CHAPTER 2

Reinforcement Learning

In the following chapter, sections 2.1 through to 2.8 are essentially a review of the book: "Reinforcement Learning" by Sutton and Barto (1998). The book provides a lucid, comprehensive and consistent account of the theory and its history, and in the author's opinion represents the best introduction to the subject. This chapter also draws significantly on an authoritative reinforcement learning survey by Kaelbling et al. (1996).

The main topics of the thesis are now reviewed in this and the two subsequent chapters. This chapter is devoted to the history, theory and practical application of reinforcement learning, while the main purpose of chapter 3 is to introduce neural networks as an implementational paradigm. Both these sections provide the foundation for chapter 4 which concludes the introductory material by reviewing existing work on the reinforcement learning of real-valued functions, with an emphasis on different approaches to representing the action space.

2.1 History

Historically there are two main strands that contribute to the field of reinforcement learning: animal psychology and Dynamic Programming.

Animal learning is traced to Thorndike (1911), who suggested that an animal, given a choice of responses in a given situation, would when encountering that same situation again, be more likely to reproduce an action that resulted in satisfaction, and less likely to reproduce one that resulted in dissatisfaction. This intuitive idea was also developed by Pavlov (1927), and is commonplace in modern psychology.

In the late 1950s the phrase *optimal control* was used to describe a technique for minimising a measure of a dynamic system's performance. Bellman developed a functional equation — now called the *Bellman equation* — for calculating the value function of a dynamic system. The process of solving a set of these equations, either analytically or incrementally in order to first estimate the values of the various states of the system, and then derive a policy for maximising the expected return over the life of the system developed into the field of Dynamic Programming (Bellman, 1957), and today represents the theoretical grounding of all RL techniques.

It was not until the early 1980s that Barto, Sutton, Watkins and others began defining modern reinforcement learning, uniting the strands, clarifying the theory and, importantly, distinguishing the field from supervised learning, thereby giving RL its own identity and its own place in the machine learning literature.

2.2 Introduction

The intuition behind reinforcement learning (RL) is very simple — an agent learns for itself how to maximise a reinforcement signal from its environment by trial and error exploration of different actions in different situations. If the signal is designed to yield high reward at goal states, and low reward in situations that are to be avoided, then in learning to maximise that signal, the agent will hopefully also learn how to achieve its goals. Unlike supervised learning, where the desired output is presented along with the input, in RL only the *value* of an action is provided, and the agent itself is responsible for discovering and selecting appropriate actions based on the relative strengths of these values.

The standard example of a simple RL problem is the *n-armed bandit* where one of n levers must be pulled at each time-step, with each lever yielding a reward according to a fixed distribution. Imagine being in a situation with two levers and one hundred pulls to make. How would you maximise your reward? Clearly one strategy involves trial and error sampling until sufficient confidence is held in the belief that one arm yields a higher expected reward than the other, at which point only that arm should be pulled. Each arm could be tried just once, and then the one yielding greatest reward pulled thereafter, but this does not allow for an unlucky sample. If the reward distributions of the two arms are similar and you still have many goes left, it makes sense to take a larger number of samples (of both levers) to be sure you get the most out of the remainder of the game. Conversely, on the 100th go, the only sensible thing to do is pull the lever with the highest expected reward according to your experience so far.

Pulling the arm which is believed to yield the best result is known as *exploiting* the current knowledge. But in order to be confident about that knowledge, all options must first be *explored* (even if they initially appear likely to be worse) in case the new information uncovers greater reward in the long run. This is known as the *explore/exploit dilemma* since on the one hand exploration is necessary to uncover reliable information, but on the other hand exploitation is necessary to make the most of that information, and they cannot both be performed at the same time. It obviously makes sense to explore more at the beginning of a trial when the information gained will be of most use, and to exploit at the end when the cost of exploring will tend to outweigh the benefits of the new information gained. In most practical applications, optimal solutions to this dilemma are not known, but some commonly used strategies will be introduced shortly. For a discussion of bandit problems, see Narendra and Thathachar (1989).

The n-armed bandit problem is actually a special case of the more general RL problem, as there is only one state that the system can be in — namely that of being faced with pulling one of the arms. From this single state there are a number of actions with each action corresponding to pulling one of the arms. In the more general problem, the system can be in any one of a number of states with the optimal action depending on the state. An example is the game of noughts-and-crosses in which each board position can be thought of as a state of the problem, and in which there are nine actions — one for each square of the grid. Not all actions will be available in each state, and of course different actions will be preferred in different states. A system for selecting an action to take in each state is referred to as the *policy*. For example, one (unrecommended) policy for noughts-and-crosses would be to always take the top-left most square available.

2.3 Markov Decision Processes

It is convenient to consider the environment as a *Markov Decision Process* (MDP), an example of which is shown in figure 2.1. The system has four states (one of which happens to be a terminal state making it a *finite horizon MDP*), and two actions. The states are numbered 1 to 4, and the actions labelled a1 and a2. In each state, the two actions will take the system into a new state with a fixed probability which is indicated to the right of the colon for each transition. In the context of the RL problem, each *state transition* also yields a reward as a scalar value. It is easy to imagine the noughts-and-crosses example drawn and labelled like figure 2.1. In this case there would be $3^9 = 19683$ states[1], nine actions, $\frac{9!}{5!4!} \times 2 = 252$ terminal states, and the transition probability from each state under each action would be unity since the game is deterministic.

In the rest of this section, it is assumed that the environment is represented by an MDP. *This will imply that each state contains sufficient information so that the probability of moving to any next state, s', and receiving any reward, r, is the same given the current state and action information as if given the entire state-action-reward history of the environment.* This can be expressed by the following equality, which defines the *Markov property*:

[1] Not all of these would be valid board positions.

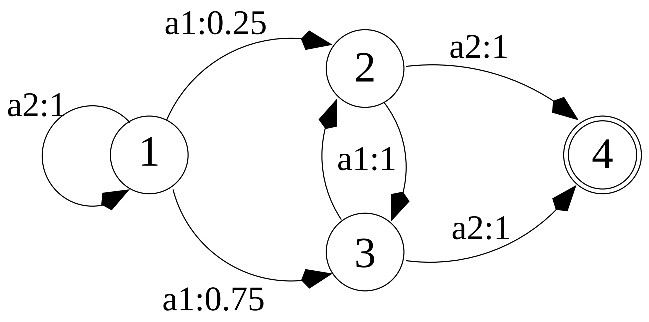

Figure 2.1: A simple Markov Decision Process consisting of four states and two actions.

$$P(s_{t+1} = s', r_{t+1} = r | s_t, a_t) = P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \dots, s0, a0) \tag{2.1}$$

for all s_{t+1}, r_{t+1}, and state-action-reward histories, where s_t is the state at time t, a_t is the action taken at time t, and r_t is the reward received in moving to state s_t.

