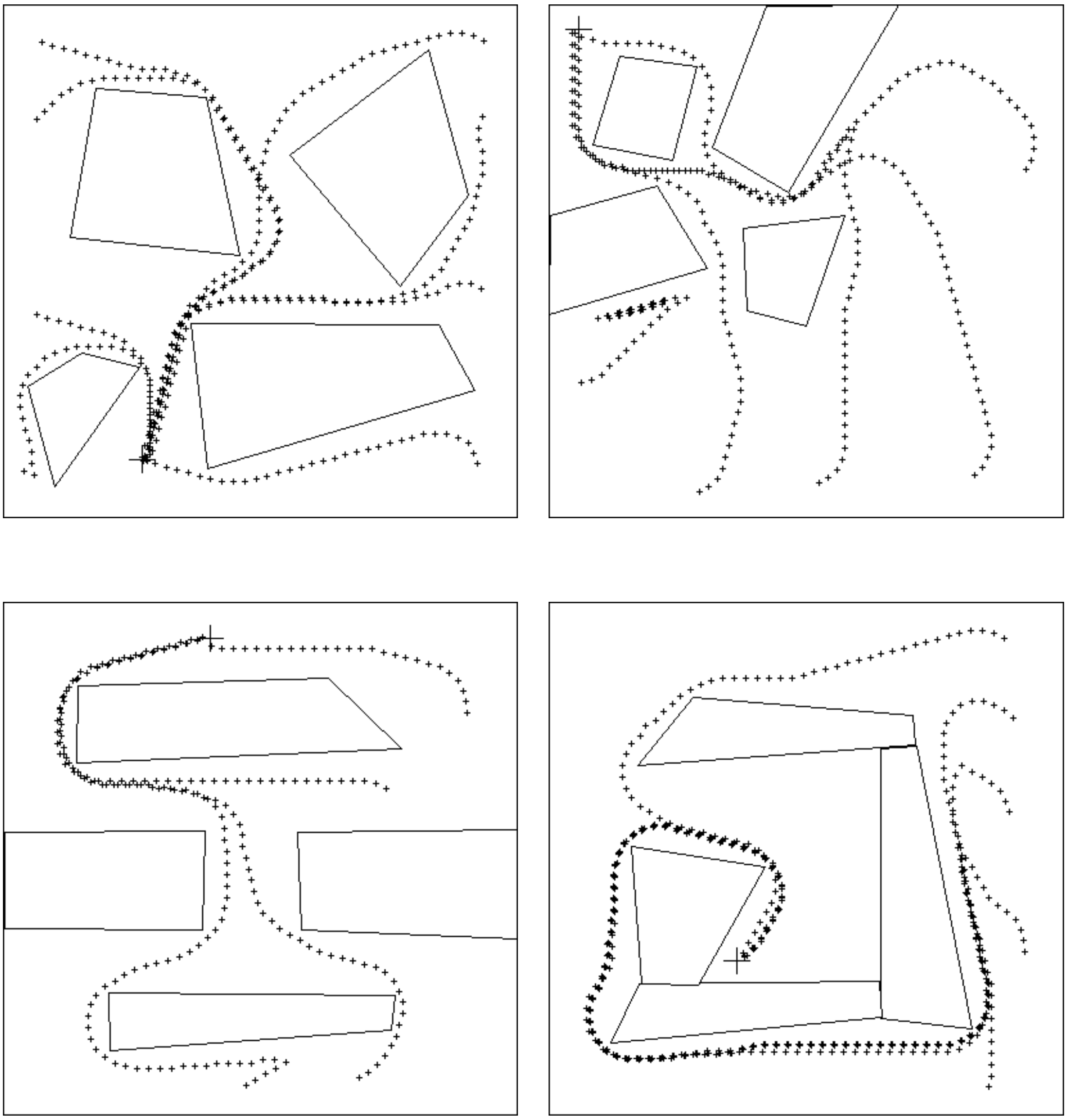


Figure 4.11: The on-line Modified Q-Learning robot trained with $\lambda = 0.25$ and $\eta = 2.0$ shown in a variety of hand constructed environments and with trajectories shown from a number of different starting locations to the goal position. The same robot controller is used in all cases and has had no training on these specific environments.

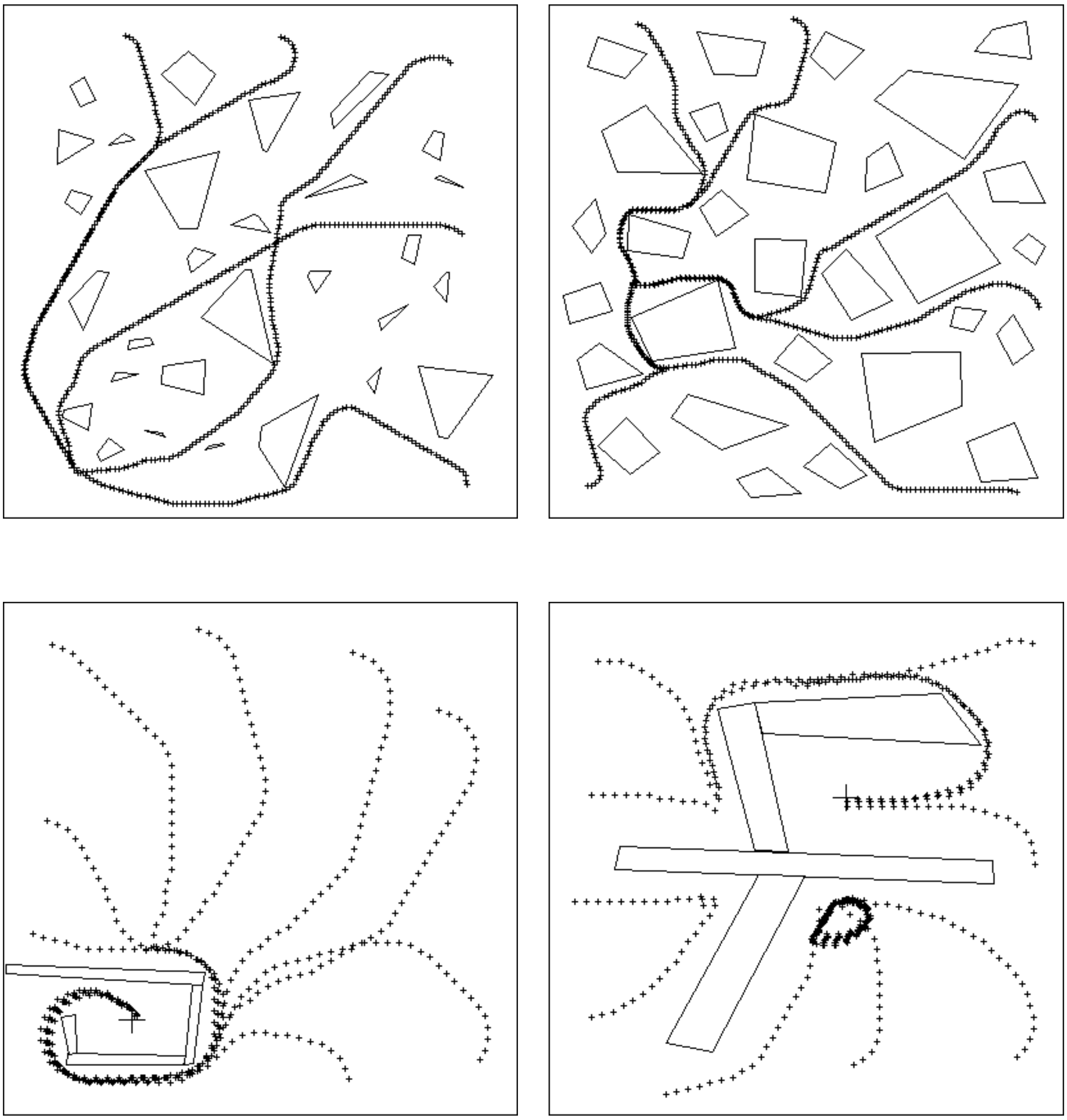


Figure 4.12: The same Modified Q-Learning trained robot shown in more unusual environments. *Top:* A room with perimeter 2.5 times bigger than seen before and increased obstacle densities. *Bottom:* The robot has only ever been trained on convex obstacles, but can cope with some concave situations, though it gets stuck in loops from certain starting positions (bottom right).

Exploration value	Average payoff	Goal	Crash	Timed-out
0.0	0.990	985	13	2
0.01	0.981	969	26	5

Table 4.8: Performance of the on-line Modified Q-Learning trained robot when faced with 1000 randomly generated circle worlds.

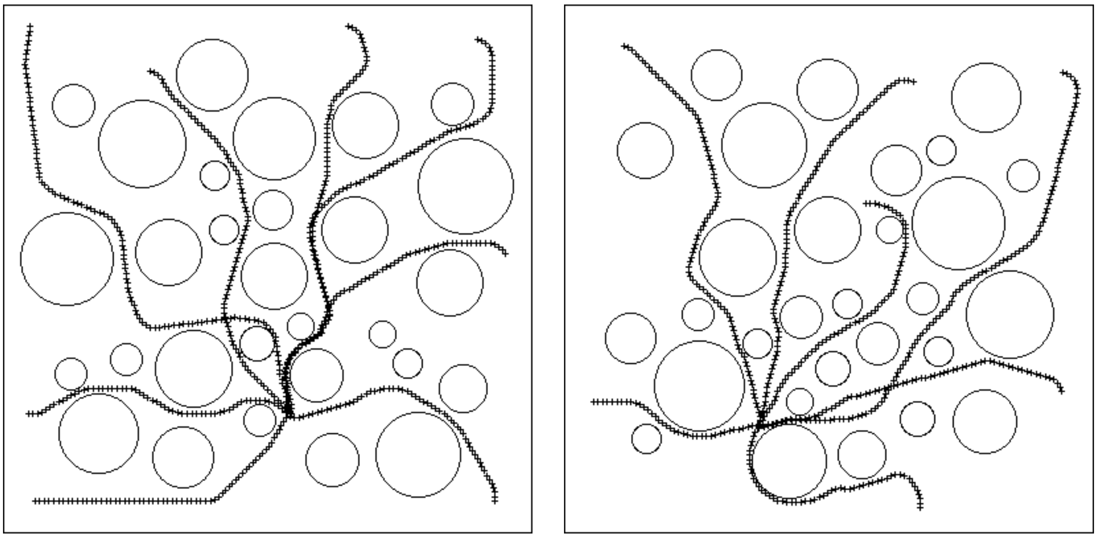


Figure 4.13: The robot presented with a novel environment consisting of randomly placed circular objects in a 2.5 times larger perimeter room than the the training rooms.

assigned a new one to keep it moving (hence the multiple goal crosses in the examples). Table 4.9 summarises the performance of the catcher robot as the relative speed of the target robot is reduced and Fig. 4.14 shows some sample trajectories of the two robots. Again, the robot control policy deals with this situation well, although the moving goal does result in more crashes (the robot gets close to obstacles and hits them as it turns towards the moving goal).

So, overall, the reactive robot controller learnt by the reinforcement learning system is remarkably robust and can cope with a wide range of situations. It also demonstrates that the training environment does not need to be as complex as the intended operating environment, as long as the system gets to experience the necessary range of situations to result in a general control policy.

4.5 Discussion of Results

The results presented over the last section have demonstrated that the reinforcement learning techniques can produce controllers for robot navigation that are highly robust. In this section, some of the issues surrounding the method are discussed.

Relative speed of target robot	Average payoff	Caught	Crash	Timed-out
1.0	0.978	963	37	0
0.9	0.981	970	30	0
0.7	0.985	974	24	2
0.5	0.992	984	13	3

Table 4.9: Results for 1000 trials of *catcher* robot as the relative speed of the target robot is varied. The maximum number of steps before time-out is 500.

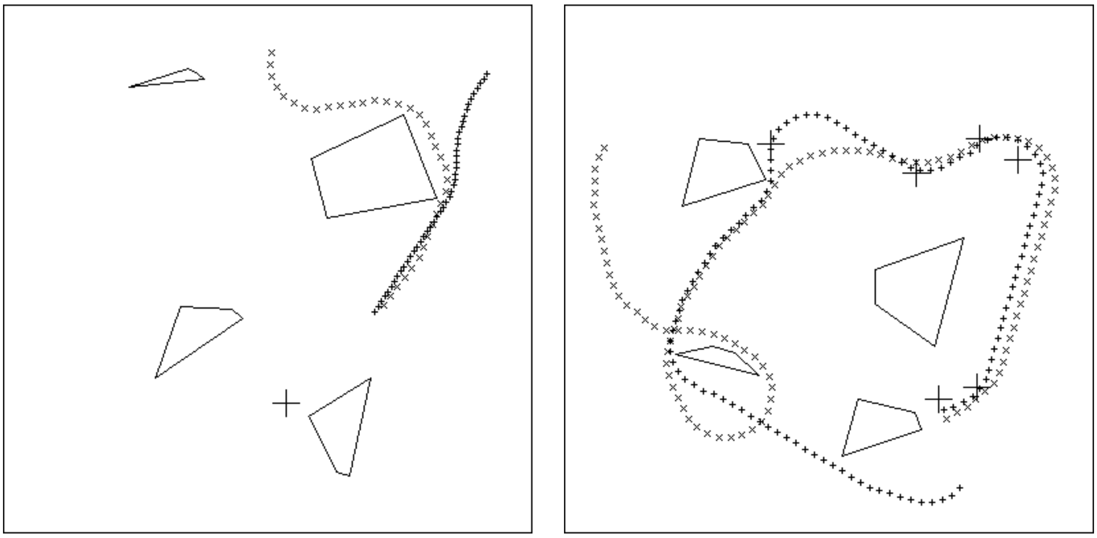


Figure 4.14: Sample trajectories of the two robots: +++ marks the trajectory of the target robot and × × × the trajectory of the catcher robot. *Left:* The target robot is travelling at half the speed of the catcher. *Right:* Example when the target is travelling at 9/10 of the speed of the catcher robot.

