# Residual Reinforcement Learning using Neural Networks

**by**

**Emma Woolford (BCOMP)**

A dissertation submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

**Bachelor of Computing with Honours**

**University of Tasmania**

**(November, 2005)**

## Declaration

## Abstract

A number of reinforcement learning algorithms have been developed that are guaranteed to converge to an optimal solution for look-up tables. However, it has also been shown that these algorithms become unstable when used directly with a function approximation system. A new class of algorithms developed by Baird (1995) were created to handle the problem that direct algorithms have with function approximation systems. This thesis focused on extending Baird's work further by comparing the performance of the residual algorithm against direct application of the Temporal Difference learning algorithm. Four benchmark experiments were used to test each algorithm with various values of lambda and alpha over a period of twenty trials. Overall it was shown that the residual algorithm outperformed direct application of the TD learning algorithm on all four experiments.

## Acknowledgements

Firstly I would like to thank my supervisor Peter Vamplew for his hard work and patience when I possibly asked the same question multiple times.

I would also like to thank my family for their support and putting up with my bad moods. I would especially like to thank my twin Andrea and brother Justin for proof reading.

I give a big thankyou my Dave for his love and support and cheering me up during difficult times and helping me in the final hours, it is very much appreciated.

My friend Rhyanna for always being there when I needed to talk.

Dima, thankyou for making me laugh through your quirky ways, bad taste in music and getting more stressed than me (by being gullible and believing every word I say) and for always providing good advice. Also for our little chats.

Also thankyou to Manny! Steve and Dima again for always being open for discussion on neural nets, when no one else was! To all the other guys in honours who made this year more worthwhile and so memorable by providing heaps of entertainment and laughs. Also thankyou to Steve again for always initiating 'footy o'clock', to get that much needed exercise.

# Contents

## Figures

# Equations

## Tables

# Chapter 2 Introduction

Since computers were invented people have been searching for ways to enable them to think and learn. Numerous methods have been developed in an attempt to solve the problem, all with different levels of success. Much of the work has focused on human knowledge such as expert systems and neural networks. The main problem with relying on human knowledge is the systems that have been created can not learn beyond the knowledge of their 'teacher'. This is because they do not have the ability to investigate these problems themselves. Methods such as genetic algorithms have been successfully developed to learn solutions and do not require human knowledge, however they do not learn through interacting with an environment.

Reinforcement Learning is a form of machine learning. It is based around the concept of an agent with little or no prior knowledge of its surroundings interacting with its environment. The agent observes its environment and at each time step the agent will receive input describing the current state of the environment; it will use this information to select an action to perform. The action which the agent performs affects the state of the environment and the agent will receive a reinforcement signal indicating how desirable that action was to take. The overall task for the agent is to learn a mapping that will maximise the total reinforcement received from state to action. The agent achieves all this with no assistance from an expert and without knowledge of whether an action is correct or what action may have been better to take.

Previously in reinforcement learning techniques have been applied to small state spaces, this means all states are able to be represented in memory individually. Neural networks are often used as a form of function approximation for large problem domains where it is not practical to store absolute state-action values. For example, if there is a large number of states or problems with continuous state variables.

## 2.1 Thesis Hypothesis

This thesis is furthering previous work done by Leemon Baird who created the residual class of algorithms - the residual gradient algorithm and residual algorithm. Baird tested this new class of algorithms on a simple linear system and compared their performance against the TD algorithm. This thesis has extended Baird's work by testing the residual algorithm on different reinforcement learning problems and training a Multilayer Perceptron (MLP) network with backpropagation on reinforcement learning tasks, instead of a linear system.

Previously reinforcement learning algorithms, such as the Temporal Difference (TD) algorithm, have been shown to converge quickly to an optimal solution with look-up tables. However, they have also been shown to become unstable when implemented directly with a general function approximation system such as a linear function approximation system.

The residual class of algorithms aims to improve the problems that the TD algorithm has when used with certain function approximation systems, thus improving the network's ability to converge to an optimal solution. The residual gradient algorithm has guaranteed converge for problems where TD does not but it is slow to learn. The residual algorithm is solves the problems of divergence and slow learning, by having the fast learning speed of the TD algorithm and the guaranteed convergence of the residual gradient algorithm.

The aim of this thesis was to find out if direct use of the Temporal Difference (TD) learning is less effective when training a neural network on reinforcement learning tasks. The TD learning algorithm will be compared against the residual algorithm. It is hoped that the residual algorithm will outperform the more common TD learning algorithm. The purpose for achieving this is to show that the residual algorithm is more likely to converge to an optimal solution than the TD learning algorithm, which has a tendency to diverge. By proving that the residual algorithm converges to a more optimal solution than the TD learning algorithm it is hoped that an improved learning system is created for the field of Artificial Intelligence (AI).

The algorithms will be tested on four benchmark problems that are standard reinforcement learning problems. These problem environments are Gridworld, Mountain Car, Acrobot and Puddleworld.

The first four chapters include all introductory and background information for the thesis – reinforcement learning, neural networks and previous work by Baird. The next two chapters discuss the methods and implementation and also the results obtained from the experiments. Finally, the last chapter concludes the results and discusses further work that could be done in this area.

# Chapter 3Reinforcement Learning

The concept that we learn by interacting with our environment is most likely the first reasoning that occurs to us when contemplating the nature of learning. Watching an infant play, wave its arms or look around, it has no perfect knowledge of its surroundings or an explicit teacher yet the infant is still able to interact and learn from the environment. They have a direct sensori-motor connection to the environment, which produces a wealth of information relating to cause and effect, the consequences of actions and therefore it learns how to achieve certain goals and avoid danger. Learning through interaction with the environment can be described as learning through 'reinforcement' which forms the foundation underlying of learning and intelligence (Sutton and Barto, 1998).

Reinforcement learning can be seen as the problem faced by the agent, which must learn through trial-and-error interactions with a dynamic environment. This is thought of not as a set of techniques but a class of problems that need to be solved. A method that is well suited to solve these problems is considered a reinforcement learning method (Kaelbling et al, 1996).

The following chapter provides a brief background and discusses the fundamentals of reinforcement learning including the model and components that comprise it. Well known reinforcement learning methods such as Dynamic Programming (DP), Monte Carlo (MC) methods and Temporal Difference (TD) learning will be examined. Finally, more advanced reinforcement learning techniques like eligibility traces will be discussed and the need for function approximation in reinforcement learning.

## 3.1 Background of Reinforcement Learning

Reinforcement Learning (RL) is learning what to do in order to maximise a numerical reward. There are two key elements of research which have led to the development of reinforcement learning. The first element revived reinforcement learning in the 1980s and it relates to the psychology in animal training with regard to learning by trial and error. The second key element is the problem associated with optimal control and its solution when using a combination of value functions and Dynamic Programming (Sutton and Barto, 1998).

Thorndike (1911) was the first to explore the theory of learning by trial and error theory. Thorndike introduced the 'Law of Effect' which looks at states that have actions followed by a positive or negative outcome are able to be reselected and altered appropriately. It is referred to as the 'Law of Effect' as it describes the effects of reinforcing events due to the inclination to select actions.

The phrase "optimal control" refers to the problem of designing a controller that minimises the measurement of a dynamic system's behaviour over a period of time. Bellman (1957) created an approach to this problem by expanding a nineteenth century theory developed by Hamilton and Jacobi (Sutton and Barto, 1998).

An individual who was primarily responsible for the reviving of the trial and error thread within reinforcement learning was Klopf. Klopf (1972, 1975) discovered that the fundamental aspects of adaptive behaviour were vanishing as researchers were focusing on supervised learning instead. Klopf believed the following factors were missing in trial and error learning: greedy aspects of behaviour, power to achieve a result from the environment and to control the environment to a desired outcome (Sutton and Barto, 1998).

Later in 1989 Watkins introduced the Q-learning method which is a combination of TD learning and optimal control policy. However, it wasn't until Tesauro (1995) developed the TD-Gammon program that reinforcement learning gained status again (Sutton and Barto, 1998).

## 3.2 Reinforcement Learning Model

Figure 3.1 shows a simplistic view of the reinforcement learning model and how an agent interacts with its environment through actions and its perception of the environment through states and rewards. The agent is the learner and decision-maker; it decides the next action to take based on the input it receives about the current state of the environment. The environment interacts with the agent and comprises everything that is outside the agent. During the next time step the agent then executes this action, observes the immediate reward and the change in the environment (Kaelbling et al, 1996).