This assumption ensures that at each state, the agent has sufficient information to make a perfectly informed decision given the boundaries of the particular problem. In the noughts-and-crosses example, coding the board positions as states results in a game with the Markov property because a complete board position contains all the salient information for winning a game. But coding a maze with states corresponding to "left corner", "right corner", "corridor" does not yield an MDP because escaping a maze requires at least an implicit knowledge of location which cannot usually be inferred from the immediate surroundings. However, if a state history is maintained so that the escapee can remember the types of previous junctions, then it may be possible to localise, but this then corresponds to a different MDP in which the states are n-tuples of the old "left/right/corridor" states. Of course if the states are coded as *particular* corners and corridors, then it *is* an MDP because there is as much information in a single state as in an entire state history (with respect to escaping the maze). The maze is an example of a *deterministic* process since one assumes transitions between states will occur with probabilities zero or one (although of course one could easily contrive an example where they do not). An example of a more general, non-deterministic MDP is encountered shortly in figure 2.2.

Note that even though an agent may have access to sufficient information so that the Markov property *is* satisfied, an inability to perceive that information (perhaps through impoverished sensing apparatus) may lead to the agent effectively facing a non-Markovian decision process. This is referred to as *perceptual aliasing*.

2.4 The basics

The standard reinforcement problem is defined using the following elements:

- **Set of states**
  A set of discrete and distinct states, **S**, corresponding to the learning agent's perception of the states of its environment. A state could be a board game position, a vector of robot sensor readings, a position within a maze etc.

- **Set of actions**
  A set of discrete actions, **A**, available to the agent. Not every action need be available in every state.

- **Policy**
  The policy, π, dictates which actions are to be taken in each state. Policies may be stochastic.

- **Reward function**
  The real-valued reward function, **R**, maps states, state-action pairs or state-action-state tuples to reward values. Reward values may be positive, negative, or zero indicating no reward. The reward function is usually unknown to the agent, and must first be explored and then exploited.

- **Value function**
  The value function, **V**, is a central idea to RL techniques and maps each state to a measure of the value of that state. The value of a state is taken to reflect the expected accumulated reward from that state on. V is usually taken to refer to the *actual* value function, while $\hat{V}$ refers to the *estimated* value function. V^{π}

refers to the value function under some policy, π, while V^* refers to the optimal value function — i.e. the value function under the optimal policy.

- **Model of environment**
  The environment model, which may or may not be known to the agent, predicts the behaviour of the environment by mapping state-action-nextstate tuples to probabilities. The environment model is provided by $T(s,a,s')$ which returns the probability of moving to state s' after taking action a in state s, for all s,a,s'.

  The environment model is provided in the form of a transition function, T, from state-action-nextstate tuples to probabilities.

This outlines the basic RL context that was introduced in figure1.1. The additional constraint has been added that the environment model behaves as an MDP.

At each time-step an agent moves from one state to another by taking one of its available actions, and in so doing receives a scalar reward. The question is, against what measure should the agent's behaviour be optimised? One answer is to attempt to maximise the sum of all expected future reward, up to a *receding finite horizon*:

$$E\left(\sum_{t=0}^{h} r_t\right) \tag{2.2}$$

where h is the horizon and r_t is the reward received from the environment at time t after an action is taken.[2] [3] This *return* has to be 'expected' because of the stochastic nature of the environment. It is not assumed for example that an action guarantees a particular state transition, only a probability of that transition. If a task is of finite length, as with the two-armed bandit example earlier, then this approach may be adequate, but since

[2]An alternative is the *fixed* finite horizon in which the reward is summed all the way up to the fixed end of the trial.

[3]Strict statistical notational convention dictates that uppercase R is used to denote the random variable representing reward inside an expectation. However, for consistency, the notation adopted here and throughout is that of Kaelbling et al. (1996) and Sutton and Barto (1998).

we may not wish to make such an assumption, a more common value to attempt to maximise is the *discounted return*:

$$E\left(\sum_{t=0}^{\infty}\gamma^t r_t\right) \tag{2.3}$$

where $0 \leq \gamma < 1$ is called the *discount factor*. The idea behind (2.3) is that rewards are exponentially decayed as they become more and more distant and this ensures a finite sum, even on an indefinitely long training episode. There is also an intuitive appeal in trying to maximise immediate reward more than distant reward. In this way, γ effectively sets the horizon.

The value function, V^{π}, is defined as (2.2) or (2.3) for each state, based on the information provided by the reward function following that state given a policy, π. The aim of reinforcement learning is to discover an optimal policy, π^*, which maximises (2.2) or (2.3). If the environment model is known explicitly in terms of the transition probabilities and the reward function, then it may be feasible to analytically solve for the value function under the optimal policy, to give first V^* and then π^*. However, in many cases the environment model is not known, and a solution must be approximated by an iterative sampling method. This thesis is concerned exclusively with problems where the environment is not known.

2.5 Dynamic Programming

Assuming that (2.3) is the value we wish to maximise, and therefore first estimate, the value function, V, can be expressed by the Bellman equation (Bellman, 1957):

$$V^{\pi}(s) = \sum_{a\in A}\pi(s,a)\sum_{s'\in S}T(s,a,s')\left[R(s,a,s') + \gamma V^{\pi}(s')\right] \tag{2.4}$$

where $\pi(s,a)$ is the probability of taking action a in state s under policy π, $T(s,a,s')$ is the probability of s' being the successor state of s following action a (i.e. the state-transition function), and $R(s,a,s')$ is the reward elicited from the environment by taking action a in state s and ending up in state s'.[4]

The Bellman equation, which can be viewed as a recursive definition of equation 2.3, asserts that the value of state s under policy π is the result of summing, for each action and each possible successor state, the expected reward of that transition plus the discounted value of that successor state.