4.5.1 Policy Limitations

The robot has the capability to reach the goal on every trial, but fails to do so. This is caused by looping (the robot dodges an obstacle and ends up back at a point it visited before) and thus being timed-out, and occasional crashes with obstacles which occur due to the limited visual field of the robot range sensors. These problems are caused by the purely reactive behaviour of the robot — it has no memory of situations that have happened previously, or the number of steps it has taken. So, for example, a wall beside the robot that has fallen behind its forward sensor arc will not be remembered, and thus the robot may turn and hit it.

Therefore, the overall ability of the control system as presented is limited by being purely reactive. One method to produce a robot capable of dealing with more complex environments (such as non-convex obstacles and mazes), would be to use a more hierarchical approach. This would involve separate Q-learning modules being taught to deal with different tasks, and then training the system to choose between them based on the situation (Lin 1993a, Singh 1992).

4.5.2 Heuristic Parameters

There are several parameters that must be set in order to use the reinforcement learning methods presented in this chapter. η, λ, γ, and T must all be set, and poor choices can result in the system failing to converge to a satisfactory policy. The difficulty is that these values are all heuristic in nature and currently need to be selected based on rules of thumb rather than strict scientific methods.

The contour plots of Figs. 4.2, 4.3, 4.6, and 4.7 show how the choice of learning rate η and the TD-learning parameter λ can effect the subsequent success or failure of the system to converge to a successful solution. Some values simply result in very slow convergence times; others in complete failure to learn a successful policy. This is because of the generalisation property of MLPs, which means that information can be 'forgotten' as well as learnt. If the parameters chosen during training are unsuitable, the robot will forget information as fast as it learns it and so be unable to converge on a successful solution. This is why no proofs yet exist regarding the convergence of Q-learning or TD-algorithms for connectionist systems.

Consequently, it is desirable to use training methods that are less sensitive to the choice of training parameters, to avoid the need to perform repeated experiments to establish which values work best. The results presented in the last section suggest that on-line updates and the use of Modified Q-Learning or Q(λ), as opposed to standard Q-learning updates, help reduce this sensitivity.

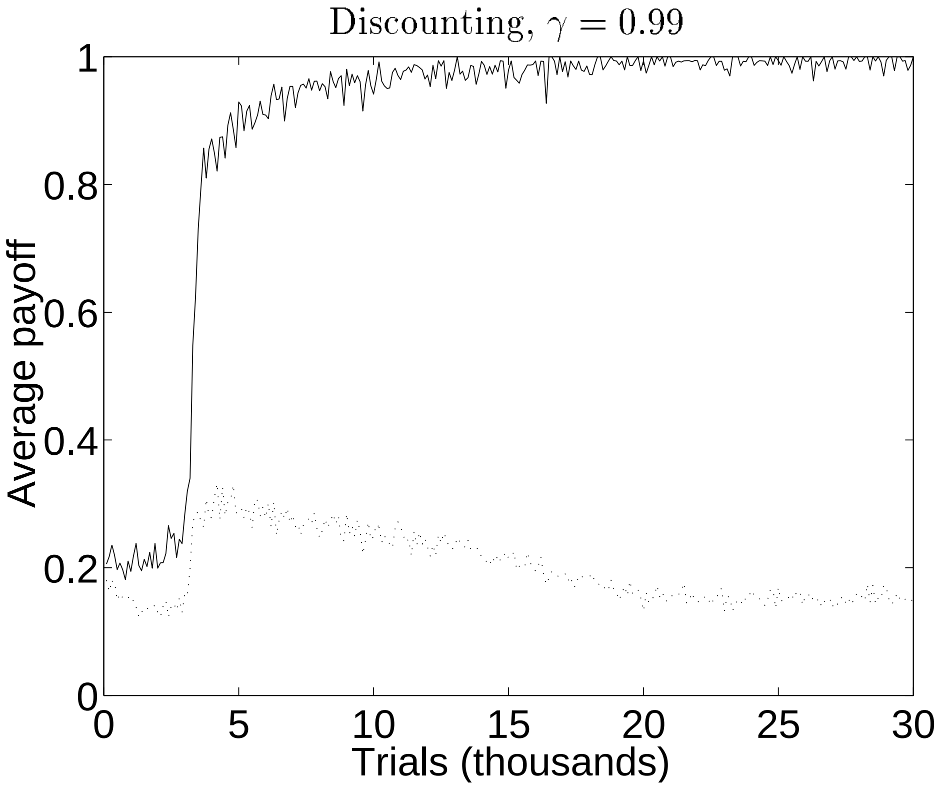
The value of the discount factor, γ, was fixed throughout the experiments presented at a value of 0.99. This was chosen so that the system would converge to solutions which used the fewest steps to reach the goal, but needed to be a value close to 1 in order that the discounted payoffs seen at states many steps from the goal would be a reasonable size. With no discounting, the robot can arrive at solutions that reap high final payoffs, but do not use efficient trajectories (and hence the robot is often timed-out). To illustrate this, Fig. 4.15 shows the training curves for two robots trained with on-line Modified Q-Learning with and without discounting. As can be seen, the undiscounted robot does considerably worse, especially in the average number of steps taken per trial, despite the fact that there is only a 1% difference in the updates being made at each time step.

Thrun and Schwartz (1993) provide limits for γ based on the trial length and number of actions available to a system, assuming one-step Q-learning is being used, but more general results are as yet unavailable. Also, an alternative to Q-learning called R-learning (Schwartz 1993) has been suggested, which eliminates the discount factor altogether by trying to learn undiscounted returns. However, results presented by Mahadevan (1994) showed that Q-learning outperformed R-learning in all the tasks he examined.

Finally, some experiments have shown that the convergence of the neural networks relies heavily on the exploration used at each stage of learning. If it is too low early on then the robot cannot find improved policies, whilst if it is too high at a later stage then the randomness interferes with the fine tuning required to have reliable goal reaching policies. When using a Boltzmann distribution, therefore, the rate of convergence is dependent on the rate of reduction of T.

4.5.3 On-line v Backward-Replay

The results of the tests for using on-line updating compared to backward-replay are interesting; on-line updating consistently performs more successfully over a wider range of training parameters for all 3 update rules, with the most marked difference in performance



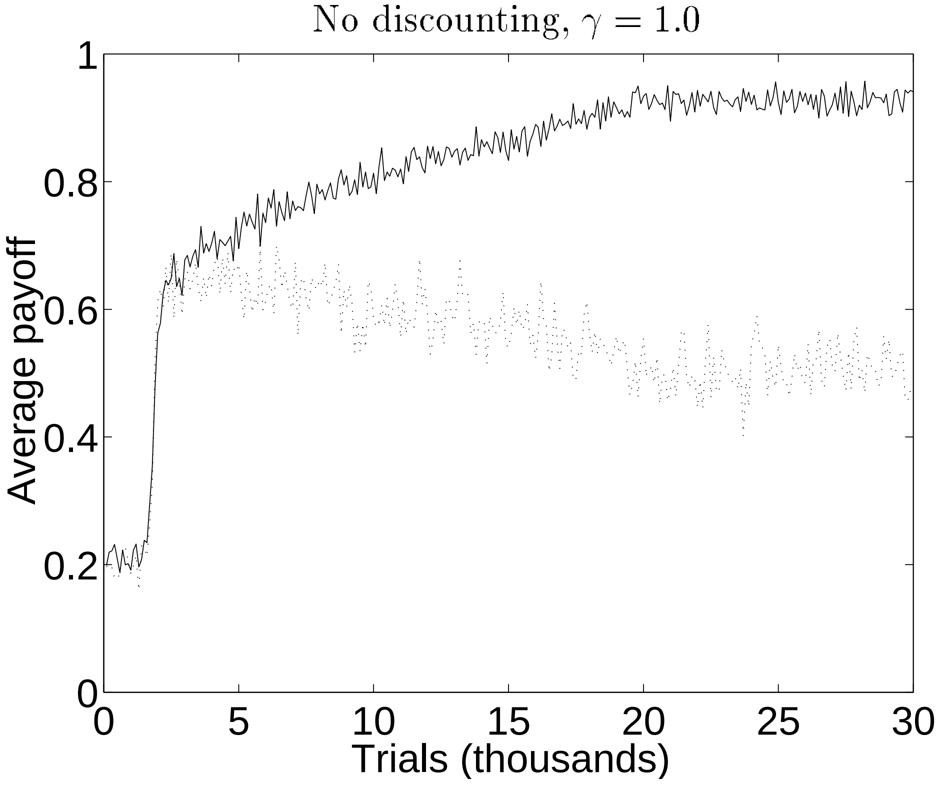


Figure 4.15: Graphs showing the effect of the discount factor γ. Both robots were trained using on-line Modified Q-Learning with $\eta = 2.0$ and $\lambda = 0.25$. The dotted lines show the normalised average number of steps per trial (with 1.0 corresponding to 200 steps).

for the series of experiments with the 'damaged' goal angle sensor. This is quite surprising, as backward-replay has the benefit of having all state-action pairs stored for the trial and updating using supervised learning based on the final payoff. However, it would seem that on-line learning has the advantage that the eligibilities act in a similar way to a momentum term,[6] providing updates that reduce the error to zero, instead of converging asymptotically in proportion to the mean squared error. Providing a momentum term for the updates performed during backward-replay could help achieve the same effect (as it does in normal supervised learning tasks), but would also introduce another parameter that would need to be experimented with during training.

4.5.4 Comparison of Update Rules

The results show the relative performance of the 3 different forms of update rule. It is important to remember that all 3 of these update rules are exactly equivalent when *purely* greedy actions are taken during a trial. The difference in the updates occurs only when exploratory actions are taken.

Using standard Q-learning with the eligibilities set to zero for non-policy actions, means the eligibilities are only allowed to build up when the robot takes a sequence of greedy policy actions. This stops the results of exploratory actions from being 'seen' by earlier actions, but also means that states see a continual over-estimation of the payoffs available, since they are always trained on the maximum *predicted* action value at each step (Thrun and Schwartz 1993). However, in a connectionist system, generalisation occurs, which means that the results of bad exploratory actions will effect nearby states even if the eligibilities are zeroed. So, this mechanism is of limited value, and simply results in the information gathered by good exploratory actions being used less effectively. The overall result is that standard Q-learning converges less quickly and over a smaller range of training parameters (especially noticeable with backward-replay on the damaged sensor problem; Fig. 4.6) than Modified Q-Learning or Q(λ) updates.

Q(λ) is in effect a combination of standard Q-learning and Modified Q-Learning, and this was shown by the way the training curve alters as the value of λ is increased (Fig. 4.5). However, the rule is slightly harder and more computationally expensive to implement than either standard Q-learning or Modified Q-Learning, and in these experiments appears to offer no advantage over just using Modified Q-Learning updates. In fact, by considering the number of updates required for training, and the quality of the solutions in terms of the average number of steps required to reach the goal, the performance of Q(λ) is actually worse. In addition, Q(λ) has the disadvantage of requiring the storage of the output gradients $\nabla_{\mathbf{w}} Q_t$ (section 3.3.3).

4.6 Summary

The on-line connectionist reinforcement learning algorithms have shown promise as methods of training systems in complex environments. In this chapter, the algorithms have been demonstrated on a mobile robot with limited sensory input and have shown that it can be trained to guide itself to a target position in a 2D environment whilst avoiding collisions with randomly positioned convex obstacles. Furthermore, it has been shown that this can be achieved without using *a-priori* knowledge to construct rules to restrict

[6] Or an integral term in a PID controller.

the policy learnt by the robot, but instead by allowing the robot to learn its own rules from an end-of-trial payoff using reinforcement learning.

Modified Q-Learning has been shown in chapter 2 to provide the most efficient updates for a discrete Markovian problem, and now in this chapter for a continuous state-space problem using neural network function approximators. Results presented in Tham (1994) demonstrate the performance of Modified Q-Learning updates when applied to a multi-linked robot arm problem, this time using CMACs as the function approximators. Q(λ) was again shown to provide similar convergence rates, but the final policies involved the arm taking a greater number of steps to reach the goal than those found using Modified Q-Learning updates.

Therefore, from the empirical evidence gathered so far, it would appear that Modified Q-Learning provides the one of the most efficient and reliable updating methods for learning the Q-function.

Chapter 5

Systems with Real-Valued Actions

In the previous chapters, reinforcement learning systems using Q-learning methods have been developed which can operate with continuous state-space inputs. However, these systems must still select actions from a discrete set, even though the optimal action at each time step may not come from the set A that is available. In such cases, it would seem logical to have some way of modifying the action set available in order to find the optimal control at each time step. In this chapter, some of the options available are discussed for modifying an *action function* $A(\mathbf{x})$ towards the optimal policy; in particular methods based Adaptive Heuristic Critic (AHC) reinforcement learning (section 1.3.6) are examined.

Normally, a single function approximator is used to produce a single action at each time step, which is modified as the system learns. However, a policy with discontinuities may be difficult to learn using a single continuous function approximator, and thus a method of combining real-valued AHC learning with Q-learning is proposed, called Q-AHC. This has the advantage that multiple function approximators may be able to represent the overall policy accurately, where a single function approximator is unable to.