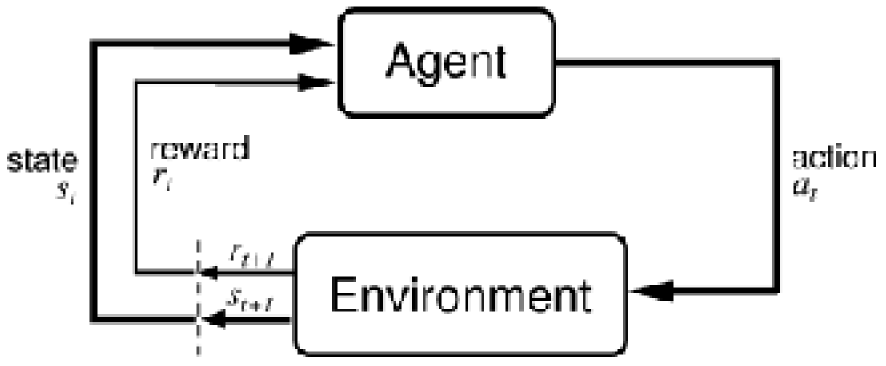


**Figure 2.3.1 Simple agent-environment reinforcement learning model (Sutton and Barto, 1998)**

### 3.2.1 States

A state lets the agent know of its current surroundings in the environment. States can be simple such as readings from a sensor on a robot or more advanced like symbolic representations of objects in an environment. At each time step an agent receives some representation of the environments current state, $s_t \in S$, where $S$ is equal to a set of all possible states. Based on this, an action $a_t$ is selected from a set of possible actions $A(s_t)$ that are available in state $s_t$. In the next time step the agent is in a new state $s_{t+1}$ due to the action it chose (Sutton and Barto, 1998).

### 3.2.2 Goals and Rewards

It can be difficult for an agent to determine an optimal policy as it cannot maximise values of all its observations simultaneously. The two main objectives of an agent is to reach the goal state. The goal state may or may not be explicit. The agent also aims to maximise the total amount of rewards it obtains, not the immediate reward but an accumulation of all the rewards it receives during learning. The goal is what the agent sets out to achieve in order to receive the maximum reward. For example, in a problem like Gridworld the agent normally starts off in a square on one side of the grid while the goal square is on the other side, hence once the agent has reached the goal square it has achieved its goal state and receives a reward of one.

The reward evaluates the agent's behaviour and can be any scalar value. It either indicates an agent's success when the goal state has been reached or failure to reach the goal state. It also may provide instantaneous evaluation of continuous behaviour by the agent. To achieve a goal the rewards must be provided in such a way that by maximising the rewards the goal is reached. This means the rewards need to be organised in a certain way indicating what needs to be accomplished. For example, if an agent was playing chess and it was rewarded for sub-goals like taking an opponents piece then the agent may lose more often as it is receiving rewards for the sub-goals. The problem with this is its less likely to reach the actual goal of winning (Sutton and Barto, 1998).

Reinforcement learning problems generally have some form of reward structure, indicating the amount of reward received for certain actions. An example is a real-world problem that was solved by Randlov and Alstrom. It involved an agent using reinforcement learning techniques to learn to ride a bicycle with online reinforcement learning, using the Sarsa (λ) algorithm. The agent received -1 when the bicycle fell over and was rewarded with r = 0.01 if the agent reached the goal. The experiment was successful and while the first ride to the goal could be as long as two hundred kilometres, after a few rides the agent quickly learnt to reach the goal in approximately seven kilometres (Randlov and Alstrom, 1998).

### 3.2.3 The Value Function

The value function can be a scalar function of a state or a state-action pairs. From a current state the value function focuses on the total reward a state expects to accumulate over time instead of obtaining the immediate reward. For example, it is possible to choose a state that has a low reward instead of a state with a high reward. This is due to the fact that the state with the lower reward may have a higher value due to subsequent states yielding greater rewards thus guiding the agent to a more desirable outcome (Sutton and Barto, 1998).

As a result, the value function is the deciding factor when choosing an action as the agent seeks an action that offers the highest state value instead of the highest reward, therefore if a state has a high value then so will the action. The disadvantage in using value functions is they are difficult to determine as they need to be estimated and re-estimated from the observations the agent makes over its lifetime (Sutton and Barto, 1998).

### 3.2.4 The Agents Policy

The goal of the agent is to find the optimal policy, which maximises the agents' future reward. The majority of current reinforcement learning techniques are based on the agent estimating the value of its actions i.e. the agent receives a reinforcement signal (reward) indicating how desirable the action they took was. In order for the agent to determine which action to perform at each time step, it will apply a mapping from perceived states to the probability of selecting every possible action when in those particular states. This mapping is known as the *agent's policy* which is denoted by $\pi_t$ where the probability that $a_t = a$ if $s_t = s$ is equal to $\pi_t(s, a)$. Reinforcement learning methods aim to optimise the mapping by specifying changes an agent makes to its policy based on the agents experience. Hence an agent's goal is to maximise its total amount of rewards over a period of time.

For an agent to maximise these rewards it must learn to select actions that were successful earlier in providing a reward, however for the agent to discover such actions it needs to test actions it has not previously attempted. Therefore the agent needs to exploit its current knowledge to obtain a reward; equally it must explore to enable improved selection of actions in the future (Sutton and Barto, 1998). The problem with this approach is neither exploitation nor exploration can be used separately without the agent failing at its task; hence a balance is needed in order for the agent to be successful.

## *3.3 Evaluative Feedback and Exploration*

The most definitive feature of reinforcement learning that differentiates it from other types of learning is the actions performed are evaluated instead of being given the correct actions through instructions. It is this that creates the need for active exploration of the problem using a trial-and-error search to discover good behaviour. The primary use of evaluative feedback is to indicate the benefit of the action taken however it doesn't identify whether the action taken was the best or worst possible action to be selected. Evaluative feedback is also the basis for other methods of function optimisation, including evolutionary methods (Sutton and Barto, 1998). Using trial-and-error indicates that the evaluative approach would introduce action selection.

How an agent selects its actions is important. For an agent to obtain a large amount of rewards, it must select actions it has chosen in the past that have also been effective in producing a reward. In order to discover such action an agent must attempt actions it has previously not selected. An agent needs to select exploratory actions occasionally to allow for better action selection in the future.

### 3.3.1 Greedy Methods

The simplest rule when selecting an action is to select the action with the highest estimated action value. This rule is called the *greedy method*. If an action appears to have a high total reward compared with other actions that action will always be chosen as the greedy method exploits its current knowledge of a situation in order to achieve maximum reward. The greedy method doesn't 'waste' time testing other supposedly inferior actions regardless of whether they are better or not. This means that an action that has yet to be tested may yield a higher reward than the one the greedy method believes possesses the highest reward. Another problem that occurs with this approach is when an action produces a small reward on its first try, it can be given a low value even if the action is generally beneficial. These types of problems demonstrate the necessity of striking a balance between actions that exploit paths which are guaranteed to return high rewards and exploratory actions which may take extra time but can produce a more optimal solution in the future (Sutton and Barto, 1998).

An alternative to the greedy method is the *ε-greedy* method, it behaves greedily the majority of the time but occasionally with a small probability ε it will randomly select an action which is independent of the action-value estimates. Sutton and Barto (1998) discovered when tested on the 10-armed testbed problem the ε-greedy method improved on the greedy method as the greedy method performed significantly worse over a long period due to it regularly having to perform sub-optimal actions. The ε-greedy method eventually performed better over the long period as it continued exploring thus improving the possibility of determining the optimal action.

The advantage of ε-greedy compared with greedy is dependant on the task. For example, if the reward variance was a large number such as ten instead of one and the rewards were noisy then ε-greedy would still perform slightly better than greedy even though more exploration would be necessary.

However if the reward variance was zero then the greedy approach would achieve superior results as it would know the true value of each action after one attempt, assuming values are initialised optimistically and hence find the optimal action without exploring (Sutton and Barto, 1998).

## 3.4 Markov Decision Processes

Markov Decision Processes are common in the field of reinforcement learning and exist when a reinforcement learning task fulfils the *Markov Property.* In order for a problem to have the Markov Property its state signal must ideally summarise past sensations that include all relevant information. The environment can also be considered to have the Markov Property if the subsequent state and expected return from the current state can be predicted accurately (Sutton and Barto, 1998).

The Markov property is important to the field of RL because similarly to MDP's, the RL problem assumes its decisions and values to be functions only of the current state. Hence, the Markov case enables the behaviour of algorithms to be understood and the algorithms can be applied successfully to many other tasks with states which are not strictly Markov.

If a particular MDP has a finite number of states and actions then it is known as a *finite* MDP. Given any state, $s$, and action, $a$, the probability of each possible next state $s'$ is defined by quantities called transition probabilities $P_{ss'}^{a} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$ and the expected immediate rewards, given any current state, $s$, and action, $a$, combined with any next state, s′, is $R_{ss'}^{a} = E\{r_{t+1} \mid a_t = a, s_t = s, s_{t+1} = s'\}$. $P_{ss'}^{a}$ and $R_{ss'}^{a}$ are considered important dynamics of a finite MDP.

An optimal policy with a finite MDP can be achieved by value functions defining a partial ordering over a set of policies through solving the Bellman optimality equation. A policy $\pi$ is defined to be better or equal to a policy $\pi'$ for all states if its expected reward is greater or equal to $\pi'$. This type of policy is called an *optimal policy.* One or many optimal policies are denoted by $\pi^*$ (Sutton and Barto, 1998).

## 3.5 Reinforcement Learning Methods

In this section three methods that are capable of solving the reinforcement learning problem with varying levels of efficiency and speed of convergence will be discussed: *Dynamic Programming*, *Monte Carlo* and *Temporal Difference Learning*.