For a suitable RL problem, this yields a set of simultaneous equations, one for each state, which can be solved to yield the value function V^{π}. The following example is taken straight from Sutton and Barto (1998) (pg 71): Consider the grid world of figure 2.2a in which each square is a state from which the agent may choose one of the following actions: "up", "down", "left" or "right". Each of these actions takes the agent to the appropriate neighbouring state and yields no reward except that attempting to move off the grid results in no movement and a reward of -1, and any action taken in states A or B results in a move to A' or B' with rewards of +10 and +5 respectively. Figure 2.2b shows the value of each state, as calculated by equation 2.4, for the policy in which each action is equally likely in each state, and with the discount factor, $\gamma = 0.9$. The negative values of edge squares in the lower half of the grid reflect the probability of the agent stumbling off the grid at these points. States A and B have high values, as do their neighbours, because of the potential for achieving the +5 or +10 rewards. However, the value of state A is slightly diminished by both its proximity to the lower edge of the grid via the special state transition $A \rightarrow A'$, and also the distance of the inevitable successor state, A', from the rewarded states A and B.

Each value is the expected discounted reward from that state onwards for the equiprobable policy. However, what we are more interested in is the *optimal policy*, π^*, which guarantees the greatest possible future reward. Equation 2.5 shows the *Bellman optimality equation* for V^*, which yields the expected return of each state if the best possible action is always taken. In the same way as before, a set of simultaneous equations

[4]We use R for the reward function (from state-action tuples), and r_t to denote the reward at a particular time, t.

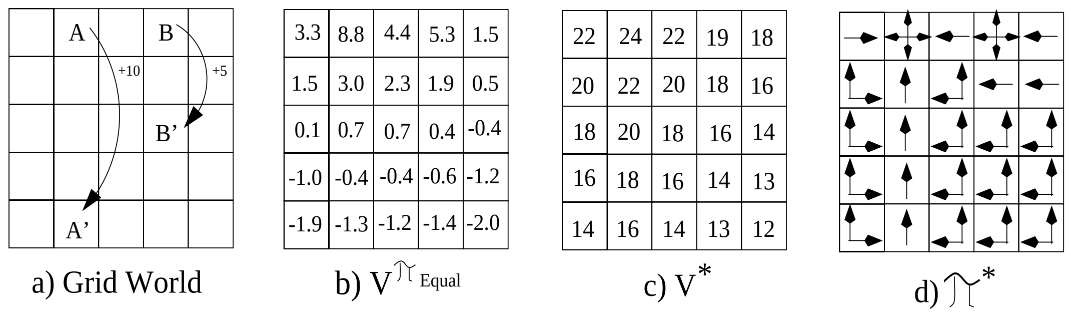

Figure 2.2: Application of the Bellman equations. Reproduced with permission from (Sutton and Barto, 1998)(pg. 170).

— one for each state — can be solved to yield V^*.

$$V^*(s) = \max_a \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right] \tag{2.5}$$

This equation is very similar to the previous Bellman equation except that instead of considering all possible actions from state s, only the action that maximises the future return is used. Figure 2.2c shows the V^* values for each state calculated using (2.5), and figure 2.2d shows the optimal policy, π^*, which can be generated by always selecting an action that maximises the right hand side of (2.5) for the current state. The optimal policy happens to prescribe moves that takes the system into state A as quickly as possible (unless avoiding state B in the process would require a detour).

However, solving n simultaneous equations in n unknowns where n is the number of states scales with $O(n^3)$ and soon becomes too expensive. *Dynamic Programming* alleviates this problem by changing the Bellman equation into an update rule that can be applied iteratively, one state at a time:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right] \tag{2.6}$$

where V_k is the value function at the k^{th} iteration. Note that V_0 should be initialised to finite values.

For some policy, π, V_k is updated at every state using the previous values of V_{k-1}. The reason successive approximations improve the accuracy is that fresh information is being injected by the term $R(s,a,s')$ during each iteration. It can be shown that V_k converges to V^π as $k \to \infty$ (Bellman, 1957), and calculating V^π by iteratively updating the value function in this way is called *iterative policy evaluation*.

Now, based on V^π, it is possible to improve the policy to take advantage of the approximated value function. For example, in figure 2.2(b), knowing $V^{\pi_{Equal}}$ suggests a number of improvements to π in which the higher value states are preferentially sought. In practice, this can be achieved by setting $\pi(s,a) = 1$ for the $a \in A$ that maximises $\sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]$, and setting $\pi(s,a) = 0$ elsewhere. This is called *policy improvement* and is guaranteed to yield a better policy, π', if one exists.

But this now means that the value function is based on an out of date policy and needs to be recomputed to reflect the new improved policy, π'. In this way, by repeatedly performing iterative policy evaluation followed by policy improvement, better and better policies are found converging on the optimum policy, π^*, and a corresponding optimum value function V^*. This is called *policy iteration*, and forms the theoretical basis of all practical RL techniques.

Equation (2.6) was an iterative version of the Bellman equation of (2.4). Similarly, the Bellman *optimality equation* of (2.5) can also be expressed as an iterative update rule:

$$V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right] \tag{2.7}$$

which also directly converges to V^* without the need to maintain an explicit policy. This corresponds to the previous update rule in which the policy is effectively updated immediately rather than waiting for policy evaluation to converge. It is also possible to update $V_{k+1}(s)$ on states in any order, and providing each state is continually visited and never left unchanged indefinitely, convergence is still guaranteed as $k \to \infty$.

The principle behind policy iteration is that information about all rewards is passed around the system so that the value of each state eventually accurately reflects its intrinsic worth to the agent with respect to the expected return.

2.6 Monte Carlo techniques

Dynamic Programming requires that a complete model of the environment be known in terms of the state transition probabilities, $T(s,a,s')$, and the reward function, $R(s,a,s')$. In practice this information is unlikely to be available, and so the *Monte Carlo* method is introduced.

Trials are now required to be finite, so a guarantee is required that a terminal state of the MDP will be reached sooner or later. As with Dynamic Programming, the Monte Carlo approach aims to generate increasingly accurate estimates of the value function. Unlike Dynamic Programming however, where $V(s)$ is recursively updated using the value function at other states, Monte Carlo techniques update $V(s)$ towards the actual reward received from state s until the end of the trial. So the value function for a given policy, π, can be written as:

$$V^{\pi}(s_T) = E\left\{ \sum_{t=T}^{\text{trial end}} r_t | \pi, s_T \right\} \tag{2.8}$$

where T is the time at which state s is first encountered. Note that because the trial is restricted to being finite, (2.2) is now being used instead of (2.3) as the quantity to maximise. However, it is still common to discount:

$$V^{\pi}(s_T) = E\left\{ \sum_{t=T}^{\text{trial end}} \lambda^{t-T} r_t | \pi, s_T \right\} \tag{2.9}$$

$V^{\pi}(s)$ can be calculated by running N trials, and for the first visit to state s (at time T) during trial k, calculating the value:

$$R_k = \sum_{t=T}^{\text{trial end}} r_t \tag{2.10}$$

Then (2.8) is approximated by the *Monte-Carlo first visit estimate*:

$$V^{\pi}(s) \approx \frac{\sum_{n=1}^{N} R_n}{N} \tag{2.11}$$

with convergence as $N \longrightarrow \infty$.