A further subject examined in this chapter is that of *vector actions* i.e. actions with multiple components which need to be set at each time step. The tasks studied in previous chapters have involved actions with multiple components (e.g. the speed and steering angle for the robot in the Robot Problem; see section 4.3). However, the issue of how to select the values of the individual components has been avoided by selecting from the set of all possible action vectors. In this chapter, a number of alternatives for producing vector actions using the Q-AHC architecture are presented and their performance compared.

5.1 Methods for Real-Valued Learning

A common reinforcement learning method for what is known as *real-valued*[1] action learning is to use the AHC methods described in the introduction (section 1.3.6).

The idea behind AHC, or actor-critic, systems is for the actor to generate actions, and the critic to generate internal reinforcements to adjust the actor. This is achieved by learning the value function, $V(\mathbf{x})$, and using the TD-error that this produces at each time step as an internal reinforcement signal ε_t to adapt the action function, $A(\mathbf{x})$. Thus, the

[1] This is something of a misnomer, but comes from the time when many reinforcement learning problems produced binary outputs to select between pairs of actions.

internal payoff from the critic is equal to,

$$\varepsilon_t = r_t + \gamma V_{t+1} - V_t \tag{5.1}$$

If the return is lower than predicted, then the action that produced it will be punished, and if it is higher, then it will be rewarded. By adjusting the action function to favour producing actions that receive the best internal payoffs, the idea is that actor should converge to producing the optimal policy.

5.1.1 Stochastic Hill-climbing

The *stochastic learning automaton* (Narendra and Thathachar 1989) is a general term defined to represent an automaton that generates actions randomly from a probability distribution, and which receives reinforcement signals to adjust these probabilities. Williams (1988) extended this idea to the more useful *associative stochastic learning automaton* (ASLA) where the action probabilities are a function of the state $\mathbf{x}$. He introduced a class of methods called REINFORCE for updating the ASLA with respect to the immediate reinforcement signal, and Extended REINFORCE methods which use temporal difference learning to perform the updates. The basic operation of these rules is to adjust the ASLA probability distribution to favour actions that receive high payoffs.

The intention was that individual ASLA units could be connected together to form a neural network structure, and that the resulting network could then be trained using reinforcement learning. This is similar to the on-line MLP training algorithm examined in chapter 3, except that in the ASLA network the output of each unit is generated randomly from a probability distribution rather than deterministically from a fixed function.

The REINFORCE training methods are termed *stochastic hill-climbing* as the improvement to the output is made by trial and error — a random output is generated, and the probability function is adjusted to make it more or less likely to be generated again based on the payoff received. This mechanism can be used as part of an Adaptive Heuristic Critic system to produce a real-valued action function. An ASLA is used as the actor to generate an action, and then updated based on the internal payoff ε_t generated by the critic (Prescott 1993, Tham and Prager 1992).

Gaussian ASLA

The method studied in this chapter involves representing the action function as a Gaussian distribution, with mean $\mu(\mathbf{x}_t)$ and standard deviation $\sigma(\mathbf{x}_t)$. Therefore, the probability distribution $p(a|\mu_t, \sigma_t)$ is given by,

$$p(a|\mu_t, \sigma_t) = \frac{1}{\sqrt{2\pi\sigma_t^2}} \exp^{-\frac{(a-\mu_t)^2}{2\sigma_t^2}} \tag{5.2}$$

where a is the action, $\mu_t = \mu(\mathbf{x}_t)$, and $\sigma_t = \sigma(\mathbf{x}_t)$. At each time step, an action a_t is selected randomly from this distribution for the current state $\mathbf{x}_t$.

The question is how to alter the functions $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ in response to the internal payoff, ε. Williams (1988) suggested using the gradient of the log of the Gaussian distribution. In this case, if the functions are considered parametrised by internal values $\mathbf{w}$, a TD-learning update rule can be used,

$$\Delta \mathbf{w}_t = \eta_t \varepsilon_t \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{\mathbf{w}} \ln p(a_k|\mu_k, \sigma_k) \tag{5.3}$$

where for the mean the gradient is,

$$\nabla_{\mathbf{w}} \ln p(a_t|\mu_t, \sigma_t) = \frac{\partial \ln p(a_t|\mu_t, \sigma_t)}{\partial \mu_t} \nabla_{\mathbf{w}} \mu_t = \frac{a_t - \mu_t}{\sigma_t^2} \nabla_{\mathbf{w}} \mu_t \tag{5.4}$$

and for the standard deviation it is,

$$\nabla_{\mathbf{w}} \ln p(a_t|\mu_t, \sigma_t) = \frac{\partial \ln p(a_t|\mu_t, \sigma_t)}{\partial \mu_t} \nabla_{\mathbf{w}} \sigma_t = \frac{(a_t - \mu_t)^2 - \sigma_t^2}{\sigma_t^3} \nabla_{\mathbf{w}} \sigma_t \tag{5.5}$$

This means the functions are altered to increase or decrease the log likelihood of the action depending on whether the action led to an increase or decrease in the predicted payoff. This is elegant since exploration is automatically taken care of by the size of the standard deviation parameter. As the system improves its predictions, the standard deviation will be reduced, and thus the system will converge to performing the mean action at all times (unless higher payoffs can be achieved for retaining a certain level of randomness).

However, experiments exposed a problem with using these gradients — namely that as the standard deviation σ_t reduces, the size of the gradients increases, due to the effect of dividing by second and third powers of σ_t in equations 5.4 and 5.5. Therefore, the gradients can potentially be of the order of $1/\sigma_t$. When the standard deviation is very low (and hence there is very little deviation from the mean action μ_t), the gradients and resulting changes to the parameters of the function approximators can be huge, which is likely to destroy the currently learnt function mapping.

However, the gradients only give a *direction* in which changes can be expected to produce improvements. By considering equation 5.4, it can be seen that the gradient for μ_t would be more useful if the $1/\sigma_t^2$ was dropped, as would the gradient for σ_t if the $1/\sigma_t^3$ was dropped in equation 5.5. For the mean this is equivalent to using a learning rate $\eta_t = \eta_\mu \sigma_t^2$ and for the standard deviation $\eta_\sigma \sigma_t^3$ where η_μ and η_σ are constants.

Stochastic Real-Valued Units

Gullapalli et al. (1994) suggested a similar method to the stochastic hill-climbing ASLA, called the *Stochastic Real-Valued* unit (SRV). This too seeks to represent the current action function using a mean and standard deviation to define a Gaussian distribution. The difference from the Gaussian ASLA is that the standard deviation σ_t is produced by a function based on the current prediction of return i.e. $\sigma_t = \sigma(V_t)$. The idea is that the standard deviation should be high when the predicted return is low, and vice versa. In this way, more exploration occurs when the expected return is low, but as it rises the system will be more inclined to follow the mean action, which should be converging to the optimal policy.

Whilst this appears intuitively sensible, the designer is immediately faced with the problem of exactly what function to use to generate σ_t. How much exploration should be allowed when the expected return is low? How quickly should the standard deviation be reduced as the return rises to ensure convergence? It becomes clear that the design of the function for generating σ_t is very problem specific, as it depends on the size of the payoffs available and how they are awarded.

5.1.2 Forward Modelling

Forward modelling generally implies learning a world model such that the next state can be predicted from the current state and action. In the work of Jordan and Jacobs (1990), this

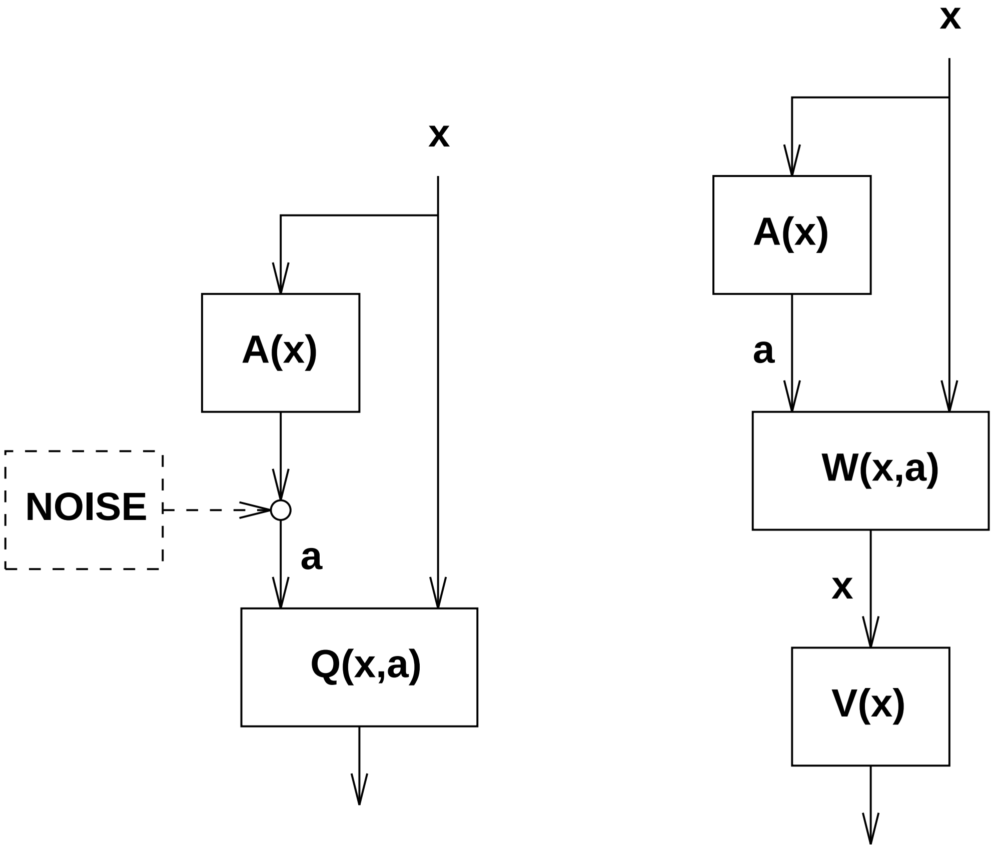


Figure 5.1: *Left:* Jordan and Jacob's Forward Model architecture interpreted as a real-valued Q-learning system. *Right:* Brody's extension using a world model.

is taken a step further; the system attempts to learn to predict the expected return rather than the next state $\mathbf{x}_{t+1}$. This method is equivalent to learning a Q-function, $Q(\mathbf{x}, a)$, where a is an input (see Fig. 5.1).

The action a is produced by an action function $A(\mathbf{x})$, and the Q-function is represented by a differentiable function approximator. Thus, the idea is to calculate the output gradient $\partial Q(\mathbf{x}_t, a_t)/\partial a_t$ from the Q-function, and use this to change the output of the action function by gradient ascent. However, this idea has a flaw, in that the only input to the system as a whole is the current state $\mathbf{x}_t$, and thus $A(\mathbf{x})$ and $Q(\mathbf{x}, a)$ can be considered as a single function of $\mathbf{x}$. This is more clearly seen from the left-hand block diagram in Fig. 5.1. In fact, $Q(\mathbf{x}, a)$ is really just learning the value function $V(\mathbf{x})$ for the current policy of $A(\mathbf{x})$, as only one action is performed in each state $\mathbf{x}$. Therefore, only the component of the gradient along the trajectory of $\mathbf{x}$ will be valid; the other components are undefined, so the gradient will not necessarily provide useful updates.

The solution is to decouple $A(\mathbf{x})$ from $Q(\mathbf{x}, a)$ by adding noise to the action output (as shown by the dashed box in Fig. 5.1), which provides the necessary exploration to learn $Q(\mathbf{x}, a)$ properly. This makes the system very similar to a real-valued AHC system, except that the changes in the action function are made using the gradients passed back by the Q-function, rather than using the TD-errors in the predictions made by a value function.

An alternative solution (Brody 1992), is the architecture shown on the right of Fig. 5.1.

In this, a world model is learnt first, and then its output is fed into a value function. In this way, the error in the value function gives an indication of the required change in $\mathbf{x}_{t+1}$, which in turn can be passed back through the world model to give the required change in the action a_t. This architecture is not entirely satisfactory, since it requires learning a full world model, which the methods investigated in this thesis have tried to avoid.

5.2 The Q-AHC Architecture

In this section, the ideas of Q-learning, where the system selects between multiple actions; and AHC learning, where an action function is adaptively changed; are brought together. By combining the two methods, the Q-learning limitation of selecting from a *fixed* set of actions is eliminated, whilst the problem of trying to represent a complex policy using a single function approximator when using real-valued AHC methods is also removed. This leads to a potentially very powerful architecture, which will be called Q-AHC learning.

The concept that the actions selected between by Q-learning do not have to come from a fixed action set A is as old as Q-learning itself. Watkins (1989) talked about hierarchies of Q-learning systems, where the actions selected between by a *master* Q-learning element would themselves be produced by other Q-learning elements. This idea was later implemented by Lin (1993a), where Q-learning elements were used to learn a set of *skills*, $s \in S$, which were then selected between by a master element which learnt $Q(\mathbf{x}, s)$.

A similar type of system, called the CQ-L architecture, was proposed by Singh (1992) (and later extended by Tham (1994)). This involves several separate Q-learning elements which are selected between by a gating element (which performs a similar role to a master Q-learning element). The CQ-L system is designed primarily to be used for sequential decision tasks, where each Q-learning element is used in sequence in order to achieve an overall task. However, in essence it is the same as hierarchical Q-learning.

5.2.1 Q-AHC Learning

The combination of the Q-learning and AHC methods is quite straightforward. Initially, the case is considered where there is only a single component to the action (i.e. it is not a vector), but in the next section a number of different architectures for producing vector actions are discussed.