Dynamic Programming methods are highly developed mathematically and require a Markov model of the environment that must be complete and accurate. Unlike Dynamic Programming, Monte Carlo methods are simplistic and do not require a model of the environment. However, they perform poorly on step-by-step incremental computation. The final method, TD Learning, is a combination of Monte Carlo and Dynamic Programming. Similarly to the Monte Carlo method, TD does not require a model but is fully incremental. The difficulty with TD is it's more complex to analyse (Sutton and Barto, 1998).

### 3.5.1 Dynamic Programming

The phrase Dynamic Programming (DP) is given to the set of algorithms that are capable of computing optimal policies if given a perfect model of the environment as a MDP. The environment is assumed to be a finite MDP, meaning the state and action sets, $S$ and $A(s)$, for $s \in S$ are finite and its dynamics are given by transition probabilities (2.4) and expected immediate rewards (2.4). Although DP algorithms have limited usefulness in reinforcement learning due to their assumptions of the perfect model and being computationally expensive DP algorithms are still important theoretically.

The primary idea of DP and also more generally in reinforcement learning is the use of value functions to organise and search for respectable policies (Sutton and Barto, 1998).

An optimal policy can easily be achieved if the optimal value functions $V^*$ or $Q^*$ have been derived to satisfy the Bellman optimally equation. This can be accomplished by converting certain Bellman equations into update rules that improve approximations of desired value functions. There are three algorithms for finding the optimal policy in dynamic programming: *policy iteration* which is able to converge in a few iterations and *value iteration* which reduces the time and complexity of policy iteration, it does this by truncating policy evaluation after just one sweep (one backup of each state). Finally, *asynchronous dynamic programming* which primarily improves flexibility (Sutton and Barto, 1998).

### 3.5.2 Monte Carlo Methods

Monte Carlo methods only require experience, not knowledge of the environment; i.e. sample sequences of states, actions and rewards all created from on-line or simulated interactions with an environment. Learning from on-line experience does not involve prior knowledge of the environments dynamics and still achieves optimal behaviour. If learning from simulation experience a model that can generate sample transitions is required but not to the extent of the complete probability distribution that is used in Dynamic Programming (Sutton and Barto, 1998).

Monte Carlo methods are an alternative approach to solving reinforcement learning problems based on averaging returns and are only defined for episodic tasks as well-defined returns are guaranteed. In completion of an episode the value estimates and policies are changed thus Monte Carlo methods are incremental in the episode-by-episode sense rather than the step-by-step meaning (Sutton and Barto, 1998).

## *3.6 Temporal Difference Learning*

Temporal Difference (TD) learning is a core component of many reinforcement learning techniques. It is considered the unification of Dynamic Programming and Monte Carlo methods as TD learning performs certain actions identical or similar to Dynamic Programming and Monte Carlo methods (Sutton and Barto, 1998).

TD learns from raw experience hence TD learning does not require a model of an environment. TD learning is similar to Dynamic Programming in that both calculate current estimates based partly on previous estimates; this process is referred to as bootstrapping. TD learning and Monte Carlo techniques use a process called sampling, which means when updating they do not incorporate an expected value. TD has the ability to continue to learn before knowing and without being aware of the final outcome (Sutton and Barto, 1998).

### 3.6.1 TD-Prediction

To solve a prediction problem TD and Monte Carlo techniques both use experience. As mentioned in section 2.6 TD has the ability to continue to learn without waiting for the final outcome. Hence TD only waits until the next time-step compared with the Monte Carlo technique which must wait till the end of an episode before it is able to update its estimate $V(\mathrm{s}_t)$. At the time $t$+1 TD immediately forms a target, making a valuable update by using an observed reward $\mathrm{r}_{t+1}$ and estimate of the value state at time $t$+1 $\mathrm{V}(\mathrm{s}_{t+1})$. The $\alpha$ is a constant step-size parameter and $\gamma$ is the discount. Equation 2.1 shows the simplest form of the TD method, recognised as TD(0).

$$V(\mathrm{s}_t) \leftarrow V(\mathrm{s}_t) + \alpha\,[\mathrm{r}_{t+1} + \gamma\, V(\mathrm{s}_{t+1}) - V(\mathrm{s}_t)]$$

**Equation 2.3.1 TD(0) estimation of state values**

TD is seen as a combination of both Monte Carlo and Dynamic Programming techniques as the TD method calculates the target as an estimation of an expected value using sample returns as the real value is unknown like Monte Carlo and similar to DP, TD can also use an estimate of the expected value provided by the model (Sutton and Barto, 1998).

## 3.6.2 Sarsa: On-Policy Control

When using TD prediction for the control problem, the methods can be classified as either on-policy or off policy. In this section the on-policy TD control method will be presented. Sarsa is an on-policy algorithm, meaning it trains on an action non-greedily and executes that same action. It learns action-value functions instead of state-value functions. For an on-policy algorithm $Q^{\pi}(s, a)$ must be estimated for the current behaviour policy π including all states *s* and actions *a*.

The update rule for TD in Equation 2.2 is performed after every transition from a non-terminal state $s_t$. If $s_{t+1}$ is terminal then $Q\ (s_{t+1}, a_{t+1})$ is defined as zero. This particular rule uses each element of the quintuple of events ($s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$), that comprises the transition from one state-action pair to the next (Sutton and Barto, 1998).

$$Q\ (s_t, a_t) \leftarrow Q\ (s_t, a_t) + \alpha\ [r_{t+1} + \gamma Q\ (s_{t+1}, a_{t+1}) - \mathrm{Q\ (st, at)}]$$

**Equation 2.3.2 Update rule for state-action pairs in TD(0)**

The Sarsa algorithm is shown in Figure 2.2. Sarsa's convergence properties are dependant on the action selection used, whether state-action pairs are visited an infinite number of times and also if the policy converges in the limit to the greedy policy (Sutton and Barto, 1998).

```
Initialise Q(s, a) arbitrarily
Repeat (for each episode):
    Initialise $s$
    Choose $a$ from $s$ using a policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $a$; observed reward $r$, and next state $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$; $a \leftarrow a'$;
    Until $s$ is terminal
```

**Figure 2.3.2 Sarsa: On-policy control algorithm**

### 3.6.3 Q-Learning: Off-Policy Control

Q-Learning is an off-policy TD control algorithm created by Watkins (1989). Off-policy means that it trains on an action that is selected greedily but performs that action non-greedily. Q-learning estimates the value of state action pairs, when these values have been learned the optimal action is considered to be the one with the highest Q-value (Sutton and Barto, 1998). Equation 2.3 shows the simplest structure of Q-learning which is the one-step.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

**Equation 2.3.3 One-step Q-Learning**

The Q-learning one-step equation has been proved to converge to a finite state of Markovian problem when used in conjunction with a look-up table that stores the value of a Q-function. The optimal policy is obtained by selecting an action in each state that has the highest predicted return, once the Q-function has converged (Rummery and Niranjan, 1994).

```
Initialise Q(s, a) arbitrarily
Repeat (for each episode):
    Initialise $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using a policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$;
    Until $s$ is terminal
```

**Figure 2.3.3 Q-Learning: Off-policy control algorithm**

Figure 2.3 shows the Q-Learning algorithm in practical form. Convergence will occur if the following three factors hold (1) all state-action pairs are visited an infinite number of times (2) function approximation has not been used and (3) a Markov problem is used.

## *3.7 Advanced Reinforcement Techniques*

Currently only the three methods for solving the reinforcement learning problem have been discussed: Dynamic Programming, Monte Carlo and TD Learning. Even though these methods have differences they can be unified to create a more useful function (Sutton and Barto, 1998). In the following section the mechanism of eligibility traces will be introduced.

### 3.7.1 Eligibility Traces

An eligibility trace can be defined as the recording of the occurrence of an event i.e. visiting a state. The Temporal Difference algorithm TD(λ) is a common form of an eligibility trace used to average n-step backups. N-step backups were introduced to aid in the explanation of how traces worked, although they proved not as usual as they needed to know the future (Perez-Uribe, 1998). All TD algorithms (e.g. Q-Learning and Sarsa) can be extended to use eligibility traces.

There are two different views of eligibility traces; the theoretical and the mechanistic. The theoretical view looks at the idea that eligibility traces bridge from TD and Monte Carlo methods. The outcome of TD methods being enlarged with eligibility traces is they produce a group of methods spanning a spectrum with Monte Carlo methods at one end and TD at the other. The mechanistic view looks at the eligibility trace as a temporary record of an event. If a TD error occurs only eligible states or actions are given either credit or blame for that particular error, hence eligibility traces bridge the gap between events and training information (Sutton and Barto, 1998).

### 3.7.2 TD(λ)

Monte Carlo methods perform a backup for every state that is based on the entire sequence of rewards observed from that state til the end of the episode. TD(λ) performs these backups toward an *average* of *n*-step returns. As stated in Sutton and

Barto (1998) a backup can be conducted toward a return that is half of a two step return and half of a four step return e.g. $R_t^{ave} = \frac{1}{2}R_t^{(2)} + \frac{1}{2}R_t^{(4)}$ Any set of returns including infinite sets can be averaged this way providing the weights on the component returns are positive and sum to the total of one. The TD(λ) algorithm is simply an alternative way of averaging *n*-step backups, The *n*-step backups which form the average are weighted proportional to $\lambda^{n-1}$ where 0≤λ≤1. The normalisation factor 1-λ guarantees the weights sum up to one; this gives the resulting backup towards a return called the *λ-return* as shown in Equation 2.4 (Sutton and Barto, 1998).