But there are two important implications of not having an environment model. The first is that the value function is no longer sufficient for finding an optimal policy because even if the agent knows which the best states are, it does not know how to get there from the current state without the state transition function, $T(s,a,s')$. For this reason, instead of estimating the value of states using $V(s)$, estimates are made of the values of *state-action pairs* using the *action value function* (also *action* function), $Q(s,a)$.[5] The theory is the same as before with $\lim_{trials \to \infty} Q^{\pi}(s,a) =$E(Total reward from taking action a in state s to the end of that trial, under π) except that now a state transition function is implicitly built into the action value function. Making a policy optimal with respect to Q^{π} is now simply a matter of always choosing the action that maximises $Q^{\pi}(s,a)$ at each state s. This is known as a *greedy policy*. This then becomes π', which in turn is evaluated by $Q^{\pi'}$ and so policy iteration continues in the usual way.

However, the second important implication of not having an environment model is that the issue of exploration must now be addressed. The agent is now responsible for sampling its own environment, whereas before the environment details were provided explicitly. So now, rather than updating the policy so that the action that maximises

[5] This formulation pre-empts Q-learning, which is introduced shortly.

$Q^\pi(s,a)$ is *always* chosen, π' is instead formed by *usually* choosing the action that maximises $Q^\pi(s,a)$ while occasionally selecting one of the other actions. This balances exploration and exploitation. These are known as ε-*soft* policies because each available action has a non-zero probability of being taken. One common ε-soft policy is to select the currently preferred action with probability $1-\varepsilon+\frac{\varepsilon}{|A|}$ and all other actions with probability $\frac{\varepsilon}{|A|}$, for some small value of ε. This is known as an ε-*greedy* policy. A similar but smoother approach is to select actions according to a Boltzmann distribution of their corresponding action values:

$$P(a) = \frac{e^{Q(s,a)/T}}{\sum_{b\in A} e^{Q(s,b)/T}} \tag{2.12}$$

With ε-soft exploration, convergence to the optimal policy is once again assured providing the policy converges to pure greedy. This is easily achieved by reducing ε or the temperature parameter, T, to zero. This ensures a shift from exploration to exploitation.

The advantage of Monte Carlo techniques of not requiring an environment model will turn out to be decisive not only when the environment is unknown, but also when $T(s,a,s')$ or $R(s,a,s')$ are known implicitly but difficult to calculate explicitly. See Sutton and Barto (1998)(pg 113) for an example. A disadvantage of having to implicitly model the environment by making the domain of the action value function state-action pairs rather than just states, is that the action function must now be stored and updated at many more indices (by a factor of $|A|$).

2.7 Temporal Difference learning

Like Dynamic Programming, *Temporal Difference* methods (Sutton, 1988) update the value function based recursively on other estimates, making the approach suitable for infinite horizon tasks and on-line, interactive learning. But like Monte Carlo methods, no model of the environment is necessary. Temporal Difference learning thus captures the best of both worlds, and for this reason dominates the standard account. As usual, the aim is to estimate the value of a state in terms of (2.3).[6] Temporal Difference learning is so called because $\hat{V}_t(s)$ (note that we now use $\hat{V}$ because we are now dealing with an estimate of the value function.) is updated based on the difference between $\hat{V}_t(s)$ and $\hat{V}_t(s')$, where s' is the state encountered immediately after s. The basic Temporal Difference update rule is:

$$\hat{V}_{t+1}(s) = \hat{V}_t(s) + \alpha\Big[\{r + \gamma\hat{V}_t(s')\} - \hat{V}_t(s)\Big] \tag{2.13}$$

which is applied immediately after receiving reward r for moving from state s to s'. The expression in the curly brackets corresponds to the contents of the square brackets in (2.7), and represents the target of the update. This is just a recursive formulation of (2.3). The rest of (2.13) moves the current estimate $\hat{V}_t(s)$ towards this target by an amount proportional to the *learning rate*, $0 < \alpha < 1$.

Providing each state is continually visited under some policy, π, and the learning and exploration rates are annealed to zero according to the constraints of (2.14), then $\hat{V}(s)$ will converge on the familiar return of (2.3) for that state, and therefore $\hat{V}$ will converge to V^{π}. In essence, this is the approach used in the backgammon player of Tesauro (1994).

$$\int_{t=0}^{\infty} \alpha(t) = \infty \text{ and } \int_{t=0}^{\infty} \alpha(t)^2 \leq \infty \tag{2.14}$$

2.7.1 Sarsa

As has already been seen with the Monte Carlo method, if an environment model is not available, the value function, V, is insufficient for improving the policy. Therefore,

[6]Equation (2.3) is being used again because non-finite MDPs are now being considered.

in the *Sarsa* algorithm of Rummery and Niranjan (1994) the action value function, Q, is again requisitioned to give the Temporal Difference update rule:

$$\hat{Q}^{\pi}_{t+1}(s,a) = \hat{Q}^{\pi}_{t}(s,a) + \alpha\Big[\{r + \gamma\hat{Q}^{\pi}_{t}(s',a')\} - \hat{Q}^{\pi}_{t}(s,a)\Big] \tag{2.15}$$

where a' is the next action to be performed from state s' according to the current policy, π.

The theory is a simple extension of Dynamic Programming, based on Bellman equations for Q^{π} and Q^{*}.[7] Following the discussion so far, we can see that the repeated update of $\hat{Q}^{\pi}(s,a)$ towards $(r + \gamma\hat{Q}^{\pi}(s',a'))$ based on sample experience will yield (2.3), for the current policy.

