Cooperation through Reinforcement Learning

By Philip Sterne
Computer Science Honours
2002
Rhodes University

Submitted in partial fulfilment of the requirements for the degree of
Bachelor of Science (Honours) of Rhodes University.

Abstract:

Can cooperation be learnt through reinforcement learning? This is the central question we pose in this paper. To answer it first requires an examination of what constitutes reinforcement learning. We also examine some of the issues associated with the design of a reinforcement learning system; these include: the choice of an update rule, whether or not to implement an eligibility trace.

In this paper we set ourselves four tasks that need solving, each task shows us certain aspects of reinforcement learning. Each task is of increasing complexity, the first two allow us to explore reinforcement learning on its own, while the last two allow us to examine reinforcement learning in a multi−agent setting. We begin with a system that learns to play blackjack; it allows us to examine how robust reinforcement learning algorithms are. The second system learns to run through a maze; here we learn how to correctly implement an eligibility trace, and explore different updating rules.

The two multi−agent systems involve a traffic simulation, as well as a cellular simulation. The traffic simulation shows the weaknesses in reinforcement learning that show up when applying it to a multi−agent setting. In our cellular simulation, we show that it is possible to implement a reinforcement learning algorithm in continuous state−space.

We reach the conclusion that while reinforcement learning does show great promise; it does suffer in performance when extending it to the multi−agent case. In particular the quality of solutions arrived at by a reinforcement learning system are suboptimal in the multi−agent case. We also show that the algorithm used for continuous state−space, does not achieve optimal performance either.

Acknowledgements:

This paper has single−handedly quadrupled the number of words I have submitted to Rhodes University in my quest for an honours degree. As such it came as a great shock to my system, and without the suggestions of my supervisor Prof. Shaun Bangay, or the insights provided by Prof. Mike Burton, this paper would have drastically diminished in quality. I must also thank the National Research Foundation for providing funding which made this research possible.

And last, but certainly not least I wish to thank Kerry Kyd, whose support through frustrating times (?'they just won't learn!?!'?), as well as invaluable proof−reading skills have eased the long marathon that has been my project.

Table of Contents:

1 Introduction

Can cooperation be learnt through reinforcement learning? Reinforcement learning is a relatively new field in artificial intelligence, and we use it to find out if it is possible for two reinforcement learning agents to learn to cooperate with each other.

We first explore reinforcement learning as a theory in itself. We assume no prior knowledge of reinforcement learning and introduce the topic gently. We examine two tasks in detail; the first is a simplified blackjack player, which we use as a means of putting the theory into practice. Most reinforcement learning algorithms contain constants and we examine the effects of different values for these constants on the performance of the reinforcement learning algorithms.

Our second task is an agent that learns to run through a maze. This task is somewhat harder than the first as it has a larger number of states, and possible actions. It also requires us to implement a full reinforcement learning algorithm.

Assuming that we now understand some aspects of reinforcement learning, we now turn to the problem of cooperation. In some instances having multiple simple agents perform a task is preferable to having a single complex agent. However this requires that the agents are able to cooperate with each other. We first examine some of the work done in theoretical work done in game theory, after which we examine two tasks in detail.

In our traffic simulation we explore the feasibility of learning cooperation between multiple agents, as well as any side−effects of having multiple agents learning form each other.

The final task is a cellular simulation. Initially we had hoped to explore the issues of cooperation, in this task as well, however we found the complexity involved in continuous state−spaces hindered us. Instead we discuss some of the issues involved with extending the state−space from the discrete case to a continuous case, showing that it is possible to do in a straight−forward manner, yet there are performance penalties that affect the learning rate.

2 Background

2.1 Justification

Much of the project presented in this paper relies heavily on the framework of Reinforcement Learning. To the reader that is unfamiliar with Reinforcement Learning this chapter should prove invaluable to the understanding of the project.

2.2 A brief history of Reinforcement Learning

The earliest beginnings of reinforcement learning are in psychology, one of the most well known examples of which is Pavlov’s dogs. In his experiment he rings a bell, and then proceeds to feed the dogs. In time the dogs begin to associate the ringing of the bell with the feeding. This is proved when Pavlov rings the bell but doesn’t feed the dogs. The dogs show great expectations of being feed and are clearly confused when the meal isn’t served.

From there, experiments have became more focused on showing that animals can learn new behaviours if they are suitably rewarded. The work of B.F. Skinner in Operant Behaviour using mainly pigeons and rats set up a fairly standard framework. He showed that these animals could learn from a ’three−term contingency’, namely an event in an environment lead the creature to a response, and subsequently a consequence. By changing the consequences we are able to teach these animals new behaviour (O’Donohue and Ferguson, 2001).

In time this method of learning has been seized upon by Artificial Intelligence researchers they have developed a formal framework through which reinforcement learning could be studied. One of the main advantages of Reinforcement Learning is that there is no requirement of understanding the dynamics of the system. For example to play Backgammon we simply have to define all the valid states of the game, as well as the necessary reinforcement signal (Barto and Sutton, 1998:261). The reinforcement signal is quite simple to define, for example +1 for a win, 0 for a draw and −1 for a loss. The system can then study previously played games, or experiment by playing against another system. All this experience is then used to improve how well it can play Backgammon. This sort of black−box modelling can prove very powerful, as Tesauro’s

Backgammon system has even discovered strategies that some of the world’s best Backgammon players are now adopting. Before we give formal definitions we will provide some more examples of reinforcement learning tasks.

These examples are designed to give one a feel for what constitutes a reinforcement learning problem, as well as providing examples with which we can refer back to in the rest of this chapter. To this end we have provided three different examples so that the more common variations of a reinforcement learning task are present.

2.2.1 The Pole Balancing task

A cart is placed in the middle of a track, with an upright pole on it. The task at hand is too learn how to balance the pole by either moving the cart left or right on the track. The task ends when the cart hits the end of either side of the track, or when the pole falls over. Here we are interested in making the experiment last as long possible. (Taken from Harmon, 1996).

2.2.2 Maze Solving task

Here an agent is placed in a maze, with one or more exit points. The agent must learn to navigate through the maze if the agent is given only its current location. The only actions available to the agent are ’North’, ’South’, ’East’ and ’West’. After exploring the maze the agent must then be able to find as short an exit path as possible. (We cover this example in depth in chapter Maze World.

2.2.3 Traffic Lights

Suppose we have to control a traffic light, with the aim of making traffic flow as smoothly as possible (Thorpe, 1997). For optimal results we should also take into account a traffic light’s position, as some positions would require different timings from others. This means that a general algorithm is not optimal for traffic control in an arbitrary position. It is possible to accurately simulate the intersection’s traffic flow on a computer. Every time traffic gets stopped in the simulation then the controller gets a penalty. The controller must then try to minimize the penalties it receives.

2.3 Definitions

Formally we may divide the reinforcement learning task into several distinct objects. At a high−level the task consists of an agent interacting with an environment. The feedback of the agent’s performance is given in the form of a reinforcement function. The agent must have some perception of the environment. The agent can try to create a model of the environment. But in most cases it learns simply the best action in each state.

2.3.1 State

An environment consists of several states. In the maze−learning task the states would consist of positions in the maze. In a certain state only some actions may be available, e.g. − There may be a wall blocking the north action in some states. In the Pole−balancing task the states could consist of the interval [−1 ... 1], with each number representing a position on the track, together with a number representing the angle between the Pole and vertical. However it turns out that continuous state variables are very difficult to deal with so instead the continuous variable is changed into a large number of discrete states. There are methods available for dealing with continuous state variables and those will be dealt later on in this paper.

2.3.2 Actions

Rather intuitively an action has some effect on the state and once an action is performed a new state is observed. However after an action has been taken then reinforcement is provided. These actions need not be deterministic, i.e. they are allowed to have probabilistic outcomes. In other words if we are in state $S_{1,}$ when we perform action $A_{1,}$ 70% of the time we might end up in State S_2 and 30% of the time we might end up in state S_3. For example in a game of cards, one might ask for another card, and then a wide range of possible cards might be dealt. Each of these possibilities will lead us to a new state.

2.3.3 Reinforcement Function

This is a mapping that accepts a state and action pair (S, A) which then gets mapped to a real number. Very often the only a subset of real numbers are used, e.g. in the Pole−Balancing task each step might result in a reinforcement of zero being given, with a reinforcement of −1 being given for the terminal states (i.e. the pole falls over or the cart hits the end of the track).

For the maze learning task a reinforcement of zero could be given for every non−terminal move, with a reward of +1 being given when an agent moves into an exit.

2.3.4 Value mapping

This represents the agent's attempt to predict the expected output of the reinforcement function. This value mapping gets more accurate as more information is obtained. However since the transition from state to state may be stochastic we may only be able to obtain an average of the reinforcement obtained. Increasing the accuracy of the value mapping is the crux of the problem as if one obtains a perfectly accurate value function (so that it predicts the reinforcement function with 0% error), then it is an easy task to obtain the optimal action in any state. As an example the value of the maze−solving task, for states near the exit will be considerably higher than for states far from the exits. In this way an agent choosing between an action that will take them to a low−valued state (i.e. far from the exit) or a state with a high−value (i.e. closer to the exit) they can pick the correct action, by simply choosing the higher valued state.

Now this sounds as if all the actions that will be taken are greedy, i.e. they consider only one step in advance. However as we shall see tasks for which greedy solutions are non−optimal can still be solved near optimally through reinforcement learning. Moreover the final result will produce the same results and we will only need to do a greedy search.

2.3.5 Policy

This is inextricably bound to the value mapping. One can generate a policy from the value mapping quite easily. A policy dictates what action should be taken in any given state. The optimal policy of an agent is defined as the mapping of states to actions that maximize the long term reinforcement signal. The optimal value mapping can also be defined. Generally we perform the action that is predicted by the value mapping as being the optimal one in a given state.

2.3.6 The Markov Property

The Markov property is an important assumption that simplifies many calculations in approximating the optimal policy. A system has the Markov property if past histories aren't important. This means that the reinforcement function doesn't depend on anything other than the current state. A really good example of a task that has the Markov property is chess, to consider the best move for a given chess position we don't have to know anything about the preceding moves, we can instead focus on the current position.

An example of a task that isn’t Markov is Poker (Sutton et al. 1998), here knowing previous histories of players can prove important, for example knowing if someone has a history of bluffing can prove very useful.

2.3.7 Dynamic Programming

Before we define what Dynamic Programming is, we need to define the term Markov Decision process as it is a central part of Dynamic Programming. This allows several simplifications, in that to determine the optimal policy we need only to consider the current actions and policies. A Markov Decision Process (MDP) consists of an environment with a finite number of states, and a clearly defined reinforcement function with the Markov property. Then Dynamic Programming can be defined as:

> ’The term ‘‘Dynamic Programming" (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).’ (Barto and Sutton, 1998).

Many of the algorithms used in DP can be transferred to reinforcement learning tasks with some modification. The biggest difference between DP and RL is that Dynamic Programming considers the environment as a whole (Bellman, 1962:297 − 319). It assumes that all the information concerning rewards and transitions are available beforehand, without requiring any agent to interact with the unknown environment.

Reinforcement Learning owes a lot to dynamic programming and in particular to Richard Bellman. It is his optimality equation, which has been used throughout the theory of reinforcement learning to derive the various convergence proofs, and update rules. (Bellman, 1962:301)

2.4 Fitting it all together

Here we cover several aspects and approaches common to many reinforcement learning algorithms.