The Q-AHC has a hierarchical architecture, with a Q-learning system as the top level which selects between a set of lower level real-valued AHC elements. The selected AHC element is responsible for generating the action used by the system. This is shown in Figure 5.2 where each action value $Q(\mathbf{x}, A)$ is shown connecting to the AHC element $A(\mathbf{x})$ that will be used if it is selected. Instead of each AHC element maintaining a value function to generate internal payoffs, the TD-error used to update the Q-function is used, e.g. for Modified Q-Learning updates,

$$\varepsilon_t = r_t + \gamma Q_{t+1} - Q_t \tag{5.6}$$

The main difference from the AHC learning algorithm described earlier is that the output gradients are only calculated with respect to the μ_t and σ_t of the selected AHC element. In other words, only the eligibilities of the action function that generated the action a_t are increased. However, the TD-error produced by the error between successive Q-function predictions is used to update *all* the AHC units and *all* the action value predictors,

$$\Delta \mathbf{w}_t = \eta_t \varepsilon_t \mathbf{e}_t \tag{5.7}$$

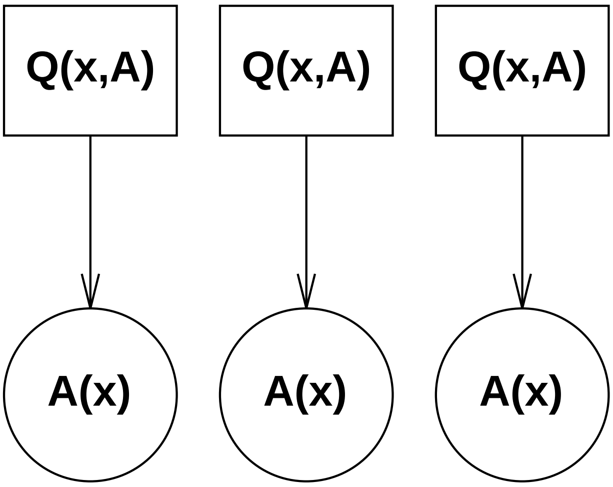


Figure 5.2: The basic Q-AHC architecture. The circles represent the AHC modules which are selected when the action value linked to them by an arrow is selected.

As with on-line Q-learning, it is the eligibilities $\mathbf{e}_t$ that determine the extent to which the individual parameters $\mathbf{w}_t$ are updated in response to the TD-error.

Of course, there is also a choice to be made of which update rule is used for the Q-learning part of the system. This can be any of those discussed in chapter 2. In the experiments presented later in this chapter Modified Q-Learning (see section 2.2.2) is used.

5.3 Vector Action Learning

The immediate problem faced in applying adaptive action function methods to a task like the Robot Problem (chapter 4) is the fact that the action is actually a vector. For example, in the Robot Problem, the action has two components, the steering angle and the speed. Thus methods are needed to determine how to alter each action component in response to the scalar internal payoff ε. Previously, this problem was avoided because each action value was associated with a separate fixed action vector.

With an AHC system, the obvious choice is to learn an overall value function, $V(\mathbf{x})$, and use the TD-errors produced by this to update all of the action elements (Tham and Prager 1992, Cichosz 1994). This ignores the *structural credit assignment* problem, which is to take into account the contributions of the individual elements to the TD-error. For example, in the Robot Problem, the robot might be heading for a collision with an obstacle and the selected angle component is to turn sharply (good idea), whilst the selected speed component is to travel at top speed (bad idea). If in consequence the robot crashes, then it will really be the fault of the speed element. However, in the above formulation, both elements will see the same internal payoff. It is questionable how much of a problem this is; good action choices by individual elements will generally see higher average payoffs than bad choices, as they contribute positively to the overall quality of the action vector. However, the lack of explicit structural credit assignment may increase the convergence times.

5.3.1 Q-AHC with Vector Actions

If the action to be performed at each step is in fact a vector of individual scalar actions, then their are more choices to be made when it comes to implementing a Q-AHC system. Fig. 5.3 shows 3 possible architectures for a system with two components to its actions.

The first architecture, Separate Q-AHC, involves treating the components of the actions as completely independent. Therefore independent Q-AHC learning elements are used to select each component of the action. The difficulty is that each action component is selected taking no account of the values selected for the other components. This makes it harder for each Q-AHC element to predict the expected return for their action component, because the effect of the other action components cannot be taken into consideration when making the prediction.

The second architecture, Combined Q-AHC, allows each action value to correspond to a particular combination of the AHC elements. This is similar to the fixed action combinations used by the Q-learning systems for the Robot Problem in chapter 4, where the 6 actions were made up of all combinations of the 3 fixed angles and 2 speeds. The problem with the Combined Q-AHC architecture is that each AHC element will see internal payoffs generated from their inclusion in different action vectors. These payoffs may conflict — e.g. one vector action is very useful, another is not, hence any AHC element that contributes to both will receive conflicting signals depending on which action vector was selected. However, much of this problem should be absorbed by the Q-function, which will learn to assign the poor action vector a low action value and so not select it very often.

The final architecture, Grouped Q-AHC, simply involves having a separate action vector of AHC elements associated with each action value. This would appear to be the most satisfactory architecture, as it reduces all action component interference problems. However, there still remains the same problem of structural credit assignment associated with training vector AHC learning systems (see section 5.3).

It should also be noted that the first architecture allows the greatest number of action combinations per element (both Q-function and AHC), whilst the last architecture allows the least. This is because the last system has a set of action vectors that are completely independent, whereas the other two rely on combinations of AHC elements.

5.4 Experiments using Real-Valued Methods

In this section, some of the real-valued action methods discussed in the previous sections are examined by testing them on the Robot Problem introduced in chapter 4.

The real-valued action systems used in these experiments are considered for the case where MLP neural networks are used as the function approximators (chapter 3). In addition, the updating method used throughout is the on-line temporal difference algorithm as discussed in section 3.3.1. The real-valued systems use MLPs not only for the prediction of the return, but also for the action functions. For Gaussian ASLA action functions using stochastic hill-climbing techniques, this means that one output is required for the mean $\mu(\mathbf{x})$ and one for the variance $\sigma(\mathbf{x})$. In the following experiments, separate single output networks were used, rather than a single network with two outputs, to avoid the weights of hidden units receiving conflicting updates from the two outputs.

In the first experiments, the robot task is restricted to single action component selection, where the reinforcement learning system attempts to select the optimum steering angle whilst the speed of the robot remains constant. Hence, the quality of the optimal

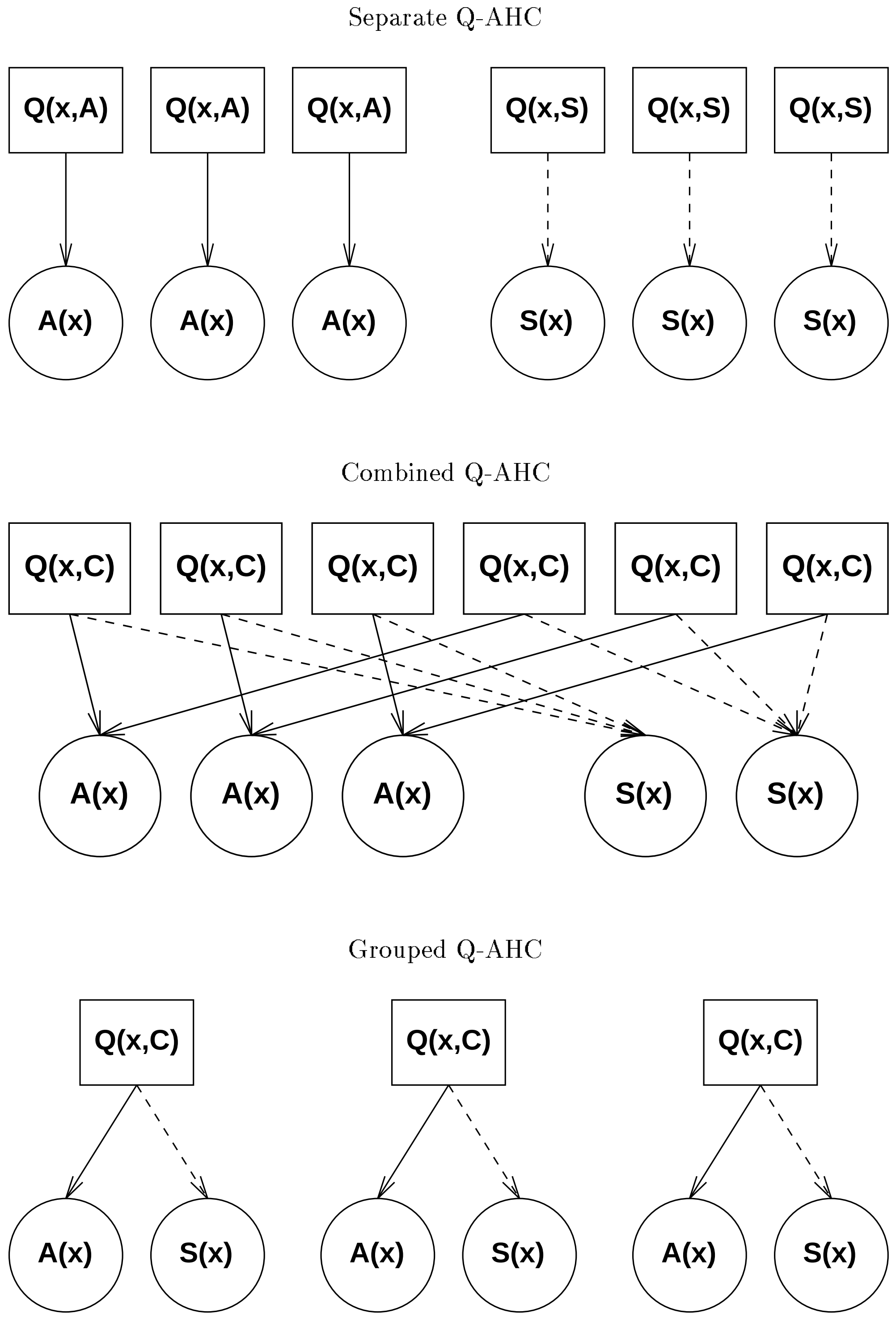


Figure 5.3: 3 alternatives for Q-AHC learning with multiple action components. In the above diagrams, the action vectors have 2 parts, $A(\mathrm{x})$ and $S(\mathrm{x})$, which correspond to the steering angle and speed AHC modules in the Robot Problem. Each is selected when the Q-value linked to them by an arrow is selected.

policy is restricted, as there are some starting situations in which the robot has no way of avoiding a collision due to the radius of its turning circle. However, this task is considered first since it does not involve the complications introduced by vector actions.

In subsequent sections, the systems are tested on the full Robot Problem, as tackled in the previous chapter by the Q-learning systems. For this task the reinforcement learning system must set the speed as well as the steering angle, and so requires the use of the different Q-AHC architectures introduced in section 5.3 for dealing with vector actions.

5.4.1 Choice of Real-Valued Action Function

The first question is what form of real-valued action function is best to use. Tests were carried out with a real-valued AHC system using Gaussian ASLAs, SRV units with a variety of different σ_t functions, and other methods including ASLAs with uniform distributions and σ_t values that reduced with time. The main conclusion drawn from these experiments was that the exact form of the real-valued AHC element was fairly irrelevant — several of the methods performed to a similar quality. The main criteria appeared to be that the standard deviation value σ_t should reduce as the expected return rose. In the end, therefore, the stochastic hill-climbing Gaussian ASLA as the action function was used for all AHC elements in the following experiments, with the 'boosted' learning rates to avoid the problem of over-large updates when σ_t was small,[2] exactly as presented in section 5.1.1.

5.4.2 Comparison of Q-learning, AHC, and Q-AHC Methods

The results presented in this section are for the single action robot problem where the system can only control its direction; the speed is fixed at the maximum. Results are presented to compare the performance of the following 3 algorithms:

- **Modified Q-Learning** (section 2.2.2)
- **Real-valued AHC learning** (section 5.1.1)
- **Q-AHC learning** with 3 AHC elements (section 5.2)

Each method was trained for the same 36 combinations of λ and η as used to test the Q-learning algorithms in chapter 4. The Q-AHC algorithm used Modified Q-Learning to update its Q-function and both Q-learning methods used the same exploration procedure and values as were used in the previous chapter (see section 4.3 for details). The actors of the AHC and Q-AHC methods used learning rates for the mean and standard deviation of $\eta_\mu = \eta_\sigma = \eta$, and the MLPs used for the $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ functions each had 3 hidden units. The real-valued steering angles were restricted to the range $[-15°, +15°]$.

All other aspects of the Robot Problem were exactly as used in chapter 4, including the payoff function and the sequence of randomly generated rooms used to train the robots. Each robot was trained on a sequence of 30,000 rooms.

The contour plots in Fig. 5.4 show the average end-of-trial payoff received by the robots over the last 5,000 trials of their training. Note that the contours on these graphs are for much lower payoff levels than those used on the Q-learning graphs in Fig. 4.2 and 4.3.

[2] In fact, without the boosted learning rates, the action function received such large weight updates that the network outputs were forced into saturation.