Given Q^{π}, the policy can then be improved to exploit this information in exactly the same way as the Monte Carlo method — by choosing the action a in state s that maximises $Q^{\pi}(s,a)$. Through policy iteration, Q^{π} converges to Q^{*} and π to π^{*}, providing as usual that the environment is modelled as an MDP, the learning rate satisfies (2.14), all states are visited infinitely often in the infinite limit (using an ε-soft policy for example), but that exploration is eventually reduced to zero (see Singh et al. (2000) for convergence proof). Note that following the discussion of policy iteration, there is no need to wait for $\hat{Q}^{\pi}$ to converge on Q^{π} before updating π. In fact, here π is effectively updated after every single update to $\hat{Q}$ simply because π is based on the current $Q-values$. This particular Temporal Difference method is called Sarsa because the update rule uses s,a,r,s' and a'(Sutton, 1996). This and the following technique are referred to as *bootstrapping* because, unlike Monte Carlo, estimates of expected return are updated largely towards other estimates which themselves are based on further estimates etc.

[7]See Sutton and Barto (1998) for these equations. They are similar to the Bellman equations already encountered, and do not add anything to this particular discussion.

2.7.2 Q-Learning

Sarsa is actually a minor and recent adaption to one of the most theoretically important, and most popular RL methods known as *Q-learning* (Watkins, 1989), which uses the update rule:

$$\hat{Q}^{\pi}_{t+1}(s,a) = \hat{Q}^{\pi}_{t}(s,a) + \alpha\left[\{r + \gamma \max_{a'} \hat{Q}^{\pi}_{t}(s',a')\} - \hat{Q}^{\pi}_{t}(s,a)\right] \tag{2.16}$$

This is identical to Sarsa except that when considering the next state-action transition, the action a' is chosen that will *maximise* the next Q-value as opposed to choosing a' according to the current policy. This means that the policy being *evaluated* is closer to the optimal policy for the current Q-function (i.e. with no exploration) even though the policy being used for *control* may still be involved in exploration. Sarsa effectively models its own exploration as part of the dynamics of the environment, while Q-learning does not. Modelling the exploration may be useful if such exploration can profoundly affect the reward (see Sutton and Barto (1998), page 150 for an example).

Q-learning is shown to converge to an optimal policy under the usual assumptions (Watkins and Dayan, 1992), and it remains the most popular reinforcement learning algorithm because no model of the environment is required, it is intuitive, easy to implement, and can be run interactively with updates made immediately, as and when states are visited. These features make the algorithm suited to a wide variety of learning tasks. For example, Araujo and Grupen (1996) use Q-learning in a foraging task to map states to high level behaviours which are generated beforehand. Digney (1996) uses *nested Q-learning* to build hierarchical control structures for use in a grid-world environment. A particularly celebrated example of this Temporal Difference method is found in Mahadevan and Connell (1991), where a robot learns to find and push boxes within a behaviour based framework. Q-learning is also employed in Crites and Barto (1996), where an extension of the algorithm is used to discover a policy for efficiently dispatching lifts to minimise waiting times.

2.8 TD(λ)

To round the theory off neatly, the Monte Carlo and Temporal Difference methods can be shown to be special cases of a more general formalism, $TD(\lambda)$ (Watkins, 1989).

In Monte Carlo methods, the value of each state is updated towards the actual reward received from the first visit to that state to the end of the episode. In the Temporal Difference algorithm, the value function is estimated recursively in the sense that it is updated towards the immediate actual reinforcement plus the discounted *estimated* value of the next state (or state-action pair). TD(λ) is a more general algorithm which provides smooth control over the degree to which actual returns and estimated returns are blended to produce the target towards which the value function is updated.

Recall that in (2.13), $\hat{V}(s)$ was updated towards the *1-step corrected return*, but just as plausible are the 2-step, 3-step or n-step returns:

$$\begin{aligned}
&\text{1-step return} = r + \gamma\hat{V}(s') \\
&\text{2-step return} = r + \gamma r' + \gamma^2\hat{V}(s'') \\
&\text{3-step return} = r + \gamma r' + \gamma^2 r'' + \gamma^3\hat{V}(s''') \\
&\vdots
\end{aligned}$$

where $s, s', s'' \ldots$ is the sequence of states as they are visited, and $r, r', r'' \ldots$ is the sequence of rewards received on entering these states. If the trial length is finite, and n large enough to reach the end of each trial, then the n-step return is just a non-bootstrapping target as used in the Monte Carlo algorithm. Hence there exist a range of methods with Monte Carlo at one extreme and basic Temporal Difference at the other. It is a simple matter to combine these two extremes in a continuous manner by updating $\hat{V}(s)$ towards a weighted sum of n-step returns:

$$R_1 + \lambda R_2 + \lambda^2 R_3 + \ldots + \lambda^{n-1} R_n \tag{2.17}$$

for $0 \leq \lambda \leq 1$ (note the distinction between λ and γ!), where R_m is the m^{th}-step return from state s. Since the weights of the n-step returns should sum to unity (for $n = \infty$) in order to respect the estimate of (2.3), an appropriate normalisation factor is introduced so that (2.17) becomes:

$$(1-\lambda)R_1 + (1-\lambda)\lambda R_2 + \ldots + (1-\lambda)\lambda^{n-1}R^n \tag{2.18}$$

The term, λ, is the continuous parameter referred to in $TD(\lambda)$, which in its limits of zero and one represents 1-step Temporal difference and Monte Carlo methods respectively. Although this may seem like a rather contrived way of combining actual and estimated returns, it actually represents the theory underpinning an intuitively appealing and popular set of algorithms defined by the use of *eligibility traces* (Watkins, 1989). Such algorithms maintain a record of recently visited states and use this history to accelerate the passing of reward information across the value or action function. Versions of this algorithm also exist for Q-learning and Sarsa in the form of $Q(\lambda)$ (Watkins, 1989; Peng, 1993; Peng and Williams, 1996) and $Sarsa(\lambda)$ (Rummery, 1995) respectively. See Tesauro's backgammon player (Tesauro, 1992, 1994) for an application of TD(λ). See Sutton (1996) for an application of Sarsa(λ) and Araujo and Grupen (1996) for an application of Q(λ) to simulated robot control.

Although there is no principled analysis available, Sutton (1996) concludes that $0 < \lambda < 1$ is likely to be optimal with $\lambda = 0$ and $\lambda = 1$ empirically performing relatively poorly. In his backgammon application, Tesauro reports that: "...λ appeared to have almost no effect on the maximum obtainable performance, although there was a speed advantage to using large values of λ [corresponding to Monte Carlo]". Jaakkola et al. (1994), amongst others, have provided a convergence proof for $TD(\lambda)$.