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^{(n)}$$

**Equation 2.3.4 General λ-return function**

A one-step return is given the largest weight $1-\lambda$, with each increasing step the weight decreases by λ until a terminal state is reached each subsequent *n*-step return is equivalent to $R_t$. Separating these terms from the main sum gives Equation 2.5.

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}R_t^{(n)} + \lambda^{T-t-1}R_t$$

**Equation 2.3.5 Extending the λ-return function**

In the above Equations if λ = 1 the main sum will eventually equal zero and the remaining term will reduce to the conventional return $R_t$ thus backing up λ-return will be identical to the Monte Carlo algorithm. However if λ = 0 then the λ-return is reduced to $R_t^{(1)}$ which is equal to one-step TD-method, TD(0) (Sutton and Barto, 1998).

## 3.7.3 Implementation of TD(λ)

The above section presented the forward view of TD(λ) which is not directly implemental as it is *acausal*, meaning it uses knowledge at each step that describes what will happen in future steps. This section will discuss the mechanistic view or the backward view of TD(λ) due to it being simple both conceptually and computationally. The backward view provides an incremental mechanism for approximating the forward view and when in the off-line case it achieves it exactly.

Additional variables exist when using the backward view of TD(λ) that are associated with each state, these variables are called *eligibility traces*. The eligibility trace is denoted $e_t(s)$ for a state *s* at time *t* and for every state at each step it decays by γλ, with the eligibility trace for visiting one state on the step incremented by one. Initially $e_t(s)$ is equal to zero until state s is visited in which it is then incremented by one this will accumulate each time the state is visited hence if the state is not visited it will slowly diminish, this is shown formally below in Equation 2.6.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t; \end{cases}$$

**Equation 2.3.6 Accumulating eligibility trace**

The γ refers to the discount rate and the λ is *trace-decay parameter* for all non-terminal states *s*. The traces indicate the amount for which each state is eligible to experience learning changes on the chance a reinforcement learning event occurs. However the increments can be performed on each step to form an on-line algorithm or alternatively saved until the end of an episode to produce an off-line algorithm. The Equation above provides the mechanistic definition needed for the TD(λ) algorithm. The on-line TD(λ) algorithm is given below in Figure 2.4.

```
Initialise $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$
Repeat for each episode:
Initialise s
Repeat (for each step of the episode):
        $a \leftarrow$ action given by $\pi$ for $s$
        Take action $a$, observe reward $r$ and next state $s'$
        $\delta \leftarrow r + \gamma V(s') - V(s)$
        $e(s) \leftarrow e(s) + 1$
        For all $s$:
                $V(s') \leftarrow V(s') + \alpha\delta\, e(s)$
                $e(s) \leftarrow \gamma\lambda\, e(s)$
        s $\leftarrow s'$
```

```
Until s is terminal
```

**Figure 2.3.4 On-line TD(λ) algorithm**

Previous work done in the reinforcement learning field usually applies techniques to problems with small state spaces, where all the states can be easily stored individually in memory. However, for problems with larger state spaces, all the states can not be stored this is where neural networks are used as a form of function approximation to enable the storage of all state action values. Neural networks with function approximation systems are also used for problems with continuos state variables. The next chapter will discuss basic neural network theory and the basis and implementation of the multilayer perceptron model. More advanced concepts of neural networks are not discussed due to their irrelevance to this thesis topic.

# Chapter 4 Neural Networks

As previously mentioned in the reinforcement learning chapter, reinforcement learning requires a device for storing states but it can only handle small state spaces. For a problem domain with a large number of states there needs to be a way of generalising the state values. Neural networks are used as a form of function approximation for such problems where it is not practical to explicitly store all state-action values.

Neural networks consist of input and output nodes used for sending and receiving data and also have a layer of hidden nodes. Neural networks are a good tool for performing pattern recognition. Neural networks simply take particular features of the input data it has received and begins classifying them accordingly. This means an agent can learn states values as well as the grouping of those states by using the neural network for state generalisation.

This chapter first gives a brief background of artificial neural networks, and then the biological neuron will be discussed to providing a basis for the artificial neuron. The following sections will describe the relatively new form of neural network – the multilayer perceptron and also associated learning rules.

## 4.1 Background of Artificial Neural Networks

The first artificial neural network began in 1943 with McCulloch and Pitts, who proved that an artificial neural network model could calculate arithmetic or logical functions (Hagan et al, 1996). The model consisted of two types of input: excitory and inhibitory. If the excitory inputs were larger than the inhibitory inputs then this caused the neuron to "fire" thus creating an output (Sondak and Sondak, 1989). This started the pathway for more research into artificial networks. Hebb followed with his work. Hebb suggested that the presence of classical conditioning existed due to the properties of individual neurons. Hebb also proposed a method for biological neurons to learn (Hagan et al, 1996).

It was about ten years after Hebb that the next major influence in the development of neural networks occurred with research by Rosenblatt (1958). His main success was the creation of the perceptron network that was able to perform pattern recognition. Not long after Widrow and Hoff (1988) created an ADAptive LINEar neuron (ADALINE) which had a similar structure to Rosenblatt's perceptron model. It was considered a pattern classification device and demonstrated key concepts of adaptive behaviour and learning (Hagan et al, 1996). However it was discovered by Minsky and Papert (1969) that the perceptron and ADALINE networks could not manage large classes of problems but were only able to perform well in limited problems thus not providing much real value (Sondak and Sondak, 1989).

This finding had a profound effect on research in the neural network field and delayed any major work in the area for about ten years. During that time small significant steps were taken by Kohonen, Anderson and Grossberg in furthering the successful future development in neural networks.

It wasn't until the 1980s when research started to accelerate again, Hopfield discovered that by using non-linearities an artificial neural network was able to solve a constrained optimisation problem. Hopfield's work was shown to accomplish practical problems and Hopfield nets were able to find suitable routes easily and quickly in constrained optimisation problems (Sondak and Sondak, 1989).

Although Minsky and Papert (1969) showed that the perceptron and the ADALINE were only able to handle limited problems, they also found that a Multilayer Perceptron (MLP) was capable of solving a more complicated problem - the XOR problem. However, no training algorithms existed that were able to be used on an MLP. In the 1980's the backpropagation algorithm was developed, it was this algorithm that provided the training algorithm needed for MLP's. It is designed to adjust weights starting from the output layer in a multi-layer feed forward network and working its way back through the network (Beale and Jackson, 1990).

## 4.2 The Biological Neuron

This section discusses the biological neuron which provides the basis of the artificial neural network methods that will be discussed in the following sections. The artificial neuron is based on the brain of a human/animal, hence both have similar attributes and are used send (output) and receive (input) messages. The human brain is extremely complex and while it has been studied immensely there is still only a basic understanding of how the brain operates at a low level. The brain contains approximately ten thousand million neurons and for every neuron there is another ten thousand attached. The neuron is the essential processing unit of the brain; it consists of three main parts: the cell body also referred to as *soma,* the *axon* and the *dendrites* as shown below in Figure 3.1 (Beale and Jackson, 1990).

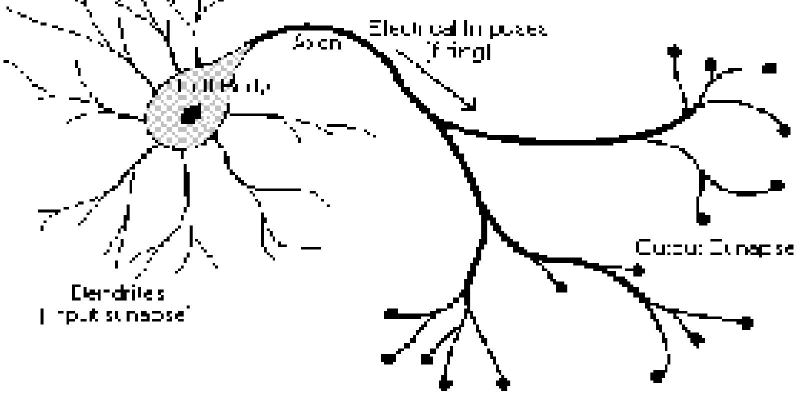


**Figure 3.4.1 A single neuron (Andreopoulos, 2004)**

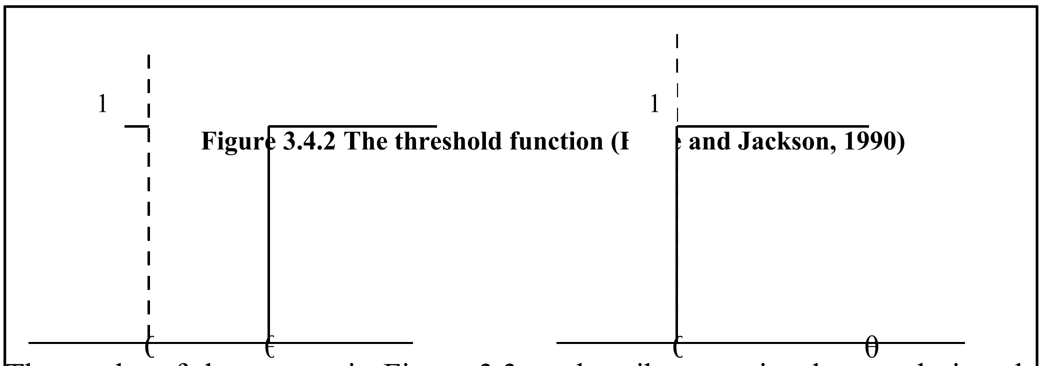
The soma has long irregular filaments called dendrites branching from it and contains the neuron's nucleus. The dendrites task is to receive input messages and pass these messages to the *soma*. The other type of nerve process which is attached to the soma is the axon; it is electrically active and acts as the output channel of the neuron. Axons are non-linear threshold devices and produce a voltage pulse also known as the action potential, lasting approximately one millisecond. This occurs when the resting potential of the soma rises above the critical threshold. The axon will terminate when a type of specialised contact called synapse occurs which couples the axon with a dendrite from another cell (Beale and Jackson, 1990).