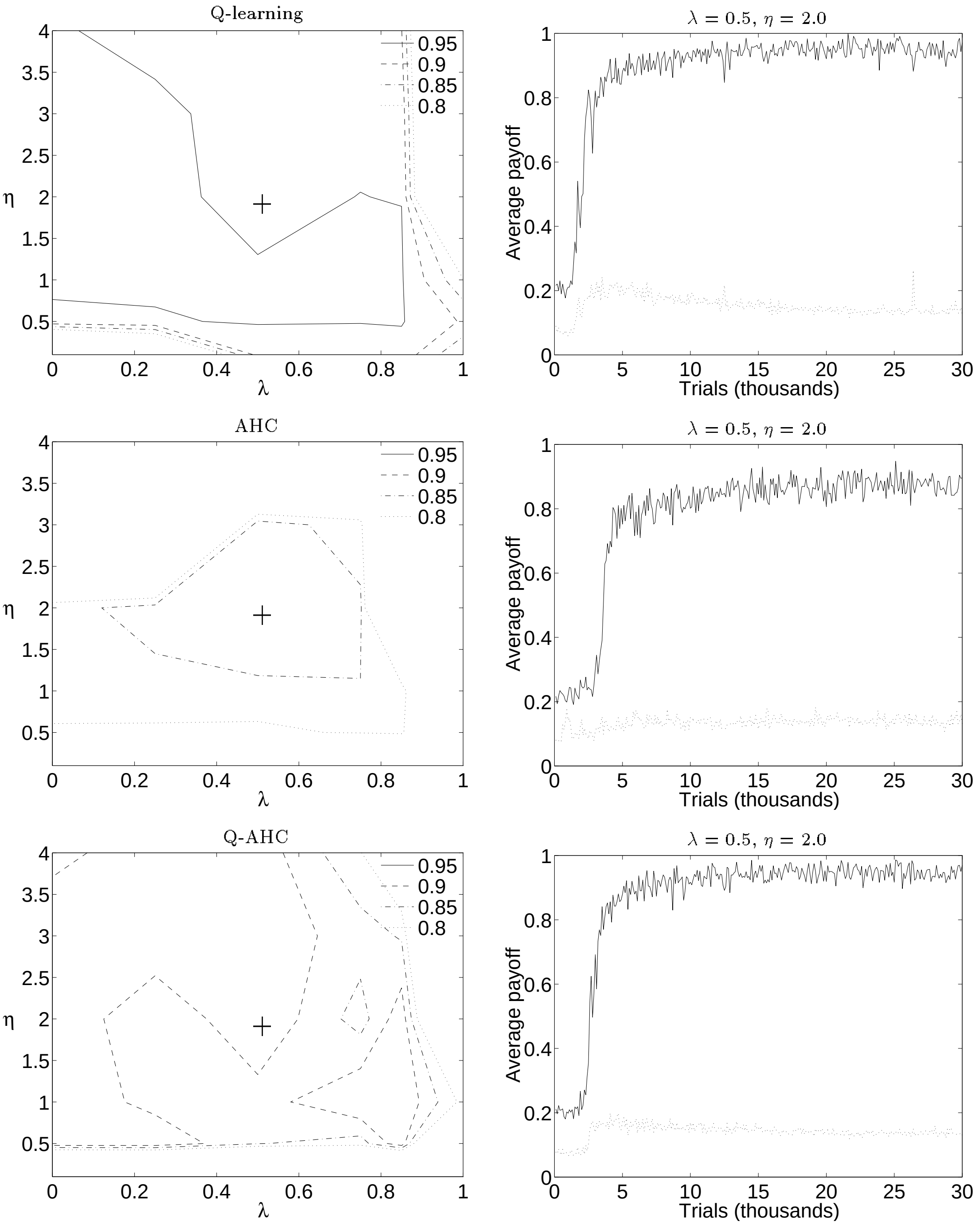


Figure 5.4: The contour plots for a wide range of λ and η parameters for Q-learning, real-valued AHC learning, and finally Q-AHC learning. The values represent the average payoffs received by each system over the final 5,000 trials of a 30,000 trial training run. *Note that the contour values are for considerably lower levels than those presented elsewhere in this thesis.*

As Fig. 5.4 shows, the best performance by far is achieved by the robots using Modified Q-Learning with a fixed action set. It is not necessarily surprising that the AHC robots did so badly, as it is possible that the policy required for success was difficult to approximate by a single small MLP, perhaps because it contained a large number of discontinuities (although experiments with networks with a larger number of hidden units did not perform any better). However, it is surprising that the Q-AHC robots, which have the potential to select actions such that they can perform exactly as the Q-learning system, did not do better.

In fact, an examination of the policies selected by the Q-AHC systems shows that for some values of the parameters η and λ, the system selects a single AHC element almost exclusively. For other values, however, the AHC elements have learnt to generate constant steering angles at opposite ends of the action range (meaning that two of the actions are at one end, and one at the other). The different actions produced are then selected between by the action selector exactly as in a fixed action set Q-learning system. Consequently, there appear to be two possible policy solutions that the system can arrive at.

Examining the systems that select one AHC element at all time steps reveals that the other two AHC elements have moved to generating actions at one end of the allowable range. In fact, the evolution of the systems that use a single AHC element appears to involve all of the AHC element outputs moving towards the same end of the action range, and thus one eventually winning out as providing the most successful policy. If instead the AHC element outputs diverge towards opposite ends of the action range, then the system discovers that a better policy is to select between them as for a fixed action set Q-learning system.

5.4.3 Comparison on the Vector Action Problem

In this section, results are presented for the real-valued reinforcement learning systems that set both the speed of the robot and the steering angle. The speed of the robot is limited to the range $[0,d]$, where d is a positive constant (see Appendix A).

The first contour plot, shown in Fig. 5.5, is for the plain real-valued vector AHC method (see section 5.3). The contour levels are the same as used in chapter 4 and so a direct comparison of performance can be made. The example training curve shows that the system has a noisy start, but then rises at around 15,000 trials to a respectable level of performance. However, as can be seen from the contour plot, it is one of the few robots to do this.

Fig. 5.6 shows the same data, this time plotted for the lower contour levels used for the plots in the previous section (Fig. 5.4). In addition, the example training curve is shown for a different setting of the learning rate η. It can be seen that the average payoff curve is still rising slowly at the end of the training run, and has not made the sudden jump that occurs in the example graph in Fig. 5.5. Also, interestingly, the Q-learning and Q-AHC systems trained in the last section on the steering angle only problem have achieved a better performance over a wider range of values than has been achieved by the vector action AHC method.

The remainder of the experiments in this section concentrate on the vector action Q-AHC methods introduced in section 5.3.1:

- **Separate Q-AHC:** Two separate Q-AHC learning elements were used, one for speed and one for steering angle. Each was made up from 3 AHC elements, and the two Q-learning elements learnt independently of one another.

Training method	Successful robots (from 36)	Updates taken (millions)	Trial length (steps)
Vector AHC	3	1.0	30.8
Separate Q-AHC	10	1.1	30.8
Combined Q-AHC	5	1.1	31.3
Grouped Q-AHC	16	1.1	31.5

Table 5.1: Comparison of vector action methods. The results are shown for successful robots averaging greater than 0.98 payoff over the last 5,000 trials for 36 different λ and η combinations.

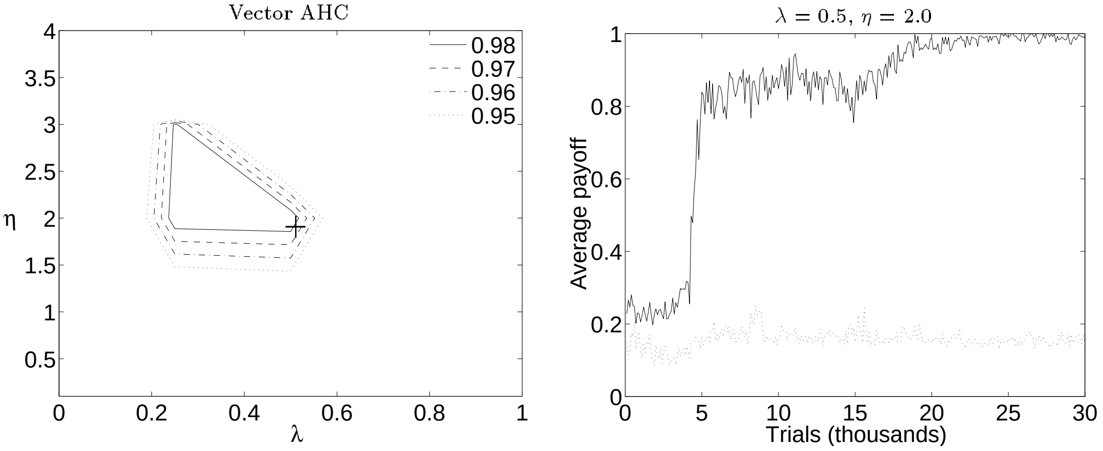


Figure 5.5: *Left:* The contour plot for the real-valued vector AHC learning system. *Right:* An example training curve taken at the point marked with a +.

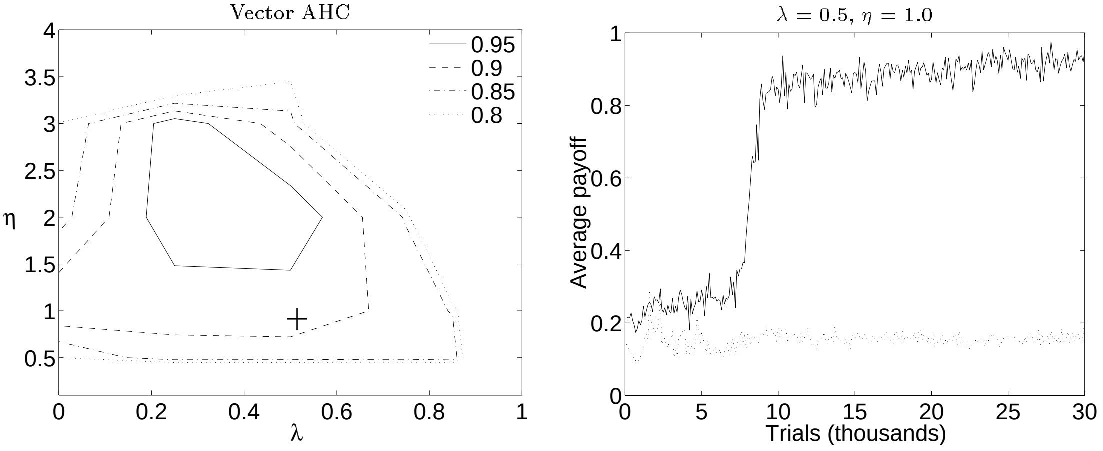


Figure 5.6: *Left:* The same contour plot for the real-valued vector AHC learning system when plotted against lower contour levels. *Right:* A different example training curve taken at the point marked with a +.

- **Combined Q-AHC:** The Q-learning element selected between speed-angle action pairs, which were made from the 6 combinations possible with 3 steering angle AHC elements and 2 speed AHC elements.

- **Grouped Q-AHC:** 3 speed-angle vector AHC elements were selected between by the Q-learning element.

In fact, the above architectures correspond exactly to those shown in Fig. 5.3. The sizes were chosen so that each of the architectures would use roughly the same number of neural networks. In fact, the Separate Q-AHC system needed 18 networks, Combined Q-AHC 16 networks, and Grouped Q-AHC 15 networks.

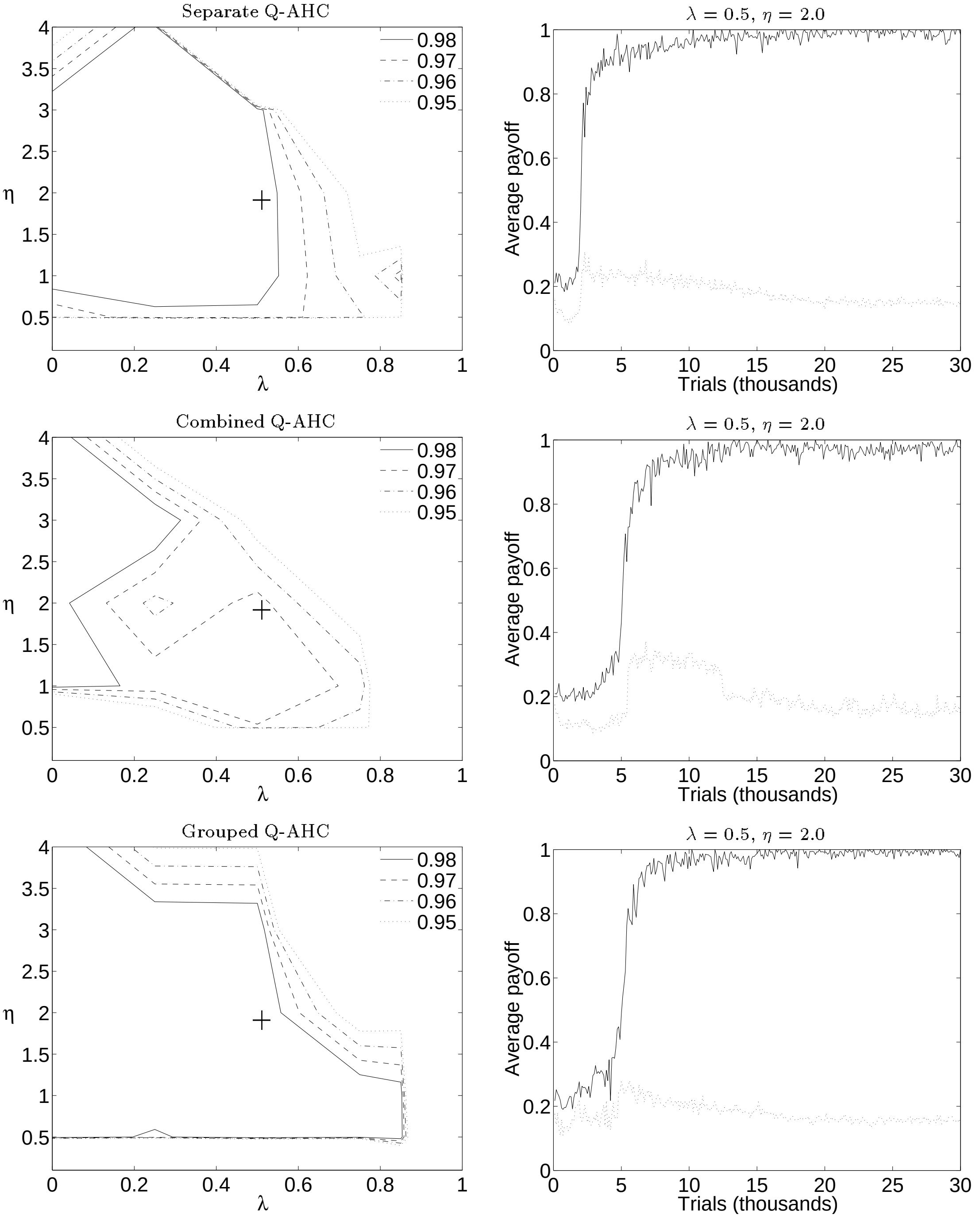
In all of the experiments presented, the Q-learning elements use the Boltzmann function to select between actions, exactly as in chapter 4. Modified Q-Learning is used for the update rule. The performance of the systems using other Q-function update rules was not examined.