Although a number of variants and extensions to the above algorithms have been proposed, the previous section provides as much history and theoretical background as is interesting and relevant to this thesis. The reader is referred to Sutton and Barto (1998) and Kaelbling et al. (1996) for a more thorough treatment. The focus now moves from the theory to the practice.

A brief notational comment is required at this stage. In the remainder of this thesis when we talk about 'Q-values' we will be referring to the *estimated* Q-values — i.e. the function, $\hat{Q}$. However, to simplify notation, we will omit the superscript and refer to estimates simply by using the function, Q. We will also adopt the simplifying notational convention of omitting the policy superscript, since there will always be an implicit assumption that we are estimating expected return for the current policy, and not the optimal policy. Furthermore, the term 'Q-value' will be used to refer to any estimate of expected return for state-action pairs.

2.9 Practical reinforcement learning

The theory provides the following: An iterative, incremental and interactive method that guarantees convergence of the value function, V, to either (2.2) or (2.3), under the assumptions that the environment is modelled as an MDP, every state is continually visited, and the learning and exploration rates are annealed appropriately. The value function, V, estimates the values of *states* of the MDP, which requires that the basic value function update rule (2.13) makes use of an explicit environment model. If the environment model is unknown, then the value function, V, is replaced by the action function, Q, which estimates the value of each state-action pair. Now the environment model is implicitly learned as part of the action function. By interleaving *policy evaluation* and *policy improvement*, an optimal policy is guaranteed to be found.

2.9.1 The assumptions

In practice the MDP assumption can rarely be met, since in many applications the sensory information fails to uniquely identify the state of the environment. This problem of *perceptual aliasing*, in which states are confused with each other, is exactly why escaping a maze is difficult, even for us. In general, the complexity and uncertainty of the real-world will make it impossible to satisfy the MDP assumption. Also note that this assumption is left unsatisfied when the environment is modelled as an MDP, but when this model is dynamic. If the state transition probabilities and reward function change over time, as may well be the case in a real-world problem, then convergence to such a moving target cannot be guaranteed.

The assumption that each state (or state-action pair) is continually visited can be satisfied by always maintaining an appropriate amount of exploration (as discussed in section 2.6). The convergence proof effectively requires that each state is visited an infinite number of times. In practice, trials must be of finite length, and some states may suffer particularly from under exposure, resulting in inaccurate value estimates. However, since the algorithms outlined above are interactive or *on-line*, the most frequently visited states will conveniently tend to have the most accurate value estimates.

The assumption of appropriate learning and exploration rates is easily satisfied in the limit of infinite trial length by the conditions of (2.14). However, in the finite case, finding a suitable set of learning rates that maximise performance is an empirical challenge.

Having established that, in practice at least, the criteria for convergence cannot be satisfied, the question now arises as to how well these algorithms perform when the assumptions are not met. Happily, the answer appears to be quite well. By choosing a suitable state representation the task can be made as close to an MDP as possible. Incorporating previous sensory data can also help to reduce the problem of perceptual aliasing. Judicious selection of the exploration rate can allow the assumption of continuously visited states to be at least partially satisfied, and in any case, the interactive nature of the algorithm suggests that accuracy will tend to reflect the exposure and therefore the relevance of different parts of the environment. The empirical selection of a suitable set of learning parameters also seems to be reasonably straightforward in the majority of cases. In addition, some evidence has been presented that other parameters, such as λ, may not have a huge impact on performance, and that satisfactory if not optimal values will be easy to find.

2.9.2 Delayed rewards

Mataric (1997) identifies two main problems that need to be addressed in reinforcement learning. The first is that of delayed rewards, or more generally, credit assignment. Although the theory provides a guarantee of optimality for infinite length trials (or an infinite number of finite trials), many practical applications, such as those involving physical robots for example, may be very restricted with respect to the number of environment samples that can be made. For this reason it is important that reward information propagates around the value function as quickly as possible. As an illustration of the problem, in Tesauro's backgammon application, TD-Gammon, the reinforcement of all states was zero except for the final state of a won game at which point the reward was one. This is beautifully simple, and requires a minimum amount of prior game knowledge, but hundreds of thousands or millions of complete games were needed to allow the reward of won games to propagate back to the early game states.

Mataric (1994, 1997) addresses the issue of delayed rewards by introducing *progress estimators* which provide a handcrafted continuous reward function which augments the reward information that is received at goal states. For example, a progress estimator might provide an estimate of the distance of an agent from a goal in a robot navigation problem. Progress estimators address the more general RL aim, identified by Kaelbling et al. (1996), of making the reward signal as local as possible. Breaking a task up into subtasks or a control system into behaviours, with each task or behaviour having its own reward function, is another approach to reducing the time between an action being taken and reward for that action being received (see Mahadevan and Connell (1991); Mataric (1994, 1997) for some examples). Tesauro's backgammon player exemplifies the problem of delayed rewards since the only information from the environment always comes on transition to a terminal state of the MDP. Attempting to provide as rich a reward signal as possible is an important part of encoding a task for an RL solution. Caution is advised though. Supplying handcoded intermediary signals to *shape* the learning process may result in the wrong behaviour being accidentally reinforced. Note that in the backgammon example, maximising the (rather weak) reward signal was *guaranteed* to maximise playing ability given enough training time. But consider what might have happened by 'enriching' the reward signal by a progress estimator designed to reward intermediate game positions that were mistakenly judged by an expert to be strategically advantageous.

Although this thesis is not directly concerned with delayed rewards, the issue is encountered at a number of points throughout the thesis. It is introduced here largely for completeness.

2.9.3 Large state spaces, and generalisation

The second of the two major problems facing RL application identified by Mataric (1997) is the possibility of large or continuous state or action spaces. Consider again the backgammon example, in which there are about 10^{20} distinct board positions, and therefore the same number of states. Representing the value of each board position explicitly is clearly impossible, so Tesauro used backpropagation to train a neural network to approximate the function, V. Note that because the environment model is known (in terms of the available successors to the current state), the value function V is sufficient for learning an optimal policy. The algorithm used is actually $TD(\lambda)$, but (2.13) also characterises the approach. However, at each state, instead of a table entry for $V(s)$ being updated, the network is trained towards the pair (I,O), where I is the input vector corresponding to the current board position, s, and O is the target inside the curly brackets (of 2.13). Because of the way the experiment was set up[8], the value function approximated by the network effectively mapped each board position to the probability of winning from that board position. Optimising the policy was then achieved by simply selecting the move from a list of legal possibilities that lead to the state with highest $V(s)$ according to the network. No exploration was built into the move selection because the dice throws themselves were considered to generate enough noise[9]. The system was trained against itself, so a large number of games could be played.

Generalising over the state space using a non-linear function approximator yielded excellent results for TD-Gammon, with the system learning to play backgammon to club standard. By hand-coding salient board features into the representation at each state, performance was improved to rival the world's best players. A pleasing addendum is that some of the opening plays learned by TD-Gammon went on to change the opening theory of the game.

[8]No discounting took place and recall that the reward was zero in all states except won positions, where it was one.

[9]The reader could refer to http:/www.funcom.com/lang_en/lang_en/games/backgammon/rules.html for an explanation of the game.

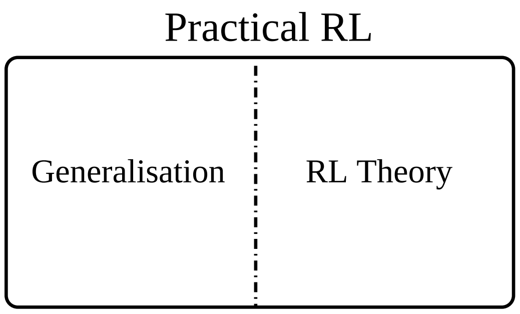

Figure 2.3: The independent relationship between RL and Generalisation theory. The diagram emphasises the point at which RL finishes, and where the responsibility for scaling RL primarily resides. In particular, a lack of ability in dealing with large or continuous state spaces is not an inherent problem of the RL theory itself. The latter provides a clear set of results within a clearly defined set of boundaries. The sophistication and potential applicability of RL is not limited by the theory, but by the techniques with which the theory is combined.

However, a fourth criterion for guaranteed convergence that has only been implicit so far in this chapter, is that the value or action function be represented explicitly as a lookup table. This drives another wedge between RL theory and its practical application since large or continuous state spaces will in general preclude this condition being met. Effective generalisation over large or continuous state and action spaces is the key issue facing real-world applications of RL, and this is the main focus of this thesis, particularly with respect to generalising over continuous action spaces. Although the lookup table assumption has resulted in further violation of the theory by practical considerations, at least Tesauro's TD-Gammon experiment suggests that we can still expect good performance providing appropriate generalisation techniques are used.

2.10 Generalisation techniques

In considering appropriate techniques for generalisation, the bridge has been crossed from the field of RL to the domain of statistics. Using a multi-layer perceptron to approximate the value function is only one approach to generalisation but any existing statistical technique for generalisation is a potential candidate for use on complex RL problems. Figure 2.3 illustrates the intended relationship.

An entire review of all generalisation techniques is clearly well beyond the scope of this thesis, but a representative sample will be considered shortly, as well as at various other places in the thesis. Therefore, to avoid duplication, only the briefest overview is given here and the reader is asked to accept a review of this subject as it is uncovered piece by piece according to the logical progression of the thesis.

2.10.1 Tiling the state space

Consider a robot learning problem in which there are two sensors each of which can take a value in the continuous range $[0, 1]$. The state space can then be represented as the unit square, and the task becomes one of dividing this space up into regions, with each region representing a discrete state of the standard RL problem. Deciding on the size, shape and number of these regions beforehand will generally be difficult, and the most obvious approach of using a high resolution grid superimposed over the entire space will result in an excessive number of states, particularly as the dimensionality of the space increases.

An alternative is to tile the space with overlapping regions, with each region corresponding to a hand-coded ‘feature’. Then any particular state can be characterised by the set of features in which it appears. This particular approach is called *tile-coding* and is also referred to as the *Cerebellar Model Articulation Controller (CMAC)* because it models Cerebellar functionality (Albus, 1975). This is a specific instance of a broader set of basic generalisation techniques described as *coarse-coding*, in which state and action spaces are carved up in advance. A second instance of a coarse-coding algorithm is the *memory-based function approximator* where a continuous Q-function is represented as a set of prototypical Q-values. The Q-value of any point in the state-action space is approximated by a suitable combination of the surrounding prototype Q-values, scaled for example by the distance between each prototype and the point in the state-action space being evaluated. Updating the Q-value of an arbitrary point in the continuous state-action space can be achieved by altering the prototype values according to their relevance, and if there happen to be no prototypes suitably close to the current position then one can be spontaneously created. Coarse-coding approaches potentially suffer from the fact that generalisation is not adaptive. For example, in standard memory-based function approximators, the prototype positions are fixed. See

Santamaria et al. (1997) for a description and comparison of coarse-coding techniques.

Another simple approach to generalisation used by Mahadevan and Connell (1991) in their famous box pushing robot application is to represent each state explicitly (in their case the state space was discrete), but to update not only $Q(s,a)$ for the current state and action, but also $Q(s',a)$ for all $s' \in S$ within a fixed *Hamming distance* of s. This makes updating the Q-function over a large state-space more tractable. However, in this example, the only way each state could be represented explicitly was by performing hand-coded dimensionality reduction on the robot's vector of sensor readings.

2.10.2 Dynamic generalisation

A more appealing approach is to construct categories on-line, based on and in response to the input data. One approach, which is considered at length throughout this thesis, is to use Kohonen's Self-Organising Map (SOM) (Kohonen, 1987) to model the distribution of the input data. In the case of the two dimensional state space considered above, a SOM could be trained towards the two-dimensional sensory vectors generated by the robot. Each unit of the map would then be usable as a discrete state in the standard RL problem, with the SOM dynamically discretising the space with a variable resolution that reflects the robot's exposure to different parts of the environment. In work that is considered later in section 4.4, Sehad and Touzet (1994) and Touzet (1997) use a SOM to map the combined state-action-reward space. While the approach is favourably compared with a number of other representational techniques for a robot learning problem, the comparison is not detailed enough to conclude anything other than that the SOM is an interesting and potentially effective approach. The use of a Kohonen map to generalise over continuous state and action spaces is discussed and analysed at length during the course of this thesis.

In a similar vein, the *k-means* clustering technique (see Lloyd (1982)) could be used, or the *Adaptive Resonance Theory* network of Carpenter and Grossberg (1987b) which adopts a more constructive approach to the plasticity/stability tradeoff. Li and Svensson (1996) choose an ART network over a Kohonen network to represent the state space in a robot navigation problem. Their decision is based on the perceived inability of the latter to operate simultaneous learning and operation phases. However, results from chapter 5 suggest that this conclusion is invalid.