There are two types of neurons that exist: the *interneuron* cells which handle local processing. The second type of neuron is the output cells; they connect different sections of the brain to together for example: connecting the brain to muscle or from the sensory organs to the brain. If any neuron receives enough active inputs simultaneously it will cause the neuron to be activated and thus will "fire", otherwise the neuron will remain in a silent inactive state (Beale and Jackson, 1990).

## 4.3 The Artificial Neuron

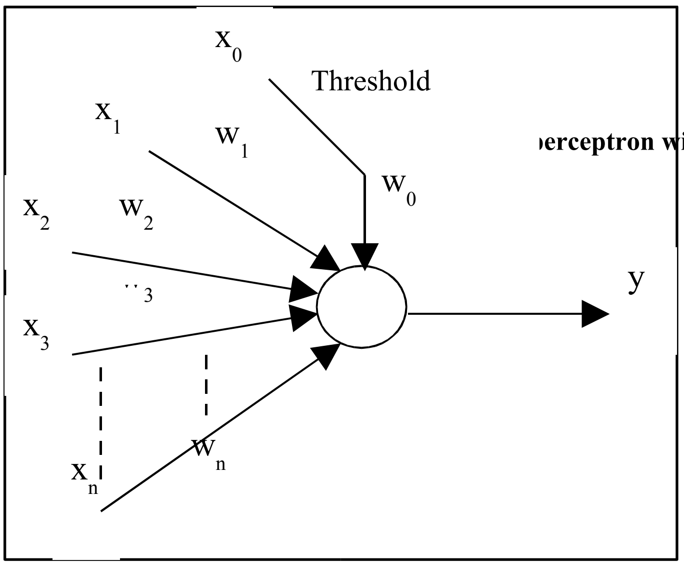
The artificial neuron was originally called the *perceptron* and was designed to capture the most important features of a biological neuron. These features can be summarised as follows: the output relies on the inputs, the output from a neuron is either on or off and a neuron only fires if a known value called *threshold* is exceeded. A neuron operates by performing a weighted sum of its inputs; this sum will then be compared to an internal threshold and hence the neuron only turns on if the threshold is exceeded. If the level is not exceeded then the neuron stays off. Hence if the weighted sum is greater than the value of the threshold then the output will equal one, otherwise if the output is less than the threshold value the output will equal zero.

Figure 3.2 illustrates this process, where the x-axis corresponds to the input and the y-axis is equal to the output (Beale and Jackson, 1990).



**Figur 3.4.2 The threshold function (I and Jackson, 1990)**

The mod f the ron in Figure 3.3 as descrit previously lesigned to be simple thus it does not possess any of the complicated features that a biological neuron possesses and hence it can be implemented easily on a computer. Figure 3.3 illustrates the basic perceptron with inputs labelled $x_0$ through to $x_n$ and the corresponding weights, the first input $x_0$ and weight $w_0$ provide the threshold value and the output y.



**ıerceptron with biases**

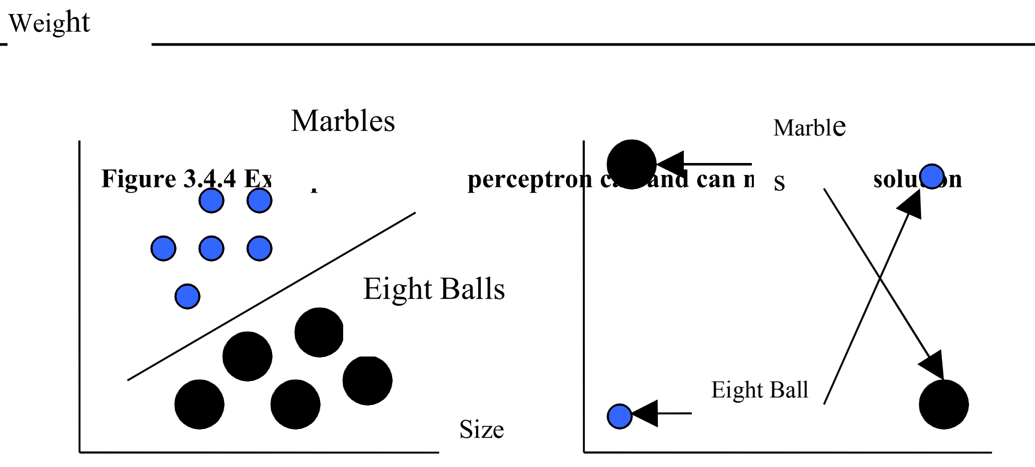
### 4.3.1 Learning using Perceptrons

The ability for a network to learn makes them truly useful; in order to keep the model of the neuron understandable it needs to have a simple learning rule. Ideas for learning rules can often stem from real neural systems. For example it is observed that humans and animals learn from situations that good behaviour is praised and reinforced and bad behaviour is discouraged.

This concept can be transferred into neural networks where good behaviour should be encouraged so it is repeated and bad behaviour is not. For example, two types of objects such as circles and squares need to be differentiated by a neuron. If the object is a circle the output should be one and the output is zero if the object is a square. In principle the neuron should learn from its mistakes. If an incorrect output is produced then the chance of an incorrect output occurring again should be reduced, therefore if it produces the correct output then nothing is done. As mentioned in section 3.1.1 the neuron performs a weighted sum and compares it to the threshold determining if the output will be one or zero. If the neuron correctly classifies the object then no action will be taken as the model is successful. Alternatively if the neurons output is zero when the object is shown a circle then the weighted sum needs to be increased. By increasing the weighted sum it will cause the threshold to be exceeded and hence produce the correct output. In order to increase the weighted sum the weights need to be increased, if the inputs are squares then the weights need to be decreased as the weighted sum needs to be less than the threshold to produce zero as the output. Hence the input weights are strengthened when the output needs to be on and weakened if the output is off. This defines the learning rule and is called Hebbian learning as it allows connections to be strengthened or weakened; this is a slight variation on the learning rule Hebb created which refers to active connections only (Beale and Jackson, 1990).

### 4.3.2 Problems with Perceptrons

The problem with perceptrons is their inability to solve non-linearly separable problems. If a problem is not linearly separable by type then the perceptron is unable to find a solution. For example, a network is trying to distinguish whether a round object is a marble or an eight ball and both objects possess different sizes and weights. When the network is given these two attributes as input, it should learn over time and eventually be able to differentiate between the two rounded objects and should be able to be separated by a line, if represented graphically, as illustrated in Figure 3.4 (a). However, if this was represented as an XOR problem the outcome would be different as it is possible for both inputs to be the same for marbles and the inputs different for eight balls, therefore it would not be possible to draw a line separating the objects as demonstrated in Figure 3.4 (b).



(a) Separation of marbles above the line and eight balls below the line.

(b) Can not separate marbles from eight balls for the XOR problem.

### 4.4 Multilayer Perceptron

A possible solution to the XOR problem would be to have multiple perceptrons with each one identifying a small linearly separable section of the inputs; the outputs are then combined into a third perceptron to produce a type of class that the input belongs too, as shown in Figure 3.5. Although this seems fine, the perceptrons in the layers are unable to learn by themselves. The second layer takes its input from the first layers output; the second layer can only adjust its weights and not the weights of the first layer which is vital in order for the network being able to learn. The only information the second layer has is the state of the neuron: ‘on’ or ‘off’, this gives no indication if the weights need to be adjusted or not. This lack of information is due to the actual inputs being masked from the outputs of the intermediate layers hard-limiting threshold function (Beale and Jackson, 1990).