The contour plots for the average payoffs received by the various architectures after 30,000 trials are shown in Fig. 5.7. It can be seen that the Grouped Q-AHC method works over the widest range of λ and η values, with Separate Q-AHC next, and Combined Q-AHC performing the worst. All of the methods outperform the vector AHC method, although the Combined Q-AHC method does not perform as well in the region where vector AHC does well. Table 5.1 summarises the performances of the different methods for the successful robots. As can be seen from this table, all the methods train using a relatively low number of updates.

The AHC elements in the Separate Q-AHC systems turn out to have each gravitated towards producing actions at the limits of the action ranges. The Q-learning elements then select between them based on the situation; choosing to travel at maximum speed most of the time unless an obstacle is in the way. This indicates that the speed and steering angle can be selected independently without too much difficulty.

Combined Q-AHC has the worst performance of the 3 architectures. This is unsurprising given the level of interference that occurs due to the AHC elements being used with different partners. The system tends towards selecting the same action value, and thus pair of AHC elements, at all times. Hence, the resulting performance is about the same as the vector AHC system examined at the start of this section. The difference is that it has several competing AHC elements to choose from and so appears to work over a wider range of η and λ parameter values than the single vector AHC learning system.

A close examination of the policies used by Grouped Q-AHC reveals that the individual action function pairs have gravitated towards producing actions at different limits of the action ranges, to produce 3 different fixed action vectors. The system then selects between them using Q-learning. For example, the Grouped Q-AHC learning system at $\lambda = 0.25$, $\eta = 2.0$ finishes the training run with the AHC pairs representing the following 3 vector actions: $(-\theta, d)$, $(+\theta, d)$, and $(+\theta, 0)$. Given that this set of 3 action vectors represents half of those available to the Q-learning systems examined in chapter 4, it is unsurprising that the overall quality of the policies found by this system are worse than the pure Q-learning systems (compare the contour plot of Fig. 5.7 with that of Modified Q-Learning in Fig. 4.3).



Separate Q-AHC

$\lambda = 0.5$, $\eta = 2.0$

Combined Q-AHC

$\lambda = 0.5$, $\eta = 2.0$

Grouped Q-AHC

$\lambda = 0.5$, $\eta = 2.0$

Figure 5.7: *Left:* The contour plots for the different real-valued vector action Q-AHC architectures. *Right:* Examples of training curves taken at the points marked with a +.

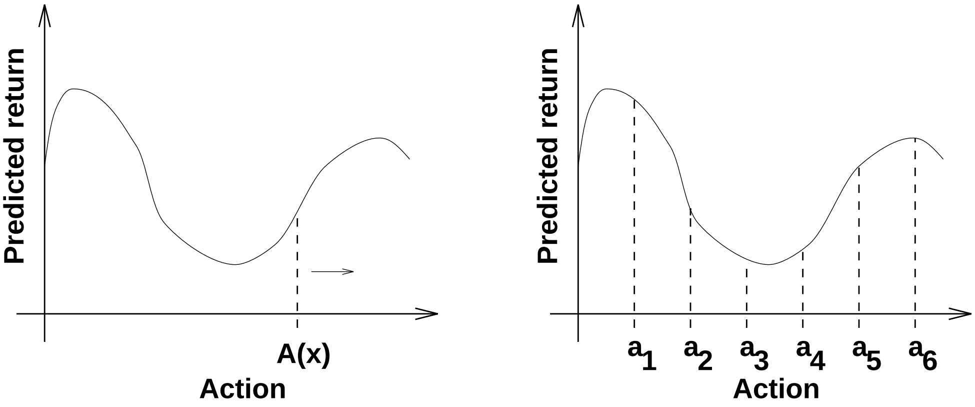


Figure 5.8: Selecting the optimal action in a state. *Left:* Hill-climbing will lead the action function, $A(\mathbf{x})$, towards a sub-optimal action. *Right:* The fixed action set, $\{a_1, \ldots, a_6\}$, ensures at least one action is *near* the optimum value.

5.5 Discussion of Results

The performance of the real-valued reinforcement learning methods examined over this chapter has been disappointing. None of the AHC or Q-AHC methods manage to do as well, over as wide a range of λ and η parameter values, as the Q-learning methods presented in the previous chapter. The direct comparison of methods on the task of setting only the steering angle (section 5.4.2) demonstrated that the Modified Q-Learning system selecting from a fixed set of actions was able to outperform both real-valued AHC and Q-AHC.

The overall performance of the Q-AHC systems is quite poor when compared with the standard Q-learning methods. However, it should be remembered that the Q-learning method involves the selection of a fixed action set *a-priori* by the designer of the system. For the robot navigation problem, choosing useful actions is not too difficult, but for some tasks, choosing an appropriate fixed action set could be more difficult. In this sense, the real-valued AHC architecture is more general, as it attempts to select the action appropriate to each state for itself. The Q-AHC systems have the best of both worlds; thus on the Robot Problem, the Q-AHC systems were able to achieve better policies than the real-valued AHC systems, because they could select between AHC elements that had gravitated towards the limits of the action ranges. The best vector action Q-AHC system was Grouped Q-AHC, which may have performed better had it been able to produce all 6 action combinations available to the fixed action set Q-learning systems in the previous chapter (it could only produce 3).

Examination of the policies used by the different systems reveals an interesting fact; the systems either end up selecting a single AHC element for the majority of the time, in which case the overall performance of the system is around that of a pure AHC system; or they chose between a selection of AHC elements which have learnt to produce widely differing actions, and thus behave and perform more like a pure Q-learning system. This property is explored further in the next section.

5.5.1 Searching the Action Space

In section 5.4.2 it was suggested that the reason why the real-valued AHC system did not perform as well as the Q-learning system could be because the small MLP network used for the action function was unable to approximate the optimal policy. However, experiments had already shown that this size of MLP was capable of learning a successful policy from the discrete actions generated by a system trained using Q-learning. In addition, it had been found that systems using networks with more hidden units did not perform any better.

Yet, when real-valued AHC systems were used to train the action function for the task of setting only the steering angle (section 5.4.2), even the best systems got to a level of average payoffs of around 0.88, and no further. This is in comparison with Q-learning systems which could reach average payoffs of 0.95 on the same task. As discussed in section 5.4.1, different forms of action function and learning parameters did not give any improvements over the Gaussian ASLA unit finally used.

The problem stems from the fact that the real-valued AHC systems studied in this chapter use a gradient ascent method (stochastic hill-climbing; section 5.1.1) to adjust the action function. Gradient ascent is best suited to searching smooth functions with a single maximum — in this situation, taking steps uphill will be guaranteed to lead to the optimal value. However, the predicted return across the action and policy space in each state may contain gradient discontinuities and/or multiple maxima. This is illustrated by the diagrams in Fig. 5.8.[3] Simply using gradient ascent may lead the action function, $A(\mathbf{x})$, in the direction of a local, but not global, maximum and thus the resulting policy would be sub-optimal.

Q-learning can avoid this due to its use of a fixed set of actions which learn action values independently of one another. The eventual greedy policy action will be the one that started closest to the global maximum (a_1 in Fig. 5.8).

With the Q-AHC system, the different action functions of the AHC elements can move towards different local maxima. The Q-learning action selector can then choose the action function that has moved to the global maximum. However, the difficulty which has been highlighted by the experiments in this chapter, is that the maximum each action function heads towards is random. Consequently, it is possible for all the action functions to head to the same maximum. If this happens, the best performance that can be achieved is the same as a pure AHC learning system with a single action function.

Possible ways to combat this effect and improve the performance of Q-AHC methods include:

- **Action Function Initialisation:** This involves setting the initial parameters $\mathbf{w}$ of the action functions so that they start in different parts of the action space. If multiple maxima exist, then the functions will hopefully start in different regions of attraction and so at least one will learn the action corresponding to the global maximum in each state.

  If MLPs are used to represent the action function, then each action output can be initialised by setting the output bias weight to the appropriate level. Tests of

[3] These diagrams are something of a simplification; in reality, the predictions of return are also a function of the policy, π, which changes with the action function. Thus the two maxima shown might exist at two separate policies — as the action function moves towards one it is actually moving towards a maximum in policy space as well as action space.

this idea for a Combined Q-AHC system resulted in systems that learnt to select between action pairs, rather than using a single pair exclusively as before, and hence which achieved a higher level of average payoff. To set the initial values for the action functions relied on the prior knowledge of which actions were best, and so was similar to starting with a fixed action set and then allowing the system to modify the values as it learnt.

- **Action Function Restarting:** When action functions share a maximum, only one ends up being selected, and so it would seem useful to re-use the other action function elements. Thus, if the usage of an action function falls below a certain threshold — e.g. not being selected as the greedy action for N successive time steps — then it could be *restarted* in a new region of the action space.[4]

  The difficulty with this idea is that the associated action value $Q(\mathbf{x}, a)$ for a restarted action function will be undefined. If it is set too low, then the newly reset action function will not be selected. If it is set too high, then the reset action function will get selected a great deal initially, but would then be judged on its starting performance, which would almost certainly be poor and result in a rapid drop in the associated action value. An alternative might therefore be to add a *selection bonus* to the restarted action value, and then gradually reduce it over a number of trials. The aim would be to encourage the system to select the restarted action function whilst the corresponding action value learnt its correct level.

- **Action Range Restrictions:** The action functions could be restricted to producing actions in separate regions of the action range. This is similar to initialising the action functions to start in difference parts of the action space, except they would never be able to accumulate at the same maximum, since each would be confined to a particular region. The result would be like using a normal Q-learning system with a fixed action set, except that the actions would be able to shift around a little towards the global optimum of their region.

5.6 Summary

In this chapter, methods of producing real-valued actions in an on-line reinforcement learning system have been presented. Firstly, several methods of producing real-valued AHC learning systems were examined. It was found that various forms of actor could provide a similar level of performance, though they tended to be very sensitive to the parameters used to train the system.

A method of combining Q-learning and AHC learning methods called Q-AHC was then introduced. This has the advantage over Q-learning of being able to produce continuous real-valued actions, rather than relying on a fixed action set. In addition, it has the advantage over real-valued AHC learning of providing a method that can cope with discontinuities in the policy more easily than a single generalising function approximator.

However, the performance of the Q-AHC system is disappointing when compared to that of pure Q-learning on the Robot Problem presented in chapter 4, though it is better than can be achieved using pure real-valued AHC learning. This is because the system

[4] This is similar to the idea used in Anderson (1993) to learn a Q-function using a resource-allocation network (RAN) (Platt 1991).

tends to use either Q-learning or AHC learning to construct its final policy, rather than a mixture of both. It was explained that this was due to each AHC element being attracted towards an arbitrary local maximum of the Q-function, with the possibility of the global maximum being missed.

Chapter 6

Conclusions

In this thesis, the aim has been to present reinforcement learning methods that are useful for the design of systems that can solve tasks in increasingly large and complex environments. The discrete Markovian framework, within which much of the work and theory of reinforcement learning methods has been developed, is not suitable for modelling tasks in large continuous state-spaces. Hence, the problems associated with applying reinforcement learning methods in high dimensional continuous state-space environments have been investigated, with a view to providing techniques that can be applied on-line utilising parallel computation for fast continuous operation.

The following areas were identified as being important features of more complex environments. The first was that learning an accurate model of such an environment could be extremely difficult, and so only methods that did not require a model to be learnt were considered. The second was that large and continuous state-spaces need methods which make maximum use of the information gathered in order to enable them to learn a policy within a reasonable time. To this end, updating methods that provide faster convergence were examined, as were generalising function approximators. Finally, methods were investigated to further enhance the reinforcement learning system by allowing it to produce real-valued vector actions.

The resulting learning methods provide many of the features required for scaling up reinforcement learning to work in high dimensional continuous state-spaces. The work presented is therefore intended to be a useful step in the direction of producing complex autonomous systems which can learn policies and adapt to their environments.

6.1 Contributions

This section summarises the main contributions made by the work presented in this thesis.

6.1.1 Alternative Q-Learning Update Rules

Several different Q-learning update rules have been considered, including new forms (Modified Q-Learning and Summation Q-Learning) for combining the method of TD(λ) with Q-learning. It has been empirically demonstrated that many of these update rules can outperform standard Q-learning, in both convergence rate and robustness to the choice of training parameters. Of these methods, Modified Q-Learning stands out as being the computationally simplest rule to implement and yet providing performance at least as good as the other methods tested, including Q(λ). Therefore, although it could be argued that other Q-learning update rules can perform as well as Modified Q-Learning, none of them appear to offer any advantages.