A dynamic approach adopted in Chapman and Kaelbling (1991) utilises *decision trees* in which an initially small number of states covering the entire space are iteratively split into smaller and smaller regions until each region behaves consistently with respect to the reward signal. This approach benefits from considering the reward information as well as the input data in generating categories, but it may not be suited to dynamic environments since the splitting is a one way process that could potentially result in too many regions being created.

In all these approaches, the emphasis is on dissecting the state space so that the value or action function can be maintained as a look-up table. Note that it is possible to perform the category construction and reinforcement learning processes in parallel, providing it is accepted that the categories underpinning the RL algorithm will change during learning. Tolerating moving states makes a further mess of the theory, but in practice such systems may still work well providing appropriate consideration is given to relative learning rates. This issue is considered in more detail in chapter 6.

2.10.3 Backpropagation

Other techniques directly approximate either the value function, or even the policy itself, thus side-stepping the need to maintain an explicit value function. Tesauro's TD-Gammon made effective use of the backpropagation algorithm to train the weights of a multi-layer perceptron (MLP) to perform powerful non-linear generalisation over the state-space. Similarly motivated is the *Complementary Reinforcement Backpropagation Algorithm (CRBP)* of Ackley and Littman (1990), in which a binary reward signal (intended to indicate whether the action is 'right' or 'wrong') is used to train a backpropagation network directly towards appropriate state-action pairs (see section 4.7. The approach is adopted by Ziemke (1996) in another robot learning problem to map robot sensors directly to optimally rewarded motor outputs. The scalability of CRBP is not clear, since the reward and outputs are binary. There is also no explicit interpretation of the action function which means the system operates as a black box.

The use of backpropagation is considered in more detail in chapter 8. Of particular interest is its relative robustness to the curse of dimensionality. For example, the input vectors of TD-Gammon had 198 dimensions!

2.10.4 Other techniques

A range of other techniques could also be conceived for generalising over the input space including the use of Radial Basis Networks (Powell, 1987), the GTM algorithm (Bishop et al., 1998), or an auto-associative MLP. A number of ad hoc techniques have also been suggested, such as the clustering algorithm of Mahadevan and Connell (1991) in which Q-values are grouped according to proximity in state-space *and* similarity with respect to the reward under each action. One interesting aspect of that work is that the state space is effectively decomposed differently for each action.

A complete comparison of these generalisation techniques with respect to the full range of reinforcement problems would be impossible. Different approaches will be more appropriate than others in different applications. A discussion of the applicability of some of these techniques with respect to specific problem features is given in chapters 6 and 8. In the meantime a number of relevant issues can be identified. For example, is generalisation adaptive or fixed? Is it linear or non-linear, supervised or unsupervised? Is the generalisation performed with respect to the reward function, the input data, both or neither. Also, a relevant question for non-stationary environments is how the tradeoff between plasticity and stability is addressed. A key distinction considered in chapter 8 is whether the representation of the state-space is *local*, as in the case of the Kohonen map for example, or *distributed* as in a backpropagation network. This may have implications when considering non-stationary environments, high dimensional state spaces, maintaining multiple actions, interpreting behaviour, and diagnosing faults. Other considerations include the resources required by the algorithm, the robustness and complexity of the algorithm, and potential for parallelisation, all of which may be particularly relevant in interactive domains such as robot learning. The consequences of pathological behaviour are also relevant. For example, what are the implications of getting stuck in a local minimum when using backpropagation, or producing twisted maps when using a SOM. All of these issues are encountered at various points throughout the thesis.

2.11 Summary

The theoretical foundations of reinforcement learning have been presented along with an account of the evolution of the most popular algorithms, including TD(λ) and Q-learning. Although in practice all of the assumptions required to guarantee convergence of Temporal Difference methods may be violated, performance is empirically found to be robust and good performance is usually achieved.

The problems of delayed rewards and continuous or prohibitively large state and action spaces are not failings of RL itself, but are inherent difficulties of the complex problems being addressed. The solution is not to say that RL is inadequate so let us invent something else, but instead to specifically address these issues, and indeed this is where much of the current research lies.

For any of the difficulties named above, or just because of an inadequate reinforcement signal, some problems may turn out to be too hard to solve using just the components of RL considered so far. In these more interesting cases a number of other ideas may need to be considered. As Kaelbling et al. (1996) note, almost all the advanced and interesting applications of RL utilise some form of innately specified knowledge about the task, be it cleverly crafted state representations (for example (Tesauro, 1994)), built-in behaviours (Maes and Brooks, 1990), reflexes (Li, 1999), pre-specified coordination of learning modules (Mahadevan and Connell, 1991), handcrafted progress estimators (Mataric, 1994, 1997), inbuilt heuristics for exploring actions (Wedel and Polani, 1996), and any number of assumptions about the nature of the reinforcement signal, or the environment model.

It is generally assumed that we are born with many innately specified tendencies, but that we also perfect our skills by increasing our knowledge of the world through trial and error interaction with our environment (Karmiloff-Smith, 1995). Looking to the animal world for inspiration yields other ideas including learning through imitation (see Hayes and Demiris (1994), and Price and Boutilier (2000) for RL-based imitation learning) and *shaping* (see Dorigo and Colombetti (1994) for an RL example) where an animal is trained incrementally on successively harder and harder versions of the task. With respect to domain specific learning, Gallistel et al. (1991) observe that animals have strong predispositions to learning certain kinds of associations. For example, a pigeon can be trained to peck a button for a food reward, but not to flap its wings. Conversely, it may be trained to flap its wings to avoid a shock, but not to peck. It is easy to see how biasing the kinds of associations that may be made in this way can greatly simplify the learning process. Even human development seems very constrained with certain things being learned at very specific times and being subject to specific constraints, prerequisites and processes (Karmiloff-Smith, 1995).

All this indicates that although we cannot expect to solve all problems by RL alone, the technique has still established itself as a valuable tool for our hardest endeavours. Designing RL-based solutions to interesting problems goes far beyond just picking one of the theoretical approaches plus a suitable set of parameters. It is the art of performing appropriate generalisation, maximising the information in the reward signal, and knowing how to bring all available information to bear on the problem, that transforms the theoretical and well understood foundations into a set of practical and exciting techniques.