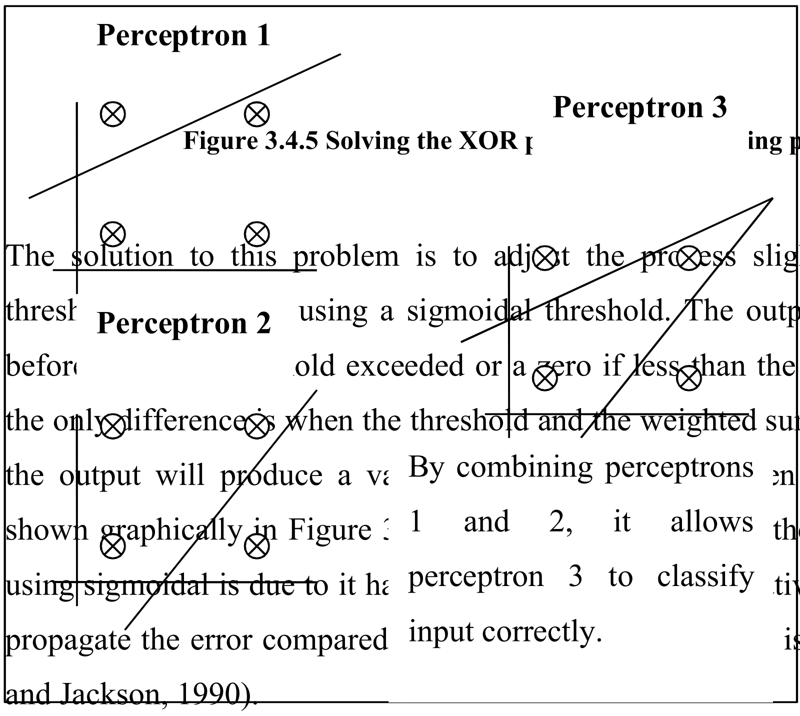

Perceptron 1
Perceptron 2
Perceptron 3
By combining perceptrons 1 and 2, it allows perceptron 3 to classify input correctly.

Figure 3.4.5 Solving the XOR p ing perceptrons

The solution to this problem is to adjust the process slightly by smoothing the thresh using a sigmoidal threshold. The output will be the same as befor old exceeded or a zero if less than the value of the threshold, the only difference is when the threshold and the weighted sum are almost equal then the output will produce a va en the two extremes. As shown graphically in Figure 3 the centre, the reason for using sigmoidal is due to it ha tive this allows it to back propagate the error compared is not continuous (Beale and Jackson, 1990).

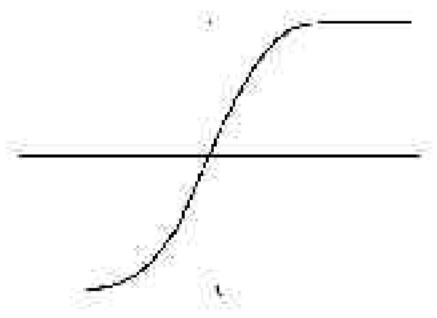


**Figure 3.4.6 Sigmoidal Threshold Function**

$$f(net) = 1/\left(1 + e^{-k^{net}}\right)$$

**Equation 3.4.1 Sigmoid Function**

Equation 3.1 shows the sigmoid function; the k is a positive constant that influences the spread of the function, $f(net)$ will approach the step function if k approaches infinity and $f(net)$ must be between the values of zero and one (Beale and Jackson, 1990).

### 4.4.1 The New Model

The new modified model of the perceptron is known as the *multilayer perceptron*. The new model consists of an input layer, an output layer and a layer in between which is neither connected directly to the input or the output layer. This layer is called the *hidden* layer. Usually a network only has one hidden layer; however, it is possible to have more than one. Each unit within the hidden and output layers acts as a perceptron unit, while the units in the input layer operate by dispensing values to the subsequent layer. This means the input layer does not perform a weighted sum and comparison, due to the model being modified from a single layer perceptron using a non-linearity step function to a sigmoid function with an additional hidden layer. Due to these changes in the model the learning rule must be altered.

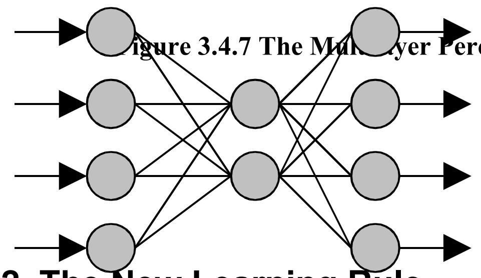


**Figure 3.4.7 The Multilayer Perceptron (Beale and Jackson, 1990)**

## 4.4.2 The New Learning Rule

The new learning rule for the multilayer perceptron is known as the *generalised delta rule* or the *backpropagation rule*. The new rule was proposed by Rumelhart, McClelland and Williams in 1986, although previous works had been done separately by Parker (1985) and Werbos (1974) that suggested Rumelhart, McClelland and Williams were not the first to recommend this new learning rule.

The operation of the multilayer network is similar to the single layer network except the learning rule is not sufficient and a more complex version is needed. As mentioned in section 3.4 the sigmoid function provides extra information needed for the new learning rule to be established. The network is first presented with an input pattern which produces a random output; this output is used when defining the error function that calculates the difference between the current output and the correct output. In order for the network to learn successfully the value of the error function must be reduced, to achieve this weights between the units are adjusted by a small amount. The error is then propagated back through the network to the hidden layer to allow for the units to adjust their weights (Beale and Jackson, 1990).

Equation 3.2 calculates the amount of error that is applied to a unit, $t_{\mathrm{pj}}$ signifies the target output and $o_{\mathrm{pj}}$ represents the actual output. The function $f'_j\left(net_{pj}\right)$ is the derivative and is used for the activation at unit $j$ which is in the output layer and the letter *p* is denoted as the error for a particular set of inputs. The function in Equation 3.2 can not be used for hidden units as the target output is unknown; Equation 3.3 refers to hidden units.

$$\delta_{pj} = f'_j\left(net_{pj}\right)\left(t_{pj} - o_{pj}\right)$$

**Equation 3.4.2 Error calculation for output units**

The hidden function works by using the error that is given from the output node in Equation 3.2 and is propagated back through the network to the hidden units and is implemented in Equation 3.3.

$$\delta_{pj} = f'_j\left(net_{pj}\right)\sum_k \delta_{pk} w_{jk}$$

**Equation 3.4.3 Error calculation for hidden units**

## 4.4.3 Multilayer Perceptron Classifiers

The purpose of the multilayer network is to solve more complex problems that a single perceptron would struggle with. A basic perceptron can only accomplish single linear separation problems; however a multilayer perceptron is able to learn more complicated shapes due the use of hidden layers. If the hidden layer and output layer both only contain one unit then it can only produce single linear separation, if the hidden layer consisted of more than one unit then the network is capable of finding two or more linear separations. In order for the unit of the second layer to produce a classification it performs a logical *AND* function on the resulting values from the units in the first layer that defined a line in pattern space. More complex shapes can be achieved as shown in Figure 3.6 simply by adding additional hidden units to the hidden layer (Beale and Jackson, 1990).

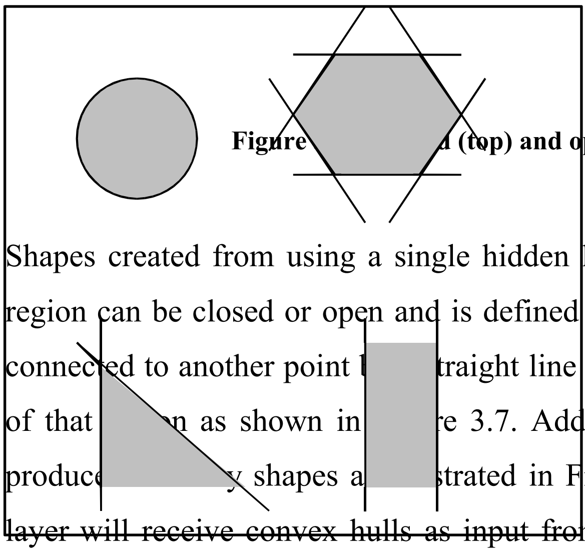
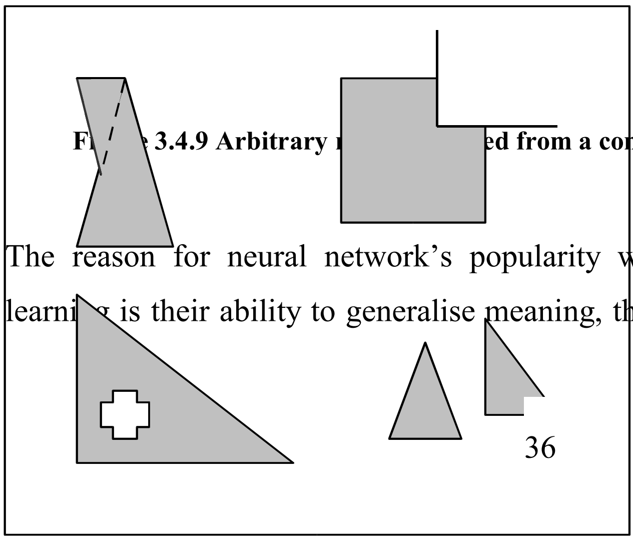


**Figure [illegible] (top) and open (bottom) convex hulls**

Shapes created from using a single hidden layer are referred to as *convex hulls.* A region can be closed or open and is defined to be a convex hull if any point can be connected to another point [illegible]traight line which does not cross over the boundary of that [illegible]n as shown in [illegible]e 3.7. Adding an additional layer of hidden units produce[illegible]y shapes a[illegible]strated in Figure 3.8. This occurs as the additional layer will receive convex hulls as input from the first layer instead of single lines, thus by combining the convex hulls it causes arbitrary shapes to be formed allowing for more complex separations of classes to be made.



**Fi[illegible] 3.4.9 Arbitrary [illegible]ed from a combination of various convex regions**

The reason for neural network's popularity when combined with reinforcement learni[illegible] is their ability to generalise meaning, they are capable of classifying sets of inputs that are unknown to the network. In order to train a neural network a training algorithm is required. Various algorithms exist for training neural networks, with all possessing varying levels of ability and different attributes. In this project the TD algorithm and a relatively new class of algorithms called the residual algorithms will be used. The following chapter details the residual gradient algorithm and residual algorithm and also looks at direct TD with function approximation.

Artificial neural networks are a highly researched field and have been central to the development of AI. An artificial neural network is similar to the brain of a human and animal in that both consist of input neurons which gather information from an external environment and output neurons that produce patterns which form actions on the external environment. This chapter has covered a brief background on neural networks including early models of networks and whether they succeeded or failed. The basis of the artificial neuron which is the biological neuron was examined in detail. Finally, the multilayer perceptron was discussed including the backpropagation rule as they will be primary in implementation. Although other areas such as Hopfield networks are equally important in the field of AI they have not been discussed as they are not relevant to this thesis topic.

# Chapter 5 Residual Algorithm

The previous chapter focused on the basics on neural networks and the MLP network with backpropagation. In order to train neural networks we must first have a suitable algorithm to train it with. Various algorithms exist for training neural networks and have been successful in their application. However, when a function approximation system is used to handle a large number of states, it has been demonstrated that the direct application of function approximation techniques may result in the system not learning an optimal policy, causing the function approximator to diverge instead of converge.

Baird (1995) suggested an alternative class of reinforcement learning algorithms called residual algorithms. Baird used a linear function approximation system and found this new class of algorithms could learn a number of problems which are unable to be solved using a direct application of TD learning. This chapter focuses on Baird's work and examines the new class of algorithms - the residual gradient and the residual algorithm, along with direct TD with function approximation.

## 1.1 Direct TD with Function Approximation

Reinforcement learning algorithms, such as the temporal difference learning algorithm, have been developed and extensively used. These types of algorithms have been shown to successfully converge to an optimal solution with look-up tables. For example, an MDP has a finite number of states and each value of an earlier state *V*(*s*), is represented by a unique entry within the look-up table. This means during learning if every possible transition is experienced an infinite number of times and updated, thus ensuring convergence to an optimal solution with the learning rate α decaying to zero at a suitable rate. Equation 4.1 denotes a weight change and by definition is the exact TD(0) algorithm for any weight *w* used in a function approximation system (Baird, 1995).

$$\Delta w = \alpha\left(R + \gamma V(x') - V(x)\right)\frac{\partial V(x)}{\partial w}$$

**Equation 4.5.1 Formal view of TD(0) algorithm**

Equation 4.1 can also be referred to as the *direct* implementation of incremental value iteration, Q-learning and also *advantage learning*. Although TD has guaranteed convergence with look-up tables, the same results are not obtained when using function approximation systems. The TD algorithm has been shown to become unstable when directly implemented with a general function approximation system such as the linear function approximation system.

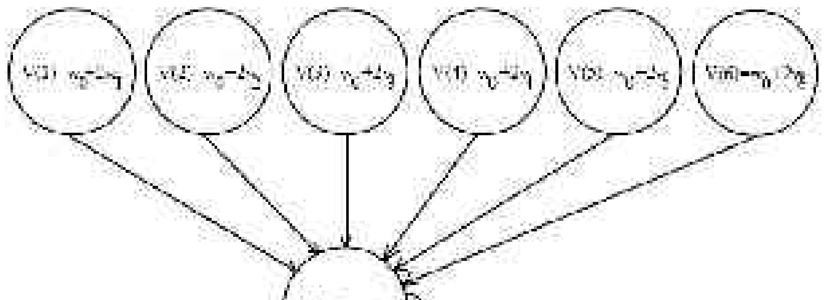


**Figure 4.5.1 The star problem (Baird, 1995)**

An example of TD becoming unstable can be found in the star problem illustrated in Figure 4.1. It shows six states with the value of each state given by a linear combination of two weights and each transition yields a reinforcement of zero. The function approximation system is basically a look-up table with an additional entry that allows for generalisation. The problem with this example is, if all the weights are initially positive and the value of state 7, *V*(7), is greater than the other values, then this will cause all of the values to grow without bound. This occurs due to the first five values being lower than the value of the successor γ *V*(7) with *V*(7) being even higher. This causes $w_0$ to increase five times for every time it is decreased which means it will rise quickly (Baird, 1995).

## *5.1 Residual Gradient Algorithm*

In the past reinforcement learning algorithms such as the direct algorithm discussed above, have been found to converge with quadratic function approximation systems (Baird, 1995). However, the direct algorithms are unable to converge for other types of functions approximation systems such as the linear function approximation system. Baird discovered that in order to find an algorithm with more stability than the direct algorithm, an exact goal should be specified for the learning system. An example of an exact goal could be for a problem on a deterministic Markov chain and with the goal of finding a value function for any state *x* and its successor state *x′* with a transition yielding an immediate reinforcement *R*.

$$E = \frac{1}{2}\sum_{x}\left[(R + \gamma V(x')) - V(x)\right]^2$$

**Equation 4.5.2 Residual Gradient algorithm**

The residual gradient algorithm solves the problem that occurs with direct algorithms by guaranteeing convergence; it works by performing a gradient descent on the mean squared Bellman residual. The Bellman residual is difference between the two sides of the Bellman equation for a particular value function *V* and state *x*. If the Bellman residual is not equal to zero then the final policy will be suboptimal. The amount to which the policy produces suboptimal reinforcement can be bounded for a certain degree of the Bellman residual. This indicates that it is possible to change the weights in the function approximation system by using gradient descent on the mean squared Bellman residual, *E*. An algorithm satisfying the latter could be referred to as the residual gradient algorithm.

The disadvantage of using the residual gradient algorithm is that it does not always learn as quickly as the direct algorithm. Applying the direct algorithm to the hall problem illustrated in Figure 4.2 would cause state 5 to converge to zero quickly due to information flowing from later to earlier states. This means the value of a successor state has no influence on the amount of time it takes the value function of state 5 to converge. If the residual gradient algorithm is applied to the star problem in Figure 4.1 then learning will be slow but it will eventually converge. For example if the initial values of $w_5 = 0$ and $w_4 = 20$, then the residual gradient algorithm would try to match their values by increasing the weight of 5 and decreasing the weight of 4. In comparison the direct algorithm would only decrease the value of 4 in order to match the weight of 5. This means information travels in two directions, with the information travelling faster in the right direction by a certain factor γ. Hence, if γ is close to the value of 1.0 then it is expected that the residual gradient algorithm will learn very slowly for the hall problem in Figure 4.2 (Baird, 1995).

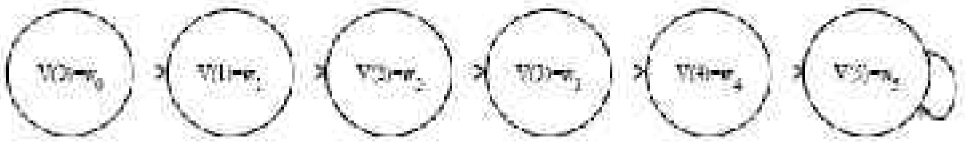


**Figure 4.5.2 The hall problem (Baird, 1995)**

## *5.2 Residual Algorithm*

Baird (1995) defined a residual algorithm to be any algorithm which was in the form of Equation 4.3. The weight change of the residual algorithm must equal the weighted average of both the residual gradient weight change and direct weight change, which means the direct and residual gradient algorithms are types of the residual algorithm (Baird, 1995). The residual algorithm is a combination of the direct algorithm and the residual gradient algorithm. It has the guaranteed convergence of the residual gradient algorithm combined with the fast learning speed of direct algorithms.

$$\Delta W_r = (1-\phi)\, \Delta W_d + \phi\, \Delta W_{rg}$$

**Equation 4.5.3 Residual Algorithm**

The value phi must be chosen appropriately, value phi can be a constant and equal as close to zero as possible, without disrupting the weights. If phi is equal to one then convergence is guaranteed, however, if phi equals zero it will either learn quickly or not at all. Baird (1995) discovered this approach was not suitable for two reasons, firstly it required an extra parameter to be selected and secondly the best phi to use primarily was not always the most appropriate phi to use after the system had been learning for a period of time (Baird, 1995). Baird found the solution to this problem was to use the lowest possible phi between one and zero.

Figure 4.3 demonstrates weight change vectors for the direct, residual gradient and residual algorithms. An optimal approach is to have the weight change consistent with the weight change vector. Therefore the residual weight change vector must be as close as possible to the weight of the direct algorithm to learn quickly and also create an acute angle with the residual gradient weight change vector for it to remain stable.

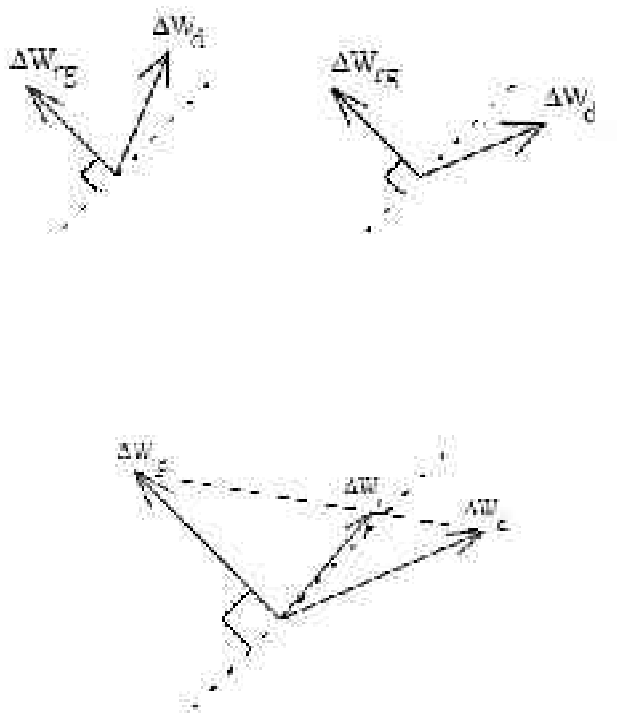


**Figure 4.5.3 Weight change vectors for the Direct, Residual Gradient and Residual algorithms**

Equation 4.4 shows the boolean expression for Figure 4.3. It guarantees a decreasing mean squared residual, at the same time conveying the weight change vector as close as possible to the direct algorithm. Ideally, the value must be greater than zero but less than one.

$$\Delta W_r \cdot \Delta W_{rg} > 0$$

**Equation 4.5.4 Boolean expression of the dot product**

If the dot product is positive this means the angle between the vectors is acute and the result of the weight change will be a decrease in mean squared Bellman residual, *E*. If the dot product is equal to zero then the residual and residual gradient weight change vectors must be orthogonal which means a small constant ε must be added to phi in order to convert the right angle into an acute angle. The constant ε stands for epsilon and is the amount of greediness used. The dot product of the numerator is divided by two separate dot products which are the denominator to find the value of phi in Equation 4.5.

$$\Delta W_r \cdot \Delta W_{rg} = 0$$

$$((1 - \phi)\, \Delta W_d + \phi\ \Delta W_{rg}) \cdot \Delta W_{rg} = 0$$

$$\phi = \frac{\Delta W_d \cdot \Delta W_{rg}}{\Delta W_d \cdot \Delta W_{rg} - \Delta W_{rg} \cdot \Delta W_{rg}}$$

**Equation 4.5.5 Calculating the value of ϕ**

It is preferable to have Equation 4.5 to yield phi between zero and one as this indicates the mean squared Bellman residual is constant, hence any phi above that value will ensure convergence. Baird came to conclusion a residual algorithm must calculate the numerator and the denominator separately and verify that the denominator is equal to zero. Phi will be equal to zero if the denominator is zero. Thus if the denominator does not equal zero the algorithm should evaluate Equation 4.5, the constant ε should then be included as mentioned above if the angle is not acute and check whether the resulting is between zero and one. The appropriate value of phi is important as it indicates whether Equation 4.3 is likely to converge (Baird, 1995).

This chapter has discussed the residual algorithms and direct TD with function approximation in detail. These algorithms were used to train a MLP network with the aim that the residual algorithm would perform better. The two algorithms were tested on four standard reinforcement learning problems, in order to determine which algorithm would perform better at training a neural network. The implementation of these algorithms and the neural network are discussed in the following chapter.

# Chapter 6 Methodology

The previous sections provided a background and brief introduction into the field of neural networks and reinforcement learning. Earlier work done by Baird was also presented. The purpose was to illustrate methods available in RL and which methods will be used in this project.

As mentioned earlier in the introduction, the aim of this thesis was to develop an improved learning system in the field of AI through expanding on work completed by Baird. In order to accomplish this, two algorithms were used: Baird’s residual algorithm and the more commonly known Temporal Difference Learning algorithm. The algorithms’ performances were compared by testing them on standard RL problems to demonstrate which algorithm was superior when training a neural network.

The implementation of the neural network used and the algorithms that were applied to the MLP system will be discussed further in this chapter. The updating traces method will be described in detail along with the reason for using the various values of lambda and alpha. Other factors that were considered such as the amount of hidden nodes and greediness used will also be mentioned and as well as the alterations applied to MLP_Sarsa.

## *6.1 Algorithms*

In the last chapter three different algorithms were discussed - these were the direct TD with function approximation, residual gradient and residual algorithms. TD is the principle algorithm used in RL as it has been successful and extensively used in numerous reinforcement learning tasks. Therefore it seemed an appropriate choice to compare against the new class of residual algorithms using function approximation systems, particularly as these were designed to overcome shortcomings in the direct application of TD.

Direct algorithms such as TD attempt to make each state match its successors but ignore the effects of generalization during learning. For example, at time t+1 the state tries to match the successor but at the same time the successor state moves up by the same amount hence the states never match. Direct algorithms are fast at learning when used with look-up tables and will converge to an optimal solution. However, when used in conjunction with function approximation systems direct algorithms have a tendency to become unstable and thus diverge instead of converge. The direct TD(0) algorithm is given formally in Equation 5.1.

$$\Delta w = \alpha\left(R + \gamma V(x') - V(x)\right)\frac{\partial V(x)}{\partial w}$$

**Equation 5.6.1 TD(0) algorithm**

However, the residual gradient algorithm attempts to make each state match both its predecessors and successors meaning at time t+1 the difference is calculated between the two states thus the states equal one another. The residual gradient algorithm also takes into the account of generalisation and has guaranteed convergence for function approximation systems but has the disadvantage of learning slowly in some cases. The residual gradient algorithm was implemented to allow for the current gradients to be found of each, in order to calculate the value of phi which in turn is used in the residual algorithm. Equation 5.2 shows the residual gradient algorithm.

$$E = \frac{1}{n}\sum_{i}\left[\left(R - \gamma V(x')\right) - V(x)\right]^2$$

**Equation 5.6.2 Residual Gradient algorithm**

The residual algorithm was implemented using Equation 5.1 below which appears in Baird's paper. The residual algorithm was developed to overcome the slow learning problem of the residual gradient algorithm and the divergence problem of the direct TD algorithm. Hence, it has the guaranteed convergence of the residual gradient algorithm and retains the fast learning speed of direct algorithm. The direct and residual gradient algorithms are classified as special cases of residual algorithms.

$$\Delta W_r = (1-\phi)\,\Delta W_d + \phi\,\Delta W_{rg}$$

**Equation 5.6.3 Residual algorithm**

## *6.2 Neural Network Implementation*

The neural network used is the Multilayer Perceptron with backpropagation. This type of network is more complicated than the linear function approximation system Baird used. The MLP has also been successful in some cases when applied with TD in previous experiments; this makes it an ideal choice for testing it with the residual algorithm, as it is pointless to test a new class of algorithms on a highly complex system when it has only been tested on a simple linear system. The MLP network is able to classify more complex patterns compared to a linear function approximation which is limited to only being able to classify simple linear patterns.

At the beginning of each trial the weights are initialised to new random values and the eligibility traces are cleared. The hidden layer uses a symmetric sigmoid activation function which is applied directly after the weighted sum of inputs to calculate the hidden layer activation. The method used to train the network was the Sarsa(λ) control method, the primary reason for choosing Sarsa(λ) was due to it being an on-policy algorithm.

This means it should improve the policy's learning considerably and strengthen actions of a sequence, due to an on-policy algorithm improving a policy gradually based on approximate values of the current policy.

## 6.3 Updating of Eligibility Traces

New arrays were created for the updateTraces() method in Figure 5.1 called outputGrad[] and resOutGrad[] these arrays were the current gradients for each algorithm and used for updating the traces. They consisted of the number of output weights used. The hidden gradients for the direct algorithm (hidGrad[][]) and the residual gradient algorithm (hiddenResGrad[][]) also needed to be created.

```
int o = 0;
if (o==outputID) // then update the traces, based on current gradients
{
        for (int h=0; h<numOutputWeights; h++)
        {
                outputGrad[h] = hidden[h];
                resOutGrad[h] = outputGrad[h] - lastOutputGrad[h];
                outputTraces[h] = resOutGrad[h] + outputTraces[h] * lambda;
                lastOutputGrad[h] = outputGrad[h];
        }


        for (int h=0; h<numHidden; h++)
        {
                for (int i=0; i<numHiddenWeights; i++)
                {
                        hidGrad[h][i] = (outputWeights[h]) /Math.abs(outputWeights[h]) * (0.25f
                        - hidden[h] * hidden[h]) * input[i];
                        hiddenResGrad[h