6.1.2 On-Line Updating for Neural Networks

Consideration has been given to the problems of applying reinforcement learning algorithms to more complex tasks than can be represented using discrete finite-state Markovian models. In particular, the problem of reinforcement learning systems operating in high dimensional continuous state-spaces has been investigated. The solution considered was to use multi-layer perceptron neural networks to approximate the functions being learnt.

Methods for on-line reinforcement learning using MLPs with individual weight eligibilities have been examined. It has been shown that these methods can be extended for use with multi-output Q-learning systems without requiring more than one eligibility trace per weight. The performance of these algorithms has been demonstrated on a mobile robot navigation task, where it has been found that on-line learning is in fact a more effective method of performing updates than backward-replay methods (Lin 1992), both in terms of storage requirements and sensitivity to training parameters. On-line learning also has the advantage that it could be used for continuously operating systems where no end-of-trial conditions occur.

6.1.3 Robot Navigation using Reinforcement Learning

The connectionist algorithms have been demonstrated on a challenging robot navigation task, in a continuous state-space, where finite state Markovian assumptions are not applicable. In this kind of problem, the ability of the system to generalise its experiences is essential, and this can be achieved by using function approximation techniques like MLP neural networks. Furthermore, in the robot task, the input vector is large enough (seven separate input variables in the task studied) that approximators that do not scale well to the number of inputs, such as lookup tables and CMACs, are inappropriate.

In the task considered, the robot was successfully trained to reach a goal whilst avoiding obstacles, despite receiving only very sparse reinforcement signals. In addition, the advantage over path-planning techniques of using a reactive robot was demonstrated by training the robot on a changing obstacle layout. This led to a control policy that could cope with a wide variety of situations and included the case where the goal was allowed to move during the trial.

6.1.4 Q-AHC Architecture

Finally, an investigation of systems that are capable of producing real-valued vector actions was made. To this end, a method of combining Q-learning methods with Adaptive Heuristic Critic methods, called Q-AHC, was introduced. However, the results with this architecture were not as encouraging as was hoped. Although the Q-AHC system outperformed the AHC system, it did not perform as well, in general, as the Q-learning methods. An analysis of why the system did not perform as well as might be expected was carried out, which suggested that the problem stemmed from multiple local maxima in the policy space. These caused difficulties for the gradient ascent methods used to adjust the action functions and could result in the system learning to produce sub-optimal policies.

6.2 Future Work

The reinforcement learning systems presented in this thesis attempt to provide many of the features required for coping with large continuous state-space tasks. However, there are still many areas that need further research, some of which are described below.

6.2.1 Update Rules

A whole variety of update rules for both Q-learning and AHC learning have been examined in this thesis. One thing that is very clear is that the established rules are not necessarily the best in performance terms, even though they are currently based on the strongest theoretical grounding. In this thesis, the update methods have been inspired primarily by the TD(λ) algorithm, rather than dynamic programming, with a view to providing methods that can be applied in non-Markovian domains.

The theory underlying the update rules presented in chapter 2 needs further investigation in order to explain under what conditions different methods can be expected to perform best. As mentioned in chapter 2, to guarantee convergence for a method such as Modified Q-Learning necessitates providing bounds on the exploration policy used during training. Also, further examination to find the features important for updating in continuous state-spaces with generalising function approximators is required. Williams and Baird (1993b) provide performance bounds for imperfectly learnt value functions, although these results are not directly applicable to generalising function approximators. The only proof of convergence in a continuous state-space belongs to Bradtke (1993), who examined a policy iteration method for learning the parameters for a linear quadratic regulator.

6.2.2 Neural Network Architectures

Throughout most of this thesis, the use of MLP neural networks as a function approximation technique has been used. The attraction of MLPs is that they are a 'black-box' technique that can be trained to produce any function mapping and so provide a useful general building block when designing complex systems. In addition, they are fundamentally a parallel processing technique and can be scaled for more complex mappings simply by adding more units and layers. Their disadvantage is that the conditions under which they will converge to producing the required function mapping is still not well understood.

Ideally, one would want to use an arbitrarily large neural network and expect it to work just as well as a small network. Work by Neal (1995) and others on Bayesian techniques may hold the answer, as methods have been provided in which the best weight values across the entire network are found by the learning procedure. Unfortunately, as with many of the more advanced learning methods, these techniques require complex calculations based on a fixed training set of data, which are not suitable for on-line parallel updating of networks for reinforcement learning tasks.

6.2.3 Exploration Methods

The exploration method used by the system is fundamental in determining the rate at which the system will gather information and thus improve its action policy. Various methods have been suggested (Thrun 1992, Kaelbling 1990) which are more sophisticated than functions based only on the current prediction levels like the Boltzmann distribution. However, most of these methods rely on explicitly storing information at each state, which can then be used to direct exploration when the state is revisited. Storing such data is not difficult for discrete state-spaces, but is not so easy for continuous ones. This is because continuous function approximators will generalise the data to other states, thus losing much of the important information. It is therefore necessary to consider exploration strategies that take this effect into account.

6.2.4 Continuous Vector Actions

The Q-AHC architecture was not as successful at providing real-valued vector actions as had been hoped. It may be that methods based on forward modelling (section 5.1.2) provide the key, as the resulting Q-function has the potential to provide action value estimates for every point in the action space. The main advantage this gives is the ability to provide initial action value estimates for judging new actions and skills. The lack of initial estimates was one of the difficulties discussed with the proposed idea of action function restarting (section 5.5.1).

The main disadvantage of the forward modelling approach is that to evaluate multiple action vectors at each time step, the same Q-function must be accessed multiple times. This either means losing the parallel processing property of the system, or maintaining multiple copies of the Q-function. However, this still remains a very interesting area for reinforcement learning research.

A Final Story

Even the best trained robots in the Robot Problem sometimes get stuck in a loop because they cannot find a way around the obstacles in their path. An examination of the predicted return during such a trial showed that the maximum action value was quite low when the robot was forced to turn away from the goal. Consequently, it was speculated that the robot might learn to use an 'emergency' action to get out of such situations. To this end, the robot was supplied with an extra action choice — to jump to a random location in the room. The robot was retrained with this new action available and the resulting policy examined. However, rather than simply travelling towards the goal and jumping if it was forced to turn back at any point, the robot had learnt a completely different and much more efficient solution. At the start of the trial, the robot chose to jump to random locations until it happened to arrive near the goal. At this point, it would return to using the conventional moves to cover the remaining distance. On average, the number of random jumps required to get near to the goal was significantly less than the number of standard moves required to reach the same point — the reinforcement learning system had arrived at a better solution than the designer had had in mind.

Appendix A

Experimental Details

The experiments described in the preceding chapters involved a number of parameters with settings that are presented here.

A.1 The Race Track Problem

In section 2.3 the Race Track problem was presented. The values of the parameters used for exploration and the learning rate were exactly as used in the original technical report (Barto et al. 1993) and are reproduced below. The Q-function values were initialised to zero for all state-action pairs in the lookup table.

The value of T in the Boltzmann exploration equation 4.1 was changed according to the following equation,

$$T_0 = T_{\max} \tag{A.1}$$

$$T_{k+1} = T_{\min} + \beta(T_k - T_{\min}) \tag{A.2}$$

where k is the *step* number (cumulative over trials), $\beta = 0.992, T_{\max} = 75$, and $T_{\min} = 0.5$. The fact that the step number, k, is used means that exploration is reduced towards the minimum value of 0.5 extremely quickly. For example, $t_{1000} = 0.52$. Yet the length of the first trial was over 1,200 steps on the small track, and of the order of 50,000 steps for the large track when non-RTDP methods were used. **Thus, the training algorithms learnt with effectively a fixed exploration constant of 0.5.**

The learning rate was set for each $Q(\mathbf{x}_t, a_t)$ visited according to,

$$\alpha(\mathbf{x}_t, a_t) = \frac{\alpha_0 \tau}{\tau + n(\mathbf{x}_t, a_t)} \tag{A.3}$$

where $\alpha_0 = 0.5$, $\tau = 300$, and $n(\mathbf{x}, a)$ was a count of the number of times state-action pair $Q(\mathbf{x}, a)$ had been visited in the course of the trials.

The value of γ was 1 in all trials.

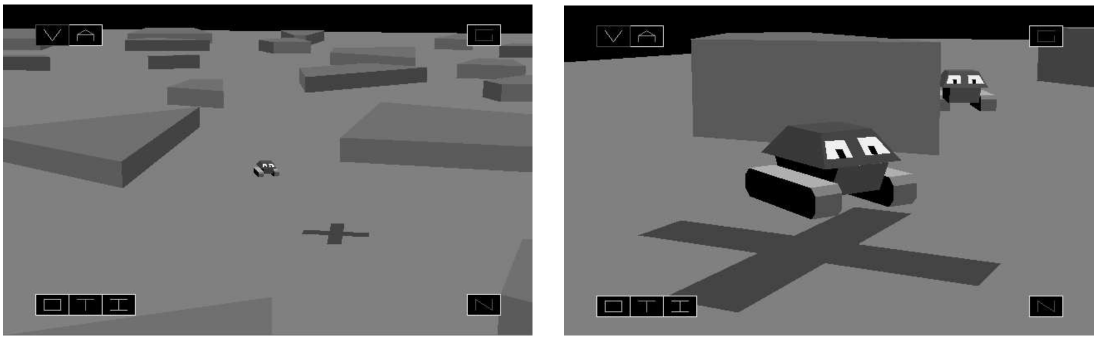


Figure A.1: Two snapshots taken from the real-time robot simulator. *Left:* The robot navigates itself through a large room full of obstacles. *Right:* He's behind you! Snapshot from a robot chase.

A.2 The Robot Problem

The Robot Problem was introduced in chapter 4 and used to test the MLP algorithms presented in this thesis. Here we present the implementation details required to reproduce this experimental environment.

A.2.1 Room Generation

The environment used in the trials consisted of a room of dimension $d_{\text{size}} \times d_{\text{size}}$ units containing 4 randomly placed convex obstacles. The goal position was generated at a random coordinate within the room. The obstacles were then generated by firstly placing a centre point (x, y) with both x and y in the range $[d_{\text{size}} - 2(r_{\text{max}} + d_{\text{gap}})] + d_{\text{gap}}$. The maximum radius for the obstacle was then calculated using the minimum of r_{max} or $d_n - r_n - d_{\text{gap}}$ where d_n and r_n were the centre to centre distance to obstacle n and its radius respectively. The actual radius was selected randomly between this maximum value and the minimum allowable obstacle size r_{min} (if the maximum radius was smaller than the minimum allowable, then a new centre point was generated and the process repeated).

Having defined the bounding circle for the obstacle, the coordinates of the vertices were then generated by firstly selecting a random angle θ_0 in the range $[0, \pi]$, and then selecting further angles θ_n at steps of $[0, 2\pi/3]$ until either 4 vertices were allocated, or the $\theta_n > \theta_0 + 2\pi$. The coordinates of the vertices were the positions on the circumference of the bounding circle at each of the selected angles.

The starting position for the robot was then generated by selecting points until one was found that was more than d_{gap} from each obstacle bounding circle and the boundary of the room.

The values used were $d_{\text{size}} = 40, d_{\text{gap}} = 2, r_{\text{max}} = 5, r_{\text{min}} = 2.5$. For the 'circle world' introduced in section 4.4.3, $d_{\text{size}} = 100, r_{\text{max}} = 10$, and the number of obstacles (which were generated as for the bounding circles described above) was increased to 29.

A.2.2 Robot Sensors

The sensor readings available to the robot were described in section 4.2. These values were coarse coded before being input to the neural networks using a scheme illustrated by

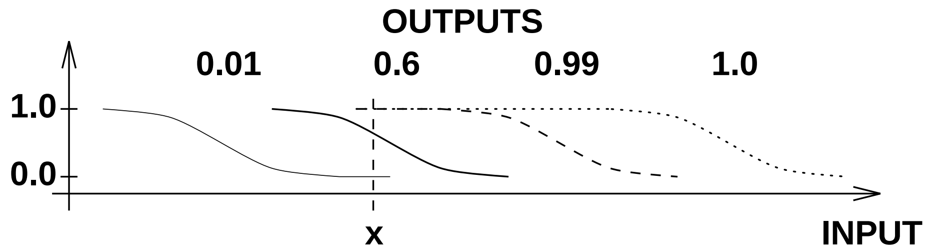


Figure A.2: Coarse coding of a real-valued input to provide a suitable input for a neural network. The diagram shows how an input value (represented by the vertical dashed line) is coded using values between [0,1] by reading off the values from 4 sigmoid functions spread across the input space.

Fig. A.2. Each real-valued input was spread across N network inputs, i_n, by feeding them through a sigmoid function,

$$i_n = \frac{1}{1+\exp^{w_n(b_n - x)}} \tag{A.4}$$

with single weight w_n and bias value b_n which were fixed (note that the input is *subtracted* from the bias *before* being weighted by w_n). x is a real-valued input, which is therefore shifted and scaled by b_n and w_n, and produces an input to the network in the range [0,1]. So, **i** is a N-tuple of values between [0,1], with the number that are 'on' (close to 1.0) rising as the size of x decreases.