2.4.1 State values vs. State−action pairs

When we are modelling the system, we might have a fairly good view of how the system works but not a detailed complete description, or in most cases the complete description is far too complex and detailed to be coded efficiently. By a complete description we mean that if there is any randomness in the system that the probability distributions for each random element can be given.

For example if we are playing blackjack, and we know which cards we hold, as well as the card that is displayed by the dealer, then that knowledge affects the probability distribution associated with the next card that will be dealt. In this way we would be able to work out the optimal behaviour from expected values, the best action in each state would be the one the maximises the expected value of the next state.

More often though rather than requiring such a complete description of the environment it is easier to simply keep track of state−action pairs denoted by $Q(s,a)$. By state−action pairs we mean the Cartesian product of all legal states with all of the legal actions. Thus rather than maintaining values associated with each state and then working out the optimal policy by working out the estimated value of the successor state for each action and each state, we can simply maintain separate estimates for each state−action pair.

Notice that we then do not require any probability distributions since a simple average would give us the expected values. In fact if we maintain a Q−table then the amount of knowledge about the environment is minimal, since we only need to know the legal states, as well as the legal actions associated with each state. Effectively then we have a black−box modelling tool, where we can treat the environment as a black−box, and we should still obtain near−optimal solutions. (Near optimal only because in the convergence proofs we require that each state, and action be tested an infinite number of times, which clearly isn't practically possible)

Because of these strengths associated with Q−tables we will use them extensively, rather than maintaining value tables.

2.4.2 Generalized Policy Iteration

Generalized Policy Iteration refers to the process of starting off with a fairly random Q−table. The values that we would choose would be rather small so that there isn't a significant bias towards any actions. This Q−table can then generate a (random) policy. By following this random policy we then obtain new values for our Q−table. Once we have updated our Q−table then we can generate a new policy from it. Obviously we can repeat this as many times as we like.

This iterative process, will continue until the Bellman optimality equation is satisfied, after which any more policy iterations will not be able to improve things (Bellman, 1962: 304)

2.4.3 Exploitation vs. Exploration

Initially we start off knowing very little about our environment, thus our Q−table is almost useless in the beginning. Thus it pays not to follow it too closely, but rather to try actions that do not have any information about them (Thrun, 1992). In this way we can gather new information which leads to a better understanding of the environment. However after many interactions with the environment such an exploratory move is unlikely to be an optimal move, so we would rather follow the existing policy. This plasticity is rather similar to simulated annealing, and can be implemented by simply picking a random action with probability ?, add then reducing ? over time. It is one of the necessary requirements for convergence to an optimal policy that this exploration be present since every state−action pair be tested an infinite number of times.

2.4.4 Optimism

An easy way to ensure that lots of exploratory moves are taken in the beginning is to set the Q−table for those moves to be high. As new experience comes in the value functions are then set to a more realistic level, however the other states are still optimistic, so the next time the state is encountered another untested optimistic action will be taken (Kaelbling, Littman and Moore, 1996:10)

One problem with this is that as we shall see in Chapter Maze World there are some situations in which there is only one solution, so once this solution is found then the next time we deliberately favour untested solutions, however since the only solution has already been we effectively trap ourselves. One way to avoid this is to update the Q−table as we gain new experience, this will lead us to experience all other possible states repeatedly until we can conclude that no other path is available. However in this situation we rather avoid using an optimistic Q−table.

2.4.5 Temporal Difference Learning

A large problem associated the learning of complex multi−stage tasks is that many of the actions that occur are all responsible for the success of the task. E.g. in a maze a whole sequence of actions are necessary to reach an exit. However when one doesn’t have perfect knowledge of the environment then how does one determine which moves were responsible for the completion of the task, and which moves actually hindered the completion of the task?

This is called ’the problem of credit assignment’ and there aren’t any algorithms that remove this problem totally, however one of the ideas that best deals with it is temporal difference learning − TD(λ) (Sutton, 1988). Here one uses a simple yet very powerful idea, when one receives a reward it shouldn’t just be associated with the previous state; action pair, but rather it should be distributed (in some way) among the all previous actions. In this way actions that lead to good states (i.e. states where potential rewards are high) are also rewarded. Before we can give the full description of this distribution we need to talk about discounting.

Discounting makes sense on an intuitive level, since we should prefer an immediate reward over a delayed reward. To give an example if we were offered R100,000 rand today or R120,000 tomorrow which would we take? Obviously most of us would wait an extra day for R20,000. However if we were offered R100,000 either today or tomorrow then we should all choose to receive the money today. Mathematically we can define a discounting factor ? as the value in the range [0...1] for which there is no difference between an immediate reward of ?, and a delayed reward of 1 after 1 time−step.

Now one can define a 2−step reward or a 3−step reward, or an n−step reward. A n−step reward is simply: $R' = R_0 + \eta R_1 + K + \eta^n R_n$. Where R_0 is the current reward and R_i represents the i’th reward received in the future. Since we do not know what rewards we will receive then we have to maintain an eligibility trace which we will discuss further in Eligibility Traces.

Temporal difference learning then goes a step further and weights these rewards further, multiplying by a factor of λ, where $\lambda \in [0..1)$. This then is known as TD (λ) learning. Since we deal exclusively with TD (0) methods, this extra λ factor simply disappears, and we will leave general TD (λ) learning out of this discussion.

This represents a slight shift, from trying to predict the immediate reinforcement; we are instead trying to predict the long term reinforcement that we will receive. Notice that once we have a reasonably accurate Q−table then the choices we make are based only on local information (i.e. we act greedily) however the behaviour that we produce is optimal for long−term solutions. As such it is one of the more powerful methods available.

2.4.6 SARSA

We now present a method of control that is widely used for its good performance and easy implementation. It is from (Sutton, 1988) and can be used as a TD (0) method, or as a TD (?) method. The TD (0) updating equation is given below:

Eq 2.2.0 SARSA update rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r - Q(s,a) + \eta Q(s',a')]$$

Where:

Q(s,a)	− is the previous state−action value
r	− is the reward given
?	− is the discounting factor (in the range [01])
?	− is the weighting factor (in the range [0...1])
Q(s',a')	− is the successive state−action value

The name SARSA is derived from the fact that the equation depends on the quintuple (s,a,r,s',a'), which we found amusing. It is an on−policy form of learning, which means that the updates are obtained from direct experience, and are dependent on the policy being followed. There are off−policy methods, which do not depend on the policy being followed, most notably Q−learning which will be discussed in the next section.

The form of the equation is quite interesting and can be explained on an intuitive level. If we manipulate Eq 2.2.0 then we can obtain:

Eq 2.2.0 − Manipulation of SARSA

$$Q(s,a) \leftarrow Q(s,a)(1-\alpha) + \alpha[r + \eta Q(s',a')]$$

From this we can see that the equation is weighted average of the old Q−value (which is an estimate of long−term reward) with the new reward received as well as an estimate of the discounted future rewards to be received.

In section Different update rules, we compare the performance of a similar update rule (Monte Carlo) with the performance of such a weighted rule against the performance of a normal average.

2.4.7 Q−Learning

Here we present an algorithm that is noticeably similar to SARSA. The difference is that while SARSA is heavily dependent on the policy being followed, Q−learning attempts to learn independently of the policy being followed. The TD (0) update is as follows:

Eq 2.2.0 Q−learning update rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \eta \max_{a'} Q(s',a') - Q(s,a)]$$

Its name derives from the fact that it attempts to learn from the Q−table, rather than the experience. It tries to be policy independent since it looks at the maximum value for all possible actions in the current state. While we might want to explore and take an action for which the state−action pair is less than the maximum, we should still update the previous action with the maximum value, since it represents the best we could achieve. This means that both SARSA and Q−learning converge to maximums, however they represent different optimal behaviours.

However this might not necessarily be accurate since it doesn't take into account the likely consequences, only the best consequences. Barto and Sutton (1998:149) present a fairly detailed description of the difference in performance between SARSA and Q−learning, by examining a task where the best possible outcome is 'dangerous' (there is also a good probability of a large penalty). In this case SARSA learns to be safe, since it maximises expected reward, while Q−learning still takes the dangerous route, and its performance is worse than that of SARSA's.

In light of this susceptibility we do not to use Q−learning, but rather other methods.

2.4.8 The importance of ?

Both Temporal Difference learning and Q−learning use ? in their update rules. What is the significance of ?? ? represents the value we place on new experience. If we know that we are in a static environment then we can decrease the value of ?, in effect lessening the value of new experience, as we feel that we understand a significant amount of the environment. However there is a trade−off as one decreases ?. As ? decreases we are able to estimate the value of a state−action pair to a higher accuracy. However as a result of the increased accuracy the policy becomes more inflexible, and unable to adapt to new changes in the environment.

Moreover if ? is decreased too quickly then the agent's estimates of rarely encountered states will have little or no meaning as they might have been encountered once or twice, but updated using a very low ? value, so that the value doesn't reflect the true values.

On a conceptual level though we feel that by keeping this responsiveness in the agent's policy that the agent will not fall into sub−optimal policies, but will be able to find the true optimal policy. As such, we choose not to decrease ?, although it might improve performance, through the increased accuracy of the state−action value estimation.

2.4.9 Eligibility Traces

As we mentioned in section Temporal Difference Learning, the updating rule requires that we have the future rewards so that we are able to reinforce correctly. A much simpler, yet equivalent, way of doing things would be to distribute the rewards backwards. If we receive a large positive reward the we can say that all those actions leading up to this reward should be reinforced. An eligibility trace is simply keeping track of which state−action pairs have been visited.

However there are two different types of eligibility traces, and they differ in what action to take after a state−action pair is repeated.
Accumulating traces: if the same state−action is encountered then the eligibility trace has one added to it.
Replacing traces: if the same state−action is encountered then the eligibility trace for that state−action is reset to one.

We also need to maintain how long ago it was visited, and that is simply done using discounting. If ? represents a table the same size as the Q−table, then

$\delta \leftarrow \lambda\delta;$

Eq 2.2.0 Replacing Trace updates

$$\delta(s,a) \leftarrow \delta(s,a) + 1$$

Eq 2.2.0 Accumulating Trace updates

$$\delta(s,a) \leftarrow 1$$

When a reward is received:

Eq 2.2.0 Updating the Q−table

$$\forall(s,a) \mid \delta(s,a) \neq 0 \Rightarrow Q(s,a) \leftarrow Q(s,a) + \alpha[r - \delta(s,a)Q(s,a)]$$

If we maintain an eligibility trace with decay factor λ then SARSA, becomes a TD (λ) method. λ as a parameter controls how far−sighted one would want the agent to be. For values close to 0 the agent attempts to maximise short−term gains, while for λ close to 1, the agent is more far−sighted and looks at long term gains.

The initial focus of early reinforcement learning research used accumulating traces. Singh and Sutton (1996) give an extensive comparison of the performance of both types. They conclude that in most cases a replacing trace has a better learning rate. So we have used replacing traces in our work. In section The Reinforcement Learning Framework, we give reasons why accumulating traces are inappropriate for that task.

2.4.10Monte−Carlo control

This is the final learning method we present. It is also forms a basis for the method we use for most of the tasks in the forthcoming chapters. While SARSA, updates the Q−tables after every action taken, in Monte−Carlo control we update the Q−tables after every episode. (This means that it isn't suitable for continual tasks, i.e. tasks for which there is no clear end) Since the updating is done only once it means that we must maintain an eligibility trace, for the whole episode. The updating rule is also based on the more intuitive measure of maintaining an average.

However as we shall see in Different update rules this update rule doesn't appear to perform quite as well as the weighted average updates used in SARSA, and Q−Learning. For this reason we modify our learning method and use a hybrid of SARSA and Monte−Carlo control.

2.4.11Summary

We have presented several different forms of reinforcement learning. Each has its own advantages and disadvantages in terms of learning rates and computational costs. SARSA generally has an excellent learning rate, although it requires updates after every action taken (which if an eligibility trace is implemented, can be computationally expensive). Q−learning attempts to learn independently of the policy it is following, but that can lead it away from the desired behaviour (optimizing the average reward). Monte−Carlo control has the advantage of updating only after an episode has passed, but this appears to slow the learning rate slightly.

The update rules either average all the previous rewards, or they weight the new experience by a fixed percentage. As we shall see this weighting appears to be more responsive to our learning tasks.

We will then proceed by implementing an updating rule that updates only at the end of an episode (from Monte−Carlo control) yet weights new experience by a fixed percentage (from SARSA, and Q−learning). We feel that this hybrid rule will combine the advantages of a low computational cost, while still keeping a fair learning rate.

2.5 Continuous state−spaces

A rather large jump in complexity occurs when we consider continuous state−spaces (Thrun and Möller, 1991). In part this is due to the fact that our table based methods cannot be directly used in continuous variables. Of course we can simply partition the continuous variable into large discrete blocks, which has been done before, and quite successfully (Keerthi and Ravindran, 1995: 12 − 13). Another option is to consider function approximators, such as neural networks, or multi−dimensional splines.

In the final task that we consider we do attempt to use a neural network, although things don’t quite go as planned. We also attempt to extend to continuous action spaces, and then use some numerical methods from multi−variate calculus, to work out the optimal actions predicted by the neural networks.

2.6 Related Work

While in the preceding few pages we gave extensive theory, in this section we show that reinforcement learning is an active field of inquiry, with several new and promising ideas being explored.

2.6.1 Feudal Reinforcement learning

Dayan and Hinton (1993) researched the possibility of having higher−order reinforcement learning systems. They use the term mangers, for a reinforcement learning system whose actions set goals for another reinforcement learning system. In this way it is possible to create an agent that learns about a system at a more abstract level, and then sets a goal for an agent that is working at a more concrete level.

They show that for certain tasks this abstraction can improve the performance dramatically. However Schmidhuber attacks this strongly, saying that for certain tasks the performance of a hierarchical based system can be lower than a normal system. (Schmidhuber, 2000:2 − 3)

2.6.2 Sharing sensation in Multi−agent systems

Is it possible to share experience directly between two agents? This question is similar to the questions on cooperation posed by our paper in chapter The theory of cooperation, although here the communication isn't learnt, instead we share direct experience from one agent to another. Tan (1993) examines this and shows sharing direct experience, or even just sharing information can lead to improvements in the learning rates, and final performances of the agents.

2.6.3 Robot World Cup

An interesting competition is held regularly, to play robotic soccer. Here a number of robots have to learn to coordinate their actions to play soccer correctly. Reinforcement Learning is playing an influential part in this competition, as it allows one to define performance simply in terms of possession or goals scored.

A wealth of papers has been written detailing experimental findings, as well as experience, in this multiple robot domain. This research definitely includes the problem of learning cooperative behaviours such as collision avoidance, and actual soccer strategies. For a more detailed look at the Essex Wizards which have performed well at these competitions see (Hu and Kostiadis, 1999) which focuses on the reinforcement learning or (Hu, Kostiadis and Liu, 1999) which looks at the robotics involved.

2.6.4 Imitation in multi−agent systems

If an agent already has knowledge of the environment then it would pay a newcomer to imitate the experienced agent. This imitation can give the newcomer a good idea about how to behave in the environment. Boutilier and Price (1999) have shown that this strategy can lead to a large improvement in the learning rate, as well as being extendible to multiple mentors.

2.6.5 RHINO

RHINO (Bücken, Burgard *et al.* 1998) is mobile robotic unit that is capable of moving around indoors and discovering its environment. It maintains an internal map as a means to navigate from one point to another. This map is generated efficiently through reinforcement learning methods. Moreover these reinforcement methods can encourage the exploration of unknown areas. RHINO also needs to perform image recognition, so that it is able to navigate through the environment without colliding into anything. (Bücken, Burgard *et al.* 1998: 20) recommend using RHINO as a robotic tour guide, although we feel that as time progresses this sort of technology will be more ambitious, leading into things such as automated garbage cleanup, or exploration in hazardous areas.

2.6.6 Markov games

Some work has been done in applying reinforcement learning to Markov games (Littman, 1994). A Markov game consists of two agents with mutually opposed interests; this means that one agent’s loss is the other’s gain. This means that a Markov game has no room for cooperation; it is only interested in competitive behaviour. We are more interested in trying to learn cooperative behaviours. But this research could lead to a fuller understanding of the work by Tesauro (Barto and Sutton, 1998:261) where through self−play a system is able to play Backgammon at world−class standards.

2.6.7 Skinnerbots

Skinnerbots (Touretzky and Saksida, 1997) use a form of reinforcement learning similar to the operant conditioning theory that has been postulated by B.F Skinner. Here we provide a shifting reinforcement function. In the initial stages we reward only simple behaviours, however after the agent has learnt these simple behaviours then the reinforcement function is shifted, to reward slightly more complex behaviours. As the agent senses this shift (from a large discrepancy in the expected and received reinforcement) then it goes into a more exploratory phase, where it tries more random actions. This learning (known as shaping) is used to train animals to perform tricks, and it has been suggested that it is how humans acquire language (O'Donohue and Ferguson, 2001:119)

This represents an interesting change in the focus of a reinforcement learning task, as the reinforcement function is normally constant, and the agent doesn't have such clearly defined exploration stages. However it has been shown that Skinnerbots can learn complex behaviour (Touretzky and Saksida, 1997)

3 Blackjack

Here we examine the first reinforcement learning task. Blackjack represents a simple reinforcement learning task as there is a very limited number of possible states that one can be in. We also removed some of the rules for Blackjack so that our eventual game is possibly too simplistic but it does make for an easy introduction as well as showing an excellent learning rate. The inspiration for this task is drawn from example 5.1, of Barto and Sutton (1998). We have chosen this task for its simplicity, as this allows us explore in detail various aspects of a reinforcement learning implementation. We compare an update rule that maintains an average, with an update rule that weights new experience. There are various constants that must be chosen for a reinforcement learning task, we also examine the effects of these constants on the performance of our blackjack agent.

3.1 Simplified Blackjack rules

In any given state there are always two possible actions − Hit and Stick. We remove the possibility of a split. A split in Blackjack occurs when one has two cards with the same value, which are then split into two separate hands. We have removed this action because we feel that splitting one’s hand is simply repeating the same learning task twice. It then gives us the benefit of storing a state simply as a single number and. Also an ace only has value one. This removes the need to consider an ace as an eleven. We feel that these changes do not alter the spirit of the game.

3.1.1 The dealer

The dealer is quite simple in that it simply hits until it gets a total of 16 or more, then it sticks. This is very similar to the Blackjack dealers found in casinos. However we do place our agent at a disadvantage in that if it does exceed 21 then it is an automatic loss, it doesn’t matter if the dealer would have exceeded 21 as well.

3.1.2 Hybrid SARSA and Monte−Carlo

We approach the problem by trying to learn the (State, Action) values. Since in a hand of Blackjack there are merely 20 legal states − all the values range from 2 to 21. Moreover from the simplifications there are only 2 possible actions. Therefore we only need to maintain a table of 40 entries.

The reinforcement function is defined as follows: if the player wins then a reward of +1 is given, if there is a draw then no reinforcement is given and if the player loses then a penalty of −1 is given.

Since there is only one reward for each hand of blackjack, this task is naturally an episodic task and the updating is done at the end of each hand, as per Monte−Carlo control. However the updating rule is drawn from SARSA. Thus the reinforcement algorithm we follow is a hybrid between SARSA (Barto and Sutton, 1998) and Monte−Carlo Control.

Eq 3.3.0 SARSA − algorithm

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$

Where:
r − is the reward received,
? − is the discounting factor.
? − is the weighting of new experience.
Q(s,a) − is the state−action value of the previous state
Q(s’,a’) − is the state−action value of the new state

Eq 3.3.0 Our Modified algorithm

$$\forall(s,a) \mid \delta(s,a) \neq 0 \Rightarrow Q(s,a) \leftarrow Q(s,a) + \alpha[r - Q(s,a)]$$

As one can see we have removed the discounting factor, instead we maintain an eligibility trace of the states encountered, and update them as a whole at the end of the game, as we shall see we still obtain the optimal policy. The eligibility trace has a discounting factor equal to one. In this way all actions are considered to contribute equally to the final reward.

3.2 Performance

We attach the code for this experiment as an appendix. Here are the results of 50 separate experiments of 1000 games, which have been averaged. The graph plotted is not average reward, but rather cumulative gains (i.e. the sum of all previous rewards). For comparative purposes we show the average performance of a player that simply plays randomly.

Figure 3.3.1 − Performance of R.L Blackjack player (Averaged over 50 trials)

Notice that from 400 trials the performance is almost approximately a straight line. (The learning can be seen from the slight curve in games 0 to 400) This indicates that after 400 games the average reward is constant and there is no more to learn. The seemingly poor performance of the player is in part due to the slight asymmetry in that if the player exceeds 21 then the dealer doesn't even play, it is counted as a loss, even though there is a fair possibility the dealer would also exceed 21 thus resulting in a draw. This slight asymmetry is suggested by Barto and Sutton (1998) and is followed here. The other contributing factor to the poor performance is that the player must take exploratory moves a certain percentage of the time, since the player seems to have completely learnt the game after 400 trials these exploratory moves simply hinder the performance.

After analysing the table obtained at the end of the experiments one can see that the optimal play is to hit until one's total reaches 16 or above, then one should stick. This means that the dealer has been using the optimal policy. What is impressive is that even playing against such a tough opponent we have still managed to obtain optimal results. However it is would interesting to know how well we would have performed if we had reduced the number of exploratory moves as the experiment progressed. This is given below in Figure 3.3.2.

Figure 3.3.2 − Reducing exploratory moves during the experiment (Averaged over 30 trials)

At the beginning both players take exploratory moves with equal probability. However for the red player we linearly decrease this probability to zero as the experiment nears completion. The increase in performance is clearly visible, and represents the true performance of the agent.

3.2.1 Different update rules

The form of our update rule (Eq 3.3.0) is at first a little strange.

Eq 3.3.0 $$Q(s,a) \leftarrow Q(s,a) + \alpha[r - Q(s,a)]$$

One of the reasons for this update rule is the assumption that the value of the State−Action pair is dependent on the policy. This implies that we should weight new information more heavily than old information. However we felt that the value of Blackjack states doesn’t vary too much with the policy followed so we investigated another updating method. This is the much more intuitive updating rule of maintaining an average, as suggested by Monte−Carlo control (Barto and Sutton, 1998). Notice that this does make the policy a lot more inflexible as if something in the environment does change then it will take a long time before the policy is changed. However this might also help, e.g. if we decide to stick on 6 and the dealer exceeds 21 then we win. The normal updating rule will then update the state−action value a lot, even though this is not a good action. An average−based updating rule will be a lot more resistant to such statistical anomalies.

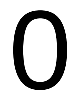

Figure 3.3.3 Comparing SARSA and Monte−Carlo update methods

As one can see the SARSA updating method is superior to maintaining an average, although not by much, which we found counter−intuitive. However this can be taken to show that the value of a state−action is dependent on the policy followed. This does make sense, for example hitting when one has a total of 3, will have a value greater for an optimal policy than for a random policy. Thus as the true values of the states change, we should weight new experience more heavily than old experience, this is what happens in the SARSA updating rule.

For the Monte−Carlo rule we assume that the value of a state−action pair is independent of the policy being followed. Under this assumption the estimate with the least variance is the simple average, of all previous experience. Thus one can see that the assumption made by Monte−Carlo control is an invalid one, since it is outperformed by the SARSA updating rule.

3.2.2 Initialising the Q−table

Another issue of concern is whether or not to initialise the whole table to zeros, or small random entries. One concern is that by initialising them to zeros then the first action in the table would be taken for every state in the beginning of the experiment, one could overcome this by choosing an ?−soft policy (i.e. choosing randomly among state−action pairs that differ by no more than ?). Or we could simply initialise the table randomly (with small numbers to ensure the policy is still sensitive to new experience). We choose to compare randomly initialised table performance with the zero−initialised table performance.

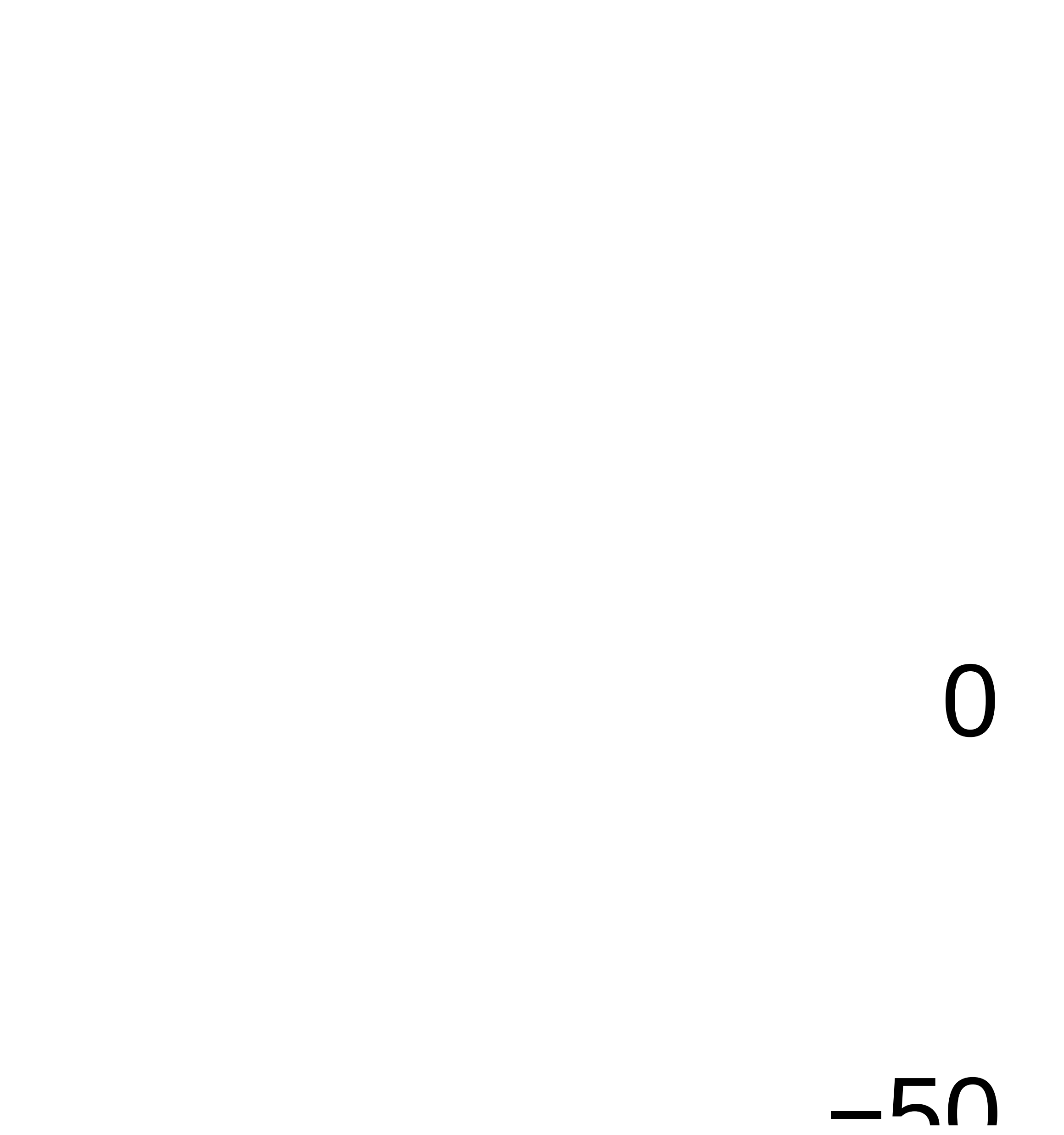

Figure 3.3.4 − Different Initialisations of the Q−table (Averaged over 100 trials)

As one can plainly see there is no significant difference on the rate of learning, nor on the final performance of the agents.

3.2.3 Different values of ?

In (Eq 3.3.0) the value of ? is chosen arbitrarily and it does appear to have a pivotal role in updating the Q−values. How important is the value of ?? Can a badly chosen value of ?, slow the learning rate tremendously? To find out we varied ? in the range [0..1]. When ? = 0 then new experience has no bearing on the Q−table, and when ? = 1 then the Q−table is based only on the last experience. Obviously one would expect the best performance would be somewhere in the middle.

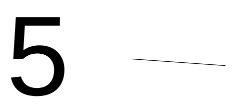

Figure 3.3.5 − Surface showing performance with multiple values of ? (Averaged over 30 trials)

Here the surface shows how the games progress. (Red shows initial performance, blue showing final performance) What is most clear from this graph is that the worst performing value for ? is ? = 0. This is to be expected, for no learning has occurred. What is encouraging though is that the performance for a large interval of ? [0.10.7] the performance, while not constant is almost uniformly good. Moreover the learning rate seems to also have little variance for this range of ?. To illustrate this more clearly we show the final performance in Figure 3.3.6.

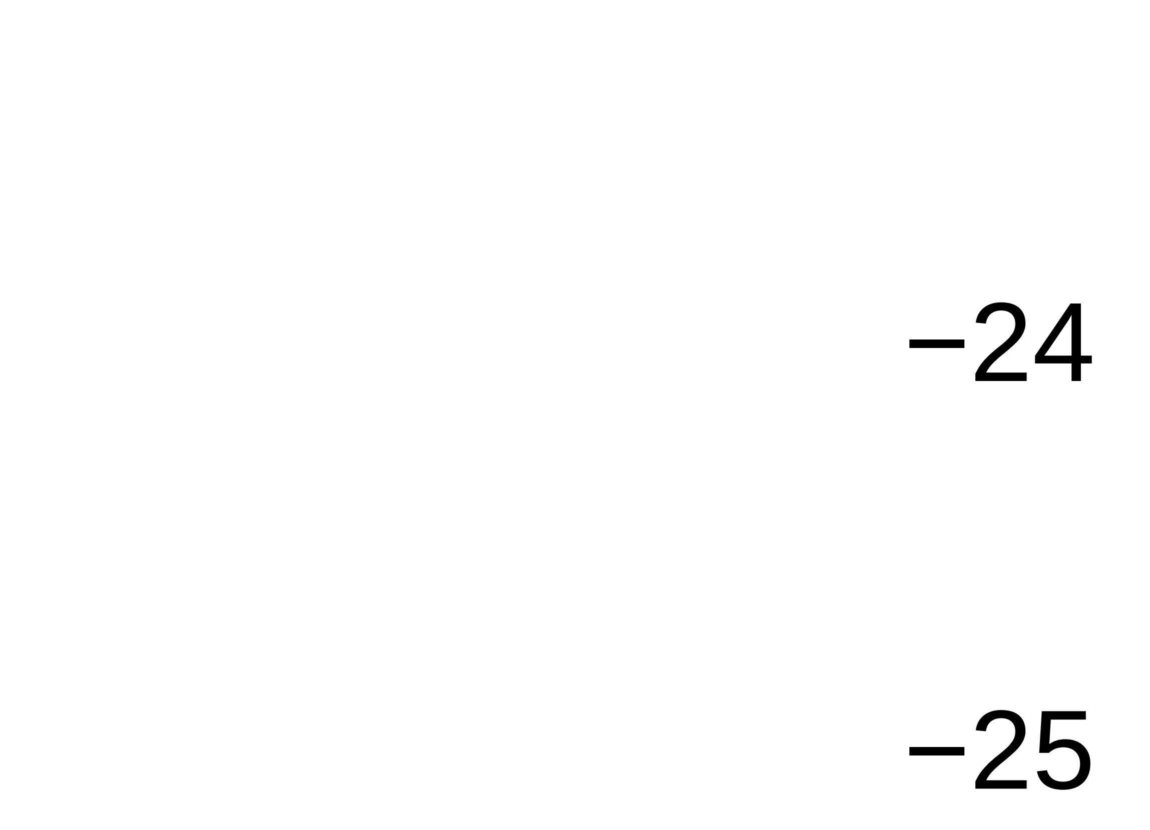

Figure 3.3.6 − Final performance of player for multiple ? values (Averaged over 30 trials)

3.3 Conclusions

In this simplistic environment we have tested the effects of various constants, update rules and initialization procedures. From the data we have gathered we can see that while some optimisations are possible, these do not contribute significantly to the overall performance of the player. This is good, since it means that the performance is robust (consistently good, over most choices of parameters). Of the two update rules that we examined, the SARSA−like update consistently performed better than maintaining an average (although this increase was marginal). As a result we have chosen it rather than maintaining an average, since to compute the average we also have to maintain a table counting how many times each state−action is visited.

4 Maze World

In this task an agent is placed in an approximately 20×20 grid. There are several obstacles in this environment and also there are now more actions − {North, South, East and West}. This means that not only is the size of the task is considerably larger, but also that rewards are received only after many iterations. Thus the problem of credit assignment is also introduced.

We have chosen this task as it represents a natural increase in the complexity of the reinforcement learning required. Not only are the optimal solutions many times longer than in the blackjack task (which means we have to implement an eligibility trace correctly) but we are also faced with more available actions at each step.

Once again there are similar questions that we can ask, about the parameters involved

4.1 Maze Generation

To generate the mazes we use the following algorithm. We initialise the maze to a grid consisting of alternating walls and empty cells. We then randomly pick a wall that separates two empty cells that aren't connected. This wall is removed and the two empty cells are updated to reflect that they are now connected. We repeat this procedure until every empty cell is connected to every other empty cell. This generates what is known as a labyrinth. A labyrinth has as its defining property that between any two points there is only one path joining those two points. Obviously this implies that there are no cycles in a labyrinth.

Figure 4.4.1 − The initial labyrinth

Figure 4.4.2 − The maze that has to be learnt

However if there are no cycles in a labyrinth then learning a labyrinth is a very easy task, since there is only one path that can be learnt. To increase the difficulty of the task we then proceed to remove a percentage of the blocks. After some experimentation we settled on removing a quarter of the blocks. Thus we have a maze that has multiple paths.

4.2 The Reinforcement Learning Framework

To fully define the task we still need to decide on a goal state. For this task we simply picked a random eligible state. Notice that we need to perform a further check to see which states are eligible. There is a slight problem when we randomly remove blocks since we can remove a block that is bordered on all four sides by other blocks. These empty blocks are defined as not eligible because if we try to define a goal or starting position in one of these blocks then the task is impossible. All occupied blocks are also defined as not eligible. Any other block is eligible.

The reinforcement function is defined as 0 for all transitions that do not lead to a goal state, and a value of one is returned when we transition to a goal state. Notice here the flexibility of this framework. We are able to define multiple goals if we wish, without creating any difficulties. Also in this case we have decided that a move of north transports the agent one block up, while south transports the agent one block down etc. This need not be the case in general; we may also define special blocks where a move of north transports the agent to any fixed position. (Barto and Sutton, 1998) cover various examples of this. Here we can start to see the trade−off's quantified: while Reinforcement Learning will not be anywhere as efficient as an A* search, or even a simple breadth−first search. Reinforcement Learning can however deal with many more arbitrary restrictions without having to fundamentally change any part of the algorithm. This is because part of the agent's energy is directed towards discovering the environment it is in. It is the position of this author that method's of this type are the most likely to be used in creating artificial life. We feel that only these methods possess the flexibility required to deal with the outside world.

However this flexibility comes at a price of not just more computation required, but also we cannot guarantee the optimality of the solutions we derive. This is in part due to the fact that the convergence proofs require that each state and action be tested an infinite number of times, while we gradually reduce the learning rate. The Maze world provides the best evidence of non−optimal solutions. In some mazes even after a significant amount of training the paths derived from the Q−values have some unnecessary twists and turns. However, in general, these non−optimal choices haven’t been experienced often, which means that the policy we can extract will be optimal for the most frequently visited states.

4.2.1 The Q−table

In our implementation we use a large table to store the Q−values for each (State, Action) pair. We give moves that would result in a collision a value of −1. This identified them as illegal, and they are not selected. Initially since nothing is known about the Q−values we set the legal entries to a pessimistic value of zero. Here if we use an optimistic value then when the agent encounters a path that has already been experienced the agent ends up trying to find a totally different path that doesn’t follow the known route. However in quite a few situations the only path is the known one. Thus the agent is effectively trapped as it is actively ignoring the only solution!

In a SARSA implementation this would not have been a problem since, the updating occurs between interactions with the environment, as opposed to once, only at the end of the episode. This means that the agent would behave nicely, actively exploring the unknown paths and reducing their estimated values all the time until the best action is the previously found exit path.

To begin the learning process we would begin by starting the agent off in a random position. Note that the only input provided is the position in (x, y) coordinates, but that this state information does have the Markov property, since we are not interested in how we got to that point, all we need to know is which point we are at. With the (x, y) coordinates we can reference the Q−table and find which action has the maximum Q−value. To maintain exploration we also select a random action a fixed percentage of the time. We do however reduce this percentage linearly so that the agent does also experience any possible side−effects of following the policy exactly.

4.2.2 Eligibility Traces

Another important aspect in learning the Maze World task, is correctly applying Temporal Difference Learning. To this end we maintain an eligibility trace, the trace keeps a record of the past history so that when one reaches the goal state not just a single state is rewarded but the whole history of past states are rewarded. Notice that since we only reinforce at the end of the experiment, an eligibility trace is essential, otherwise, only the state that transported the agent to the goal would get reinforcement. Moreover we need to decay the trace away, or else every action will appear as beneficial as the last action. This means that the agent would not be able to distinguish between following the solution forwards or backwards!

The eligibility traces must have a discounting factor less than one and not too close to zero. If the discounting factor is made to be one then the agent makes no distinction between states that lead him towards the goal and states that lead him away from the goal, since after a while he will eventually reach the goal, and all visited Q−entries will appear to have contributed the same amount to reaching the goal. On the other hand if ? is close to zero then $?^n$ will round off to zero for sufficiently large n. This means that any solutions that require longer than n steps will be ignored, which is not what we want.

Another point to mention is that the traces are replacing traces and not accumulating traces, this means that when a previously visited eligibility trace is encountered then it is reset to one, rather than simply adding one. The benefits of this can be seen in the following example: if an agent moving randomly as it would in the beginning then it might move between two blocks repeatedly, thus an accumulating trace could then have values well above one even though those actions are not helping in the solution of the Maze, when the agent finally does solve the maze then those actions are unduly reinforced. This perpetuates the problem as the next time the trial is run then the agent might simply choose to alternate between those two states, which would again give them undue reinforcement when the maze is finally solved (by taking a random action).

4.3 Example of Maze−Learning

Here we present a detailed description of the progress of a single trial.

Figure 4.4.3 − Initial Policy

Figure 4.4.4 − Policy after single run through Maze

Figure 4.4.5 − Policy after 20 runs

Figure 4.4.6 − Policy after 2000 runs

In each of these figures the arrows indicate which direction our agent will choose if they are placed in that block, viewed collectively the arrows represent the agent’s policy. If there are two or more arrows in a block it indicates uncertainty and if the state is encountered one of those actions will be chosen at random. In Figure 3.3.1 we can see that the Policy starts off having no idea as what an optimal policy should look like. When the agent enters the maze it has no idea of what constitutes a good move, thus it simply moves randomly until it finds the exit. After the first run though it does have some idea of how to reach the exit, however if one examines Figure 4.4.4 then one sees that some of the paths are tortuous and convoluted. This is to be expected since the movement was random. Thus we don’t pay too much attention to the information we have attained, we still choose a random move 75% of the time. In time however the agent’s random moves build up a fairly accurate picture of the maze, and the agent slowly reduces the chance of a random move. In this way the performance increases and the values of the Q−table approach their true optimal value.

Since the maze is consistent − there are no blocks that behave randomly, or blocks with unknown behaviour it is easy enough to compute what would constitute an optimal policy, shown in Figure 4.4.7. The number of times that the policy of our agent differs from the optimal policy is an intuitive statistic of performance. In Figure 4.4.6 the performance of our agent stands at 7. Moreover in these 7 cases where there is such a difference are normally far from the goal state and do not contribute significantly to what would be a fairly efficient run through the Maze.

Figure 4.4.7 − Optimal Policy

100

Figure 4.4.8 − Learning curve for runs through the maze (Average over 30 trials)

From this figure we can see that the bulk of the learning occurs before 100 runs, after that the learning is almost negligible, however it does still occur, as one can see from the fact that after 2000 trials our example had a much better performance of 7.

4.4 Different values of ?

Once again we might ask the same questions about ? − How sensitive is the performance of our agent dependent on our choice of ?? Can we fine−tune ? so that the learning rate is optimal? Figure 4.4.9 gives the results. From this we can see that as long as we discard ? = 0 (which implies that no learning takes place) then we appear to get little variation in the performance statistic. On closer inspection there is a slight improvement for ?−values close to 1, but it doesn’t appear considerable.

160

Figure 4.4.9 − Performance after 100 runs for different values of Alpha (averaged over 30 trials)

4.5 Another updating rule

After some thought over the required task we thought that we might be able to derive a better updating rule for this task. Here we attempt to remember the shortest path that we have found, updating a path only when a new shortest path has been found. This can be done quite simply if one initialises the Q−table to zero. Then whenever the end of an episode is reached we update the Q−table as follows:

Eq 4.4.0 Alternative updating method

$$Q \leftarrow \max(Q, \delta)$$

(For all legal state−action pairs)

In this way it is impossible to forget a good solution, and it only updates with better solutions. This is a very inflexible policy and if the environment were to change midway through an experiment this update rule would not be able to adapt at all. That said though it would appear as if this is tailor−made for a maze−learning agent. We plot the performance graphs for both update rules in Figure 4.4.10.

Figure 4.4.10 Different performances for different updating rules (Averaged over 30 trials)

Our intuition is correct and we can see that our ’max’ update rule does perform better than the SARSA−like update rule. It should still be pointed out that this rule will lead to very poor performance in other tasks, and as such the usefulness of this rule is very limited.

4.6 Conclusions

It does appear that reinforcement learning is able to perform very well, learning most of the optimal actions in a short space of time; moreover this performance is not dependent on the choice of ?. We have also shown that a tailor−made update rule can be constructed whose performance is better than the hybrid updating rule that we use. However this update rule is of limited applicability and so we won’t consider it in the following chapters, where we now consider the issues associated with multi−agent systems.

5 The theory of cooperation

In this chapter we explore the second theme of this paper: cooperation. We explore the work done in game theory and state some of the conditions necessary for cooperation to be a viable strategy. Thereafter we examine critically examine the tasks we are to present in chapters The Traffic Simulation and 7. Is it plausible to expect cooperation to be a strategy which can be learnt?

5.1 Introduction

Robert Axelrod poses the following question in *’The evolution of cooperation’*,

> "In situations where each individual has an incentive to be selfish, how can cooperation ever develop?" (Axelrod, 1984: 3),

This is of great relevance to our work here. If we examine civilisation we can see numerous examples of people cooperating for the greater good. But is our cooperation a sign of people simply submitting to a greater authority such as government?

If we examine nature we are also able to identify numerous examples of cooperation. A very nimble plover is able to fly into a crocodile’s mouth, and peck food out from between the crocodiles teeth. Here the crocodile is almost offered a tasty meal, and yet refuses to eat. Moreover if the crocodile did choose to commit the traitorous act then other plovers wouldn’t be aware of this traitorous action, and the crocodile would still be covered in the dental plan. Surely that the crocodile does not eat is an example of cooperation. But if cooperation can evolve then it is necessary to know under what conditions it would evolve. This is especially so if we are interested in creating an environment suitable for cooperation to be learnt.

Most situations in life can be modelled in the form of a game (Sigmund, 1993). The trickiest game that involves cooperation is called the Prisoner’s dilemma. It represents the trickiest form of cooperation since both parties are tempted to not cooperate, but rather to defect on each other. The following standard description that accompanies this game accounts for this game’s title:

You and a friend are have committed a crime, both of you have been arrested, and are not allowed to communicate with each other. The police have some evidence, but not enough to convict both of you for the crime. If both of you keep quiet then the police will only be able to send both of you to jail for a year only. If one keeps quiet and the other tells all then for keeping quiet they will go to jail for 3 years, and the other will get off scot−free. If however you both confess about each other then you will both go to jail for 2 years (Poundstone, 1992:117−118).

If one examines this rationally then one could construct the following argument:

- I cannot know what the other person is going to do.
- If he is going to ’rat’ on me then I should rat on him. (Jail time reduced from 3 to 2 years)
- If he is gong to keep quiet then I should rat on him. (Jail time reduced from 1 year to nothing)

Therefore I should rat on him.

However if your partner in crime is also using this argument then he will also rat on you, with the net result that both of you end up spending 8 years in jail. In this way rationality doesn’t seem to work very well.

This problem can also be put in a matrix form as follows:

	Player 1 − Cooperates	Defects
Player 2 − Cooperates	(1,1)	(0,3)
Defects	(3,0)	(2,2)

Here cooperation means keeping quiet, and defection means ratting on the other person. Now the problem of cooperation, is starting to seem a lot harder, since one is sorely tempted by the temptation payoff (doing no time in jail), yet for this to happen someone else must suffer.

Obviously, if you know that you will never see your partner again, then on an intuitive sense the rational argument starts to make sense. However what if there is a good chance that you will interact again? This repeated interaction changes the nature of the prisoner’s dilemma. Axelrod refers to this as the shadow of the future, and to examine its effects he hosts several computer tournaments.

5.2 Computer tournaments

Axelrod let anyone to enter a possible strategy as a solution to this iterated prisoner's dilemma. For a strategy to count as valid it must base its decisions only on the past actions that have occurred. It is not allowed to enquire as to whether or not the other strategy is going to cooperate or defect. The strategies were then paired off in a round−robin style tournament. Strategies were also paired against themselves, as an extra round.

Surprisingly the entry that won the first tournament was also the shortest entry. It was called TIT FOR TAT. It chooses to cooperate on the first move, and then on all subsequent moves it chooses to do whatever the opponent has done in the previous move. The results of the first tournament were published along with some detailed analysis of TIT FOR TAT. The second tournament was then held with entrants specifically attempting to beat TIT FOR TAT. The format was the same, and there were more entries. Again TIT FOR TAT won, although in both tournaments there are possible strategies that could have won, but were not entered. However none of the strategies are as robust in their performance as TIT FOR TAT.

5.3 Analysis of Competitions

Why does TIT FOR TAT do so well? Axelrod (1984) examines this in detail and derives three basic properties that any strategy for the iterated prisoner's dilemma should possess.

5.3.1 Niceness

Axelrod finds that almost all the top−placed rules are nice. They are never the first to defect. This property helps their score when they meet each other. Both of the rules cooperate with each other and since neither will defect first, eliminating costly retaliatory defections, they both get a high score.

5.3.2 Forgiveness

Of the nice strategies the strategies that perform the worst are also the least forgiving. Generally these unforgiving strategies cannot cope with the occasional defection. Instead they start an unending pattern of mutual defection.

5.3.3 Responsiveness

Again the rules that do well, generally assume that the other player is interested in cooperation. So they make an effort to extract it. If it fails then they haven’t lost too much, however if they do manage to establish cooperation they will have gained a lot.

5.3.4 Conclusion

All of these characteristics are positive; they seem not to belong to a game where the highest payoff involves defection. However this illustrates that for long−term performance one must seek a solution that is equitable for both sides concerned, in spite of the obvious temptations posed by defecting.

In fact all these positive characteristics show that future interactions have a significant effect on what we should do at present. But this sounds very similar to temporal difference learning, where future rewards are also attributed to previous actions. This would seem to suggest that temporal difference learning can learn to cooperate, since it would take future interactions into account.

5.4 Applicability in the real world

As far as an abstract game theory, this seems fair enough, but how applicable is it in the real world? A prime example of a real scenario fitting the prisoner’s dilemma occurs in OPEC (or any other similar cartel). Here the oil−producing countries collaborate on setting a price to sell oil. If one of the countries though, were to break this agreement and sell the oil at a slightly lower price then its profits would increase dramatically as the whole world would rather purchase the cheaper oil. The country would make a lot more money since, it does not have to compete against other countries; they are no longer competitive. Thus there is a large incentive to undercut each other and yet they seldom do this.

Another example occurs in the trench warfare of World War I (Axelrod, 1984). Here the soldiers would face each other for weeks on end, with each side being able to incur heavy casualties. Yet each side would rather show off this ability, rather than exercising it. How did such cooperation between antagonists come into being? The answer is that this environment is correctly modelled by the prisoner’s dilemma. Cooperation here involves not shooting to kill. If a side chooses to cooperate then there is a great temptation for the other side to defect and try to win some land. However if one side tries this then the other side will return the lethal fire soon after, with many people dying. This is not ideal for either side, so both sides prefer not to attack too accurately for fear of accurate reprisals. Axelrod (1984) details how difficult it was to get the troops to fight properly, since both sides tended to favour cooperative behaviour.

5.5 Assumptions required

While we have detailed the many examples occurring in real life that fit the iterated prisoner’s dilemma, and shown that TIT FOR TAT is a successful strategy in such a game, we still need to state the implicit assumptions that are made in modelling this game.

Central to the success of TIT FOR TAT is the ability to remember the previous action dealt to it by its opponent. This has in it two implicit assumptions. The first is that the opponent’s moves are easily interpreted as a defection or as cooperation. In many reinforcement learning tasks this requirement is not met, for example in chapter The Traffic Simulation we explore the task of learning to cross an intersection. If a collision occurs then it is unclear who has defected or even if both have defected. How is a reinforcement learning system then to apportion blame, or credit?

The second implicit assumption is that the opponent is identifiable. As obvious as this seems, it is not trivial for the implementation of a reinforcement learning system. If a multi−agent system is to remember which agents are being cooperative towards which other agents then an extra degree of complexity is introduced. Moreover since we would need to maintain this cooperation as a state variable it implies we would need to at least double the size of our Q−table. However this is assuming that only one other agent is encountered in any possible state, if it is possible to encounter n agents, then the Q−table would increase in size by 2^n.

If we try to remove the identifiability requirement, then any strategy should defect since any cooperation can be taken advantage of with impunity. This problem is known as the tragedy of the commons. In medieval Europe, many towns had a communal grazing area. One could choose to graze one’s flock on one’s own land, or one could choose to graze the communal land. Since it was a big advantage for a single person to use the communal land (or defect) and only a small disadvantage to the community, everyone would tend to use the communal land as much as possible. As no−one monitored the communal land it was unlikely that one would be identified as a defector. This resulted in the ruin of the communal grazing area, and is a problem faced today with over−fishing in international waters. Only through the strict monitoring of these common areas, can cooperation once again be established.

In this way we can’t remove the identifiability requirement, however many problems it may introduce for a multi−agent system, since cooperation wouldn’t be learnt. Obviously one could enforce that the agents must cooperate, but this paper is interested with systems where the cooperation is learnt.

5.6 Chicken

Fortunately if we examine the tasks that we wish to solve we find out that the situations are more accurately modelled as a game of chicken (Sigmund, 1993). In this game the payoff is slightly different, and as we shall see this makes a large difference.

The name of this game is derived from a reckless game where two antagonists drive towards each other, with the first to swerve being labelled the ’chicken’, and they have to pay some bet. If neither swerves then an accident occurs and both are worse off, since they have to pay for repairs (Poundstone, 1992:197). In fact this example is very apt since in both chapters The Traffic Simulation and 7, we are interested in collision avoidance.

The payoff matrix is as follows (assuming a bet of R10):

	Swerve	Drive straight
Swerve	(0,0)	(10,−10)
Drive Straight	(−10,10)	(−100,−100)

This is different from the payoff table in Introduction, since driving straight is not the best choice in both cases. What now emerges is that we are trying to predict the behaviour of our opponent and do the opposite. While in the prisoner's dilemma if we know the opponent is going to defect then we defect as well.

Real world scenarios of chicken occur all the time. One of the best examples to give is the Cuban Missile crisis. Here both America and the Soviet Union needed to appear to each other as being willing to go to war if necessary, even though that was the worst outcome possible. Whoever seemed most willing to go to war would then win and the other would back down (or chicken out). Eliciting cooperation is then not necessarily as straight−forward as one might think.

For us this slight change means that instead of trying to elicit cooperation, we must obtain some form of coordination, or protocol, which can be learnt, or communicated. Obeying a protocol is still a form of cooperation, yet it doesn't have the associated difficulties of the prisoner's dilemma, but can it be implemented in a reinforcement learning system?

6 The Traffic Simulation

In this chapter we explore the possibility of learning cooperation between independent reinforcement learning systems. We explore whether reinforcement learning always leads to the optimal protocol, for such coordination. The task that we have chosen for this is a traffic simulation. However it is significantly different from Thomas Thorpe (1997).

Here we are not interested in optimising the traffic lights, but rather we are interested in learning a global policy of negotiating intersections. For us this problem may not be entirely obvious, but at one stage there must have been no rules of the road. Then as congestion became a problem (as well as increased speed) a clearly defined set of rules (or protocol) was needed. While in real life these rules probably evolved as was needed and then declared as law, it is interesting to see if they would arise solely through reinforcement learning.

6.1 Description

We borrow from the maze task and use the same algorithms to produce a city grid. Moreover we repeatedly train a few cars, with different goals, to navigate through the city to their respective goals. This leaves us with a traffic simulation, namely several cars, each travelling through the city, thus creating the risk of a collision. This collision avoidance is the task to be learnt.

As the cars arrive they are then given a picture of the intersection. This picture includes how many cars are at the intersection, as well as a traffic light. This traffic light is used as an arbitrator, and all the cars receive the same signal, it simply alternates between one or zero in the simulation, which signifies red or green.

6.2 Analysis of the problem

The definition of a collision is simplified slightly to the following: a collision does not occur if both cars proceed straight at the intersection, nor does it occur when both cars turn at the intersection, it only occurs when a car is turning and another is proceeding straight. This definition implies that if four cars proceed straight in an intersection then there is no collision, however if one car does turn then all cars are involved in a collision. This definition allows a great deal of efficiency to be potentially achieved if the coordination is properly done. It also simplifies the required collision detection code.

With the simplifications of the collisions, as well as the fact that the traffic light is the same for everyone, it means that it is quite easy to derive an optimal policy. If the traffic light is red then cars that are turning can proceed, if the traffic light is green then the cars that are driving straight can proceed. Moreover a car that is at an empty intersection can proceed. However our reinforcement learning method does not achieve this optimal policy and it is instructive to see why. Notice also that this policy does not depend on any of the positions of the other cars, this extra information is given effectively to complicate the protocol, and the system needs to learn that in all states we only need to pay attention to the traffic light.

We consider events to be independent of each other. This isn't quite true since if a car decides to wait at a busy intersection then the next step in simulation time will see the car at the same busy intersection. As we shall later see this decision does introduce several problems.

Initially we test to see whether or not this problem is satisfiable. We check this by allowing cars to share the same Q−table. In effect then all the cars are trying to find a policy that is consistent (i.e. if all cars adopt this policy then there are no collisions). We then extend this naturally by allowing different Q−tables, for different cars. Another interesting experiment is to test whether or not the traffic light helps the coordination of the cars, or if they can manage by themselves.

Another factor to be explored is that of exploration. Is it necessary? If a protocol is being sought then surely exploration simply disrupts any potential solutions. If there is an emerging protocol, and a car takes an exploratory move, thus creating a collision. It will make the potential protocol appear as unviable since the collision will lead to a high penalty awarded for all the cars following the protocol.

6.2.1 Is it achievable?

A concern is that there is no guarantee that a group of agents, each presented with a view of each other, are able to arbitrate fairly. If this is the case then the cars that are receiving the short end of the stick will have more incentive to disobey this protocol and will ruin the protocol for everyone concerned.

Figure 6.6.1 Number of collisions. (Averaged over 30 trials)

Figure 6.6.2 Number of Cars Moving. (Averaged over 30 trials)

Figure 6.6.3 Congestion (Total number of cars at intersections)

At first view Figure 6.6.1 appears encouraging; the cars are definitely reducing the number of collisions that are occurring. Nevertheless Figure 6.6.2 and Figure 6.6.3 show that the number of cars waiting at an intersection increases while simultaneously the number of cars moving decreases. The policy thus doesn't appear to be maximally efficient. This is however due to the experiment parameters. For the city that the experiment was run there were only 54 free blocks and 120 cars. This means that the city was already in an extremely congested state. Considering this it seems reasonable that the policy is cautious, in entering an intersection.

6.2.2 Traffic Lights

The traffic light is represented as a binary red or green. The state of the traffic light is the same for all cars concerned. In this way one would expect it to play a significant role in coordination.

Figure 6.6.4 Number of Collisions (without a traffic light)

Figure 6.6.5 Number of Cars moving (without a traffic light)

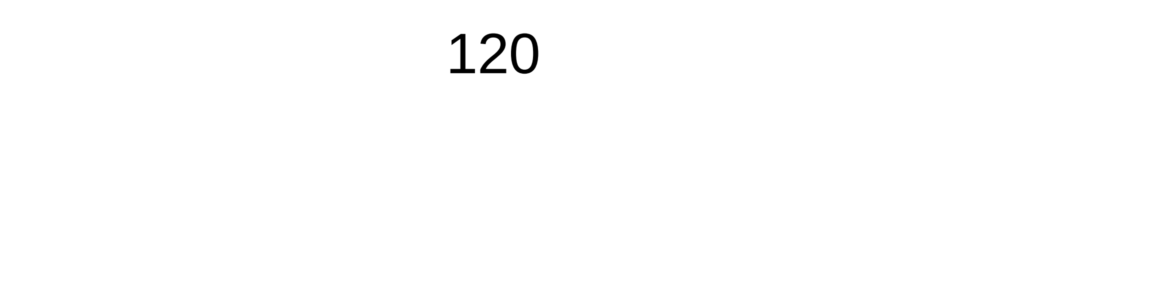

Figure 6.6.6 Congestion (without a traffic light)

As is clear from the graphs the reinforcement learning system quickly observes that the best way to avoid a collision is not to proceed at all. Notice that the system could have reached the opposite conclusion if we distributed the rewards over previous decisions as well, and gave a high enough weighting factor. The system would then learn that stopping at an intersection leads to a continual penalty, so it is probably best to proceed, no matter what the congestion is like.

6.2.3 Exploration

The theme of exploration, which is central to reinforcement learning, might disrupt any emerging protocol enough to make it look like an unsuitable alternative. In a simple task as this (i.e. with only two possible actions) is exploration necessary, or do protocols emerge quicker without it?

Figure 6.6.7 Collisions

Figure 6.6.8 Cars Moving

100 10

Figure 6.6.9 Congestion **Figure 6.6.10 Reinforcement given**

The increased variation from the exploratory steps is evident in the much ’fuzzier’ line. Initially an exploratory move was taken with probability 0.2, we linearly decreased this until two−thirds of the way through the experiment, when the probability reached zero. Once there are no more exploratory moves a very interesting change occurs in the behaviour of the system.

While we are exploring, the traffic appears to be flowing well, with a slightly higher collision rate than in other experiments. However as we stop exploring, we immediately become more cautious, and the collision rate drops dramatically, however the congestion also increases as fewer cars are proceeding as before. Overall the total reinforcement given per simulation also decreases.

This startling behaviour can be accounted for, if we consider what happens when a collision does occur. This would change the value of the state considerably, possibly making it unattractive when compared to waiting at the intersection. If this happens, and we never make exploratory moves then we will never enter an intersection from that state, instead we accept the small penalty for waiting. This leads us to conclude that exploration is necessary for the learning of a protocol, since if it doesn’t occur the protocol will probably be sub−optimal.

6.2.4 Conclusions

It does appear that the problem is suitable for a single−agent system. However it still does not achieve the levels of performance we might have hoped for. This appears to be symptomatic of the system learning against itself. When a collision occurs the system must arbitrate between the two cars, however it is not able to do this, and penalises both at the same level, thus if the same scenario is encountered both cars tend to still display the same behaviour. In this way learning an efficient, consistent policy is difficult. Is it easier in a multi−agent system?

6.3 Multi−agent systems

The experiments we have been looking at are concerned with the viability of this experiment. Since we can say that it does appear to be viable, we can now focus on multi−agent systems. For us this entails maintaining separate Q−tables for different cars. Due to its computational cost we restrict ourselves to experiments with four independent Q−tables, but we feel certain that the results we obtain extend to any number.

Figure 6.6.11 Collisions (Separate Q−tables)

Figure 6.6.12 Cars moving (Separate Q−tables)

Figure 6.6.13 Congestion (separate Q−tables)

There is visibly less variation in the figures if one compares them to figures in section Is it achievable?. This is due to the fact that the reinforcement is being distributed over a few systems. Surprisingly enough there is no appreciable difference in the learning rate. We initially felt that the learning rate should be slower as a consequence of having four times the entries in Q−tables to update, however this was not the case.

6.4 Conclusions

In hind−sight we can see that our decision to consider the events as independent appears costly, it has hindered learning significantly since the system is not able to accurately include the cost of waiting.

However despite the fact that our system has not been well designed we are still able to see that having multiple agents interact is a feasible idea. Yet there are still difficulties in that the systems were not able to learn the optimal policy, instead they learnt a suboptimal one. This sub−optimality is a direct result of the reinforcement learning system being unable to differentiate between penalties from the environment and penalties from other reinforcement learning agents. It is therefore unable to tell if the policy is performing badly because it is ignoring environmental effects, or if the policy is performing badly because other agents are still learning the policy. This does suggest that more research is necessary as it does not seem that current single agent reinforcement learning algorithms converge to optimal policies.

7 Cellular Simulation

This is the final reinforcement learning task we present. Our central goal is to explore the complexities of extending a reinforcement learning task from discrete states to continuous states. This prevents us from using a Q−table, and instead we now use standard back−propagation neural networks. We initially also wished to explore the possibilities of cooperation being learnt, in a simulated biological system. Robert Axelrod devotes a chapter to the 'cooperation without friendship or foresight' that occurs in biological systems (Axelrod, 1984:88).

This simulation is inspired by Conway's game of life (Sigmund, 1993). In that simulation there are only two states − occupied and empty. The simulation is discrete and no movement of the cells is possible, instead they reproduce once a certain density is achieved. If the density is too high then the cells die from overcrowding. If one watches a simulation of the game of life it displays seemingly random, and yet life−like behaviour with population growth and shrinkage for a long time before the system settles down.

Figure 7.7.1Example environment

In our simulation, there are several types of objects in the environment. First and foremost is the animal, it is the only type that is capable of movement. Necessary to the animal's existence are food and water. There are also obstacles that need to be avoided, as they do not provide sustenance and incur large penalties if hit.

In Figure 7.7.1, the blue circles represent water; the green represent food and the purple represent obstacles. There are two animal species, in orange and red, and their sight inputs are shown as lines.

If an animal does not eat for a long time it does not die as in an evolutionary experiment. Instead it is punished for not eating. In a similar way if an animal collides into an obstacle or another animal then it is also punished. Collisions with food and water are reinforced with a large reward. Balch has researched various reinforcement functions and found that a local reinforcement function (rewarding an individual's behaviour, rather than a species' behaviour) leads to more homogenous agents (i.e. they use similar strategies) and, for a foraging task such as this one, better performance (Balch, 1999)

7.1 A successful strategy

What does solving this task require the animals to learn? A reasonable policy would have an efficient searching pattern, in which the animals might travel in a large circle looking for food. It also requires that once food is sighted then it is efficiently reached and eaten. As we have mentioned there are obstacles in this world which need to be avoided. We expect that the most difficult part of the policy would be to learn collision avoidance of each other. This collision avoidance is similar to the traffic simulation covered in chapter The Traffic Simulation.

7.2 Using the Neural network

What is a neural network? A neural network is essentially a collection of functions and a table of weights. It matches input patterns to output patterns by transforming the inputs. It achieves this through multiplying the inputs with the weights and applying the functions to this product. This is known as a layer of the network. A network can have several layers, to give it extra flexibility in matching the outputs.

Initially the weights are initialised randomly, but after we have some input−output pairs, we are able to adjust the weights so that the network is able to closely reproduce the outputs if we give it the inputs. We do this by calculating the error at the outputs and propagating it backwards through the different layers. This gives rise to the term back−propagation network.
This type of learning is called supervised learning as there is a teacher, which provides the model answers. However in reinforcement learning we only have a single number that indicates our performance, we are not given any model answers.

So a neural network is very useful for supervised machine learning. However this doesn't match the reinforcement learning framework. In reinforcement learning we don't have any model answers, we are supposed to discover them ourselves.

A rather straightforward way to use a neural network in a reinforcement learning problem is to predict the Q−table. If we have the state and the action we can try to predict the reinforcement we will receive. The reinforcement is given to us as a model answer which we can use to train the network. Of course this approach is still rather myopic, we should be trying to maximise long−term rewards, not short−term. To get around this we perform batch learning. We let the creatures experience the environment for multiple steps, long enough to build up a significant amount of experience. Then we distribute the rewards back into the past. In this way, we can create a long−term planning agent. Once the rewards have been distributed backwards we are then able to train the neural network. After a long simulation we should be able to predict the reinforcement for a state−action pair with a high degree of accuracy.

7.2.1 Obtaining the action

However we are still avoiding a crucial topic; how do we use this neural network to predict which action to take? We can do this through a numerical technique known as gradient descent (Burden and Faires, 1997:614). Starting at a random point in the action space we can evaluate the estimated reinforcement for the given state and random action. Since the functions used in a back−propagation neural network are required to be differentiable we can also work out the gradient for the random action. This gradient tells us the direction in which the estimated action would increase the most.

Figure 7.7.2 Surface for predicted reinforcement

Figure 7.7.3 Path followed by gradient descent

This algorithm is analogous to climbing a hill. We start in a random place and try to climb as fast as we can. However we do not always reach the global maximum, if we had started in an unusual spot then we might climb towards some local maximum, but not the global maximum. In Figure 7.7.2 we can see the surface has a definite maximum on the left−hand side, however if we were to have started on the right hand side, then we would have been lead towards the right hand corner which has a slight local maximum, however this action appears far from optimal according to the network.

There are other ways of using neural networks to learn an environment, and predict the best action to take; some of these methods require new architectures such as radial basis functions (Barto and Sutton, 1998:208). Another method includes solving the Hamilton−Jacobi−Bellman equation involving partial derivatives (Nikovski, 1998). One can also use multiple neural networks; one to build a model of the environment, and one to try and learn the reinforcement function. By combining both of these networks complex tasks have been successfully learnt (Thrun and Möller, 1991)

7.2.2 Implicit levels of cooperation

Similar to chapter The Traffic Simulation we are able to achieve different levels of cooperation. We could have a single network which learnt the different state−action values, although this is suboptimal if different species have different reward values for different items. As an example we also attempted to introduce a predator species that receives a reward for collisions with animals. If we were to use a single network for both species then it would perform badly as the network would not be able to differentiate between penalties for animals bumping into each other and rewards for the predators making a kill. It is very natural to have a neural network for each species, and we would expect the best performance, using this setup, since the networks are given a large amount of experience, in a short amount of time.

If we were to maintain a separate network for each animal, it would require a lot of space and computation. Moreover the simulation would have to be run for a long time before sufficient experience has been obtained for meaningful training to occur. However an interesting option that is available to us if we use individual networks, is to periodically compare each network's performance within a species. If a particular individual is performing well, then it is not difficult to copy the network weights across from the superior network to other inferior networks. This is in effect introducing an evolutionary aspect to reinforcement learning. Some authors maintain that reinforcement learning and evolutionary computation, work well in conjunction with each other (Grefenstette, Moriarty and Schultz, 1999), (Mclean, 2001).

7.3 Results obtained

To effectively compare how well our learning algorithm works we constructed a random species that chooses its moves randomly. We gave this species the same rewards as the animals. We only trained the networks after every 50 simulation steps; we also kept the experience of the previous 50 simulation steps to prevent the networks from forgetting recent information.

In Figure 7.7.4 we can see that for the first 50 simulation steps the performance is comparable with the random strategy; however this is to be expected, since the neural network has been initialised randomly. After the 50 steps, training is performed and the performance begins to increase. We were disappointed that at no point is a strategy learnt that manages to obtain a positive average reward. However we can see the cumulative reward does fluctuate as the training progresses. These fluctuations are caused by the fact that the neural network does forget previous experience; it is trained only on recent experience. While the neural network approach does consistently do better than a random strategy, if one watches the simulation then there is no discernable evidence of learning taking place, both species appear to move around randomly.

Figure 7.7.4 Comparative performance between random animals and collective animals.

7.4 Conclusions

While using a neural network to estimate the Q−table, and then performing a gradient−descent method on the neural network is the simplest extension from discrete to continuous state−spaces, it does not appear to have achieve very good results. This may in part be due to a particular network architecture chosen (although we have repeated the experiment with numerous architectures and the performance is consistently low).

This suggests that other more complex methods are needed if we are to achieve reinforcement learning in continuous state−spaces.

8 Conclusions

This paper provides an introduction to the theory of reinforcement learning. If we wish to use reinforcement to control an agent, then we maintain a table of all possible states, as well as all possible actions. This is called the Q−table. The Q−table estimates the reinforcement an agent will receive if it takes a particular action in a particular state. This gives rise to a greedy agent, which seeks to maximise short−term reinforcement. By distributing the rewards over previous actions using an eligibility trace, we are able to create long−term optimising agents, as is demonstrated in section Example of Maze−Learning where our maze−running agents learn near−optimal policies.

In chapter Blackjack we explore a simplified Blackjack. This simple task allows us to examine some of the effects of using different update rules, as well as the effects on the learning rate of the different constants in an update rule. We find the effects of these constants are minimal, and as such reinforcement learning is robust.

We then attempt the slightly more complex task of maze−running. In this task a solution might take many more steps before the completion of the task; again we examine the choice of different constants in solving this task. Again we find that if we exclude some obviously bad choices of constants, then the reinforcement learning algorithms perform well over a broad range of constants.

This robustness should prove instrumental to reinforcement learning’s success in real world applications. Moreover solving a reinforcement learning task does not require an understanding of what a good strategy should consist of. This makes it possible to use reinforcement learning algorithms as a black−box tool. Neither a full understanding of the environment, nor of reinforcement learning is required to implement a workable solution. However specific knowledge, of the environment or reinforcement learning, can be used to fine−tune solutions to produce better results as we did in section Another updating rule, where we devise a custom updating rule that performs better than the standard rule in learning a maze.

After examining the theory of cooperation, and how it relates to a reinforcement learning system, we then extended reinforcement learning to multi−agent systems. As a possible extension it would be worthwhile to explore how a reinforcement learning system performs on the iterated prisoner's dilemma, not only against itself, but also against other strategies such as TIT FOR TAT.

Chapter The Traffic Simulation presents an examination of the intricacies associated with multi−agent systems. We do this by analysing our traffic simulations. In particular we note that such systems are feasible, but their performance is sub−optimal. This sub−optimality is introduced as the reinforcement learning system is unable to differentiate between penalties from the environment and penalties from other reinforcement learning agents. A possible extension could be to explore giving a different form of reinforcement depending on the origin of the reinforcement (environmental reinforcement or peer reinforcement).

In the final task we present a cellular simulation. This simulation uses a continuous state and action space; however because of this we can not make use of the standard reinforcement learning algorithms which use Q−tables. To overcome this we instead use a neural network which we train on the state and action to predict the reinforcement. By making use of a numerical technique known as gradient descent we are able to use the reinforcement learning framework in a continuous state−space. This solution places no extra constraints on the reinforcement learning framework, so that it is applicable to any continuous state−space for a reinforcement learning algorithm.

However, the performance of this technique was only marginally better than a random strategy, so while it is feasible it is certainly not an optimal way to solve these types of tasks. An obvious extension of this paper would be to then investigate other methods, mentioned in Obtaining the action. Another option is to turn the continuous state−space into discrete blocks, and use the table methods in the preceding tasks.

Our investigation into using reinforcement learning to learn implicit cooperation has been met with only mild success, since we have shown that it is possible, but not optimal. However research into reinforcement learning is still very active and the problems we have encountered will, no doubt, be dealt with in the near future. As such we feel that the answer to the central question of our paper "Can cooperation be learnt through reinforcement learning?" is a qualified "yes".

9 References

(Most of these papers are available at www.citeseer.com)

Axelrod R. (1984) *The Evolution of Cooperation*. Basic Books − New York

Baird L. and Moore A. (1999) Gradient Descent for general reinforcement learning. In Kearns, Solla and Cohn − *Advances in Neural Information Processing systems 11*. MIT Press, Cambridge MA.

Balch T. (1999) Reward and diversity in Multirobot Foraging. In IJCAI−99 Workshop on Agents Learning About, From and With other Agents (1999)

Barto A.G. and Sutton RS. (1998) *Reinforcement Learning: An Introduction*. MIT Press − Cambridge MA.

Bellman R.E. (1962) *Applied Dynamic Programming*. Princeton University Press − Princeton NJ.

Boutilier C. and Price B. (1999) Implicit Imitation in Multiagent Reinforcement Learning *Proceedings in the 16th International Conference on Machine Learning* (325 − 334)

Bücken, Burgard, Fox, Fröhlinghaus, Hennig, Hofmann, Krell, Schimdt and Thrun (1998) *Map Learning and High−speed Navigation in RHINO*. MIT Press − Cambridge MA

Burden R.L. and Faires J.D (1997) *Numerical Analysis (Sixth Edition)* Brooks/Cole Publishing − California

Dayan P. and Hinton G. E. (1993) Feudal Reinforcement Learning. In Lippman D.S., Moody J.E, and Touretzky D.S. (editors) *Advances in Neural information Processing Systems 5* (1993: 271−278) San Mateo CA.

Giles C.L and Jim K. (2000) Talking Helps: Evolving Communicating Agents for the Predator−Prey Pursuit Problem. *Artificial Life (2000 vol 6−3)* 237 − 254.

Grefenstette J.J Moriarty D.E. and Schultz A.C. (1999) Evolutionary Algorithms for Reinforcement Learning. In *Journal of Artificial Intelligence Research*. (1999−11) : 211−276

Harmon M.E. (1996) *Reinforcement Learning: A Tutorial.* Avionics Circle Wright Laboratory − Wright−Patterson OH

Hu H. and Kostiadis K. (1999) Reinforcement Learning and Co−operation in a simulated Multi−agent System. In *Proceedings of IROS'99*. Korea

Hu H., Kostiadis K. and Liu Z. (1999) Coordination and learning in a team of mobile robots. In *Proceedings of IASTED Robotics and Applications Conference*. California CA

Kaelbling L.P., Littman M.L. and Moore A.W. (1996) Reinforcement Learning: A Survey. In *Journal of Artificial Intelligence Research* (1996:4) 237−255

Keerthi S.S. and Ravindran B. (1995) *A Tutorial Survey of Reinforcement Learning.* Sadhana (published by the Indian Academy of Sciences)

Littman M.L. (1994) Markov Games as a frame−work for multi−agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning*. (157 − 163) Morgan Kaufman publishers − New Brunswick NJ

Matari? M.J., Østergaard E.H. and Sukhatme G.S. (2001) Emergent Bucket Brigading. In *Proceedings of the 5th International Conference on Autonomous Agents*. Canada

Mclean C.B (2001) *Design, evaluation and comparison of evolution and reinforcement learning models.* Masters Thesis, Computer Science Department, Rhodes University.

Möller K. and Thrun S.B. (1991) *On Planning and Exploration in non−discrete environments.* Technical Report 528, GMD, Sankt Agustin

Nikovski D.N. (1998) *Fast Reinforcement Learning in continuous action spaces.* Robotics Institute Carnegie Mellon University, Pittsburgh PA

O'Donohue W and Ferguson K.E (2001) *The Psychology of B.F. Skinner* Sage Publications − California CA.

Poundstone W (1992) *Prisoner's dilemma*. Anchor Books − New York.

Schaerf A., Shoham Y. and Tennenholtz M. (1995) Adaptive Load Balancing: a study in Multi−agent Learning. In *Journal of Artificial Intelligence Research (1995:*2) 475−500.

Schmidhuber J. (2000) Evolutionary Computation versus Reinforcement Learning In *Proceedings of 3rd Asia−Pacific Conference on Simulated Evolution and Learning* (SEAL2000), Nagoya, Japan.

Sigmund K. (1993) Games of Life: Explorations in Ecology, Evolution and behaviour. Oxford University Press − Oxford

Singh S.P and Sutton R.S. (1996). Reinforcement Learning with Replacing Eligibility Traces. In *Machine Learning* 22:123−158

Skapura D.M. (1996) *Building Neural Networks*. ACM Press − New York NY

Sutton R.S (1988) Learning to Predict by the Methods of Temporal Differences. In *Machine Learning* (1988:3) 9−44

Tan M. (1993) Multi−agent Reinforcement Learning: Independent vs. Cooperative agents. In Proceedings, Tenth International Conference on Machine Learning. 330 − 337 Amherst MA

Thorpe T.L (1997). *Vehicle Traffic Light Control using SARSA.* Colorado State University

Thrun S.B (1992). The role of exploration in learning control. In *Handbook of intelligent control: Neural, fuzzy and Adaptive approaches*. Van Nostrand Reinhold Publishers− New York NY

Thrun S.B and Möller K (1991) On Planning and exploration in non−discrete environments. . In Lippman D.S., Moody J.E, and Touretzky D.S. (editors) *Advances in Neural information Processing Systems 3* (1991:450−456) San Mateo CA

Touretzky D.S. and Saksida L.M. (1997) Operant conditioning in Skinnerbots. In *Adaptive Behaviour 5* (1997)

Tumer K. and Wolpert D.H. (2001) *An Introduction to Collective Intelligence*. NASA technical report: NASA−ARC−IC−99−63

Tumer K. Wheeler D.H. and Wolpert D.H. (2001) *General principles of learning−based Multi−agent systems*. NASA Ames Research Center