The number of sigmoid functions and their weight and bias values are given here by the following formulae,

$$w_n = \frac{4N}{r} \tag{A.5}$$

$$b_n = \left(\frac{2n-1}{2}\right)\frac{r}{N} \tag{A.6}$$

where N is the number of network inputs, and r is the range $[0, r]$ of values of x over which the inputs will be most sensitive. Thus, the values given below are for N and r.

The five range sensor inputs used 3 network inputs each ($N = 3$) and had a range $r = 8$ (c.f. $d_{\text{size}} = 40$). The goal distance used $N = 5$ and $r = 25$. The relative angle to the goal, ψ (range $[0, 2\pi]$), was coded in two halves (to represent the concepts of goal-to-left and goal-to-right); $\pi - \psi$ was fed into 3 inputs ($N = 3, r = \pi$) and $\psi - \pi$ into another 3 ($N = 3, r = \pi$).

The overall thinking behind this form of coding was that more inputs should come 'on' as the value x became more important. Thus short ranges to obstacles result in the related network inputs switching on, as does a low range to the goal, or large relative goal angles (if the robot is facing towards the goal, all the angle network inputs will be zero).

The robot was considered at the goal if it was within a radius of 1 unit of the goal position and crashed if within 1 unit of an obstacle. At each time step, the maximum distance it could move forward was $d = 0.9$.

Appendix B

Calculating Eligibility Traces

For completeness, the calculation of the output gradients and hence the eligibility traces is given here for the case where the back-propagation algorithm is used.

A multi-layer perceptron is a collection of interconnected units arranged in layers, which here are labelled $i, j, k...$ from the output layer to the input layer. A weight on a connection from layer i to j is labelled w_{ij}. Each unit performs the following function,

$$o_i = f(\sigma_i) \tag{B.1}$$

$$\sigma_i = \sum_j w_{ij} o_j \tag{B.2}$$

where o_i is the output from layer i and $f(.)$ is a sigmoid function.

The network produces I outputs, of which only one, o_i^*, is selected. The output gradient is defined with respect to this output for the output layer weights as,

$$\frac{\partial o_i^*}{\partial w_{ij}} = f'(\sigma_i) o_j \tag{B.3}$$

where $f'(.)$ is the first differential of the sigmoid function. This will be zero for all but the weights w_{ij} attached to the output unit which produced the selected output, o_i^*.

For the first hidden layer weights, the gradient therefore is simply,

$$\frac{\partial o_i^*}{\partial w_{jk}} = f'(\sigma_i^*) w_{ij} f'(\sigma_j) o_k \tag{B.4}$$

These values are added to the current eligibilities. Generally, there would be one output gradient for each output $i \in I$ and hence I eligibilities would be required for each weight. This is so that when temporal difference error, E_i, of each output arrived, the weights could be updated according to,

$$\Delta w_{jk} = \eta_t \sum_i E_i e_{jk}^i \tag{B.5}$$

where e_{jk}^i is the eligibility on weight w_{jk} which corresponds to output i. However, in Q-learning, there is only a single temporal difference error which is calculated with respect to the output which produced the current prediction. Hence only one output gradient is calculated at each time step and only one eligibility is required per weight.

Bibliography

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity, *Proceedings of the Seventh AAAI Conference*, pp. 268–272.

Albus, J. S. (1981). *Brains, Behaviour and Robotics*, BYTE Books, McGraw-Hill, chapter 6, pp. 139–179.

Anderson, C. W. (1993). Q-learning with hidden-unit restarting, *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann.

Barraquand, J. and Latcombe, J. (1991). Robot motion planning: A distributed representation approach, *The International Journal of Robotics Research* **10**(6): 628–649.

Barto, A. G., Bradtke, S. J. and Singh, S. P. (1993). Learning to act using real-time dynamic programming, *Technical Report CMPSCI 93-02*, Department of Computer Science, University of Massachusetts, Amherst MA 01003.

Barto, A. G., Sutton, R. S. and Anderson, C. W. (1983). Neuron-like adaptive elements that can solve difficult learning control problems, *IEEE Transactions Systems, Man, and Cybernetics* **13**: 834–836.

Bellman, R. (1957). *Dynamic Programming*, Princeton University Press, Princeton, New Jersey.

Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*, Prentice Hall, Englewood Cliffs, NJ.

Bertsekas, D. P. and Tsiksiklis, J. N. (1989). *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ.

Boyan, J. A. (1992). *Modular neural networks for learning context-dependent game strategies*, Master's thesis, University of Cambridge, UK.

Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation, *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann, pp. 295–302.

Brody, C. (1992). Fast learning with predictive forward models, *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann, pp. 563–570.

Brooks, R. A. (1986). A robust layered control system for a mobile robot, *IEEE Journal of Robotics and Automation* **2**: 14–23.

Cichosz, P. (1994). *Reinforcement learning algorithms based on the methods of temporal differences*, Master's thesis, Warsaw University of Technology Institute of Computer Science.

Cichosz, P. (1995). Truncating temporal differences: On the efficient implementation of TD(λ) for reinforcement learning, *Journal of Artificial Intelligence Research* **2**: 287–318.

Cybenko, C. (1989). Approximation by superpositions of a sigmoidal function, *Mathematics of Control, Signals, and Systems* **2**: 303–314.

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

Funahashi, K. (1989). On the approximate realization of continuous mappings by neural networks, *Neural Networks* **2**: 183–192.

Gullapalli, V., Franklin, J. A. and Benbrahim, H. (1994). Acquiring robot skills via reinforcement learning, *IEEE Control Systems Magazine* **14**(1): 13–24.

Hassibi, B. and Stork, D. G. (1993). Optimal brain surgeon and general network pruning, *International Conference on Neural Networks*, Vol. 1, San Francisco, pp. 293–299.

Holland, J. H. (1986). Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems, *in* R. S. Michalski, J. G. Carbonell and T. M. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach*, Vol. 2, Morgan Kaufmann, Los Altos, CA.

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

Jaakkola, T., Jordan, M. I. and Singh, S. P. (1993). On the convergence of stochastic iterative dynamic programming algorithms, *Technical Report MIT Computational Cognitive Science 9307*, Massachusetts Institute of Technology.

Jacobs, R., Jordan, M. and Barto, A. (1991). Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks, *Technical Report COINS 90-27*, Department of Computer and Information Science, University of Massachusetts, Amherst.

Jervis, T. T. and Fitzgerald, W. J. (1993). Optimization schemes for neural networks, *Technical Report CUED/F-INFENG/TR 144*, Cambridge University Engineering Department, UK.

Jordan, M. I. and Jacobs, R. A. (1990). Learning to control an unstable system with forward modelling, *Advances in Neural Information Processing Systems 2*, Morgan Kaufmann.

Jordan, M. I. and Jacobs, R. A. (1992). Hierarchies of adaptive experts, *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann, San Mateo, CA, pp. 985–993.

Kaelbling, L. P. (1990). *Learning in Embedded Systems*, PhD thesis, Department of Computer Science, Stanford University.

Kant, K. and Zucker, S. W. (1986). Toward efficient trajectory planning: The path-velocity decomposition, *The International Journal of Robotics Research* **5**(3): 72–89.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots, *International Journal of Robotics Research* **5**(1): 90–98.

Lee, Y., Song, H. K. and Kim, M. W. (1991). An efficient hidden node reduction technique for multilayer perceptrons, *IJCNN'91*, Vol. 2, Singapore, pp. 1937–1942.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching, *Machine Learning* **8**: 293–321.

Lin, L. (1993a). Hierarchical learning of robot skills by reinforcement, *IEEE International Conference on Neural Networks*, Vol. 1, San Francisco, pp. 181–186.

Lin, L. (1993b). *Reinforcement Learning for Robots Using Neural Networks*, PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Lin, L. (1993c). Scaling up reinforcement learning for robot control, *Machine Learning: Proceedings of the Tenth International Conference*, Morgan Kaufmann.

Mackay, D. (1991). *Bayesian Methods for Adaptive Models*, PhD thesis, California Institute of Technology, Pasadena, California.

Mahadevan, S. (1994). To discount or not to discount in reinforcement learning: A case study comparing R learning and Q learning, *Machine Learning: Proceedings of the Eleventh International Conference*, Morgan Kaufmann.

Millan, J. R. and Torras, C. (1992). A reinforcement connectionist approach to robot path finding in non-maze-like environments, *Machine Learning* **8**: 363–395.

Moller, M. (1993). A scaled conjugate gradient algorithm for fast supervised learning, *Neural Networks* **6**: 525–533.

Narendra, K. and Thathachar, M. (1989). *Learning Automata: An Introduction*, Prentice-Hall, Englewood Cliffs NJ 07632, USA.

Neal, R. M. (1995). *Bayesian Learning For Neural Networks*, PhD thesis, Graduate School of Computer Science, University of Toronto.

Peng, J. and Williams, R. J. (1993). Efficient learning and planning within the Dyna framework, *ICNN*, Vol. 1, San Francisco, pp. 168–174.

Peng, J. and Williams, R. J. (1994). Incremental multi-step Q-learning, *in* W. Cohen and H. Hirsh (eds), *Machine Learning: Proceedings of the Eleventh International Conference (ML94)*, Morgan Kaufmann, New Brunswick, NJ, USA, pp. 226–232.

Platt, J. C. (1991). A resource-allocating network for function interpolation, *Neural Computation* **3**: 213–225.

Prescott, T. J. (1993). *Explorations in Reinforcement and Model-based Learning*, PhD thesis, Department of Psychology, University of Sheffield, UK.

Prescott, T. J. and Mayhew, J. E. W. (1992). Obstacle avoidance through reinforcement learning, *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann, San Mateo, CA, pp. 523–530.

Puterman, M. L. and Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems, *Management Science* **24**: 1127–1137.

Ram, A. and Santamaria, J. C. (1993). Multistrategy learning in reactive control systems for autonomous robotic navigation, *Informatica* **17**(4): 347–369.

Reed, R. (1993). Pruning algorithms — a survey, *IEEE Transactions on Neural Networks* **4**(5): 740–747.

Riedmiller, M. (1994). Advanced supervised learning in multi-layer perceptrons — from backpropagation to adaptive learning algorithms, *International Journal of Computer Standards and Interfaces* **16**(3): 265–278.

Ross, S. (1983). *Introduction to Stochastic Dynamic Programming*, Academic Press, New York.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). *Parallel Distributed Processing*, Vol. 1, MIT Press.

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems, *Technical Report CUED/F-INFENG/TR 166*, Cambridge University Engineering Department, Cambridge, England.

Sathiya Keerthi, S. and Ravindran, B. (1994). A tutorial survey of reinforcement learning, *Technical report*, Department of Computer Science and Automation, Indian Institute of Science, Bangalore.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments, *Proceedings of the Tenth IJCAI*, pp. 1039–1046.

Schwartz, A. (1993). A reinforcement learning method for maximising undiscounted rewards, *Machine Learning: Proceeding of the Tenth International Conference*, Morgan Kaufmann.

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks, *Machine Learning* **8**(3/4): 323–339.

Singh, S. P. and Sutton, R. S. (1994). Reinforcement learning with replacing eligibility traces, In preparation.

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 differences, *Machine Learning* **3**: 9–44.

Sutton, R. S. (1989). Implementation details of the TD(λ) procedure for the case of vector predictions and backpropagation, *Technical Report TN87-509.1*, GTE Laboratories.

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*, Morgan Kaufmann, Austin, Texas, pp. 216–224.

Sutton, R. S. and Singh, S. P. (1994). On step-size and bias in temporal-difference learning, *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, Centre for Systems Science, Yale University, pp. 91–96.

Tesauro, G. J. (1992). Practical issues in temporal difference learning, *Machine Learning* **8**: 257–277.

Tham, C. K. (1994). *Modular On-Line Function Approximation for Scaling Up Reinforcement Learning*, PhD thesis, Jesus College, Cambridge University, UK.

Tham, C. K. and Prager, R. W. (1992). Reinforcement learning for multi-linked manipulator control, *Technical Report CUED/F-INFENG/TR 104*, Cambridge University Engineering Department, UK.

Thrun, S. (1994). An approach to learning robot navigation, *Proceedings IEEE Conference of Intelligent Robots and Systems*, Munich, Germany.

Thrun, S. and Schwartz, A. (1993). Issues in using function approximation for reinforcement learning, *Proceedings of the Fourth Connectionist Models Summer School*, Lawrence Erblaum, Hillsdale, NJ.

Thrun, S. B. (1992). Efficient exploration in reinforcement learning, *Technical Report CMU-CS-92-102*, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213-3890.

Thrun, S. B. and Möller, K. (1992). Active exploration in dynamic environments, *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann, pp. 531–538.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning, *Machine Learning* **16**(3): 185–202.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*, PhD thesis, King's College, Cambridge University, UK.

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

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it, *Proceedings of the IEEE*, Vol. 78, pp. 1550–1560.

Williams, R. J. (1988). Toward a theory of reinforcement learning connectionist systems, *Technical Report NU-CCS-88-3*, College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115.

Williams, R. J. and Baird, L. C. (1993a). Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems, *Technical Report NU-CCS-93-11*, Northeastern University, College of Computer Science, Boston, MA 02115.

Williams, R. J. and Baird, L. C. (1993b). Tight performance bounds on greedy policies based on imperfect value functions, *Technical Report NU-CCS-93-13*, Northeastern University, College of Computer Science, Boston, MA 02115.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks, *Neural Computation* **1**: 270–280.

Wilson, S. W. (1994). ZCS: A zeroth level classifier system, *Evolutionary Computation* **2**(1): 1–30.

Zhu, Q. (1991). Hidden Markov model for dynamic obstacle avoidance of mobile robot navigation, *IEEE Transactions on Robotics and Automation* **7**(3): 390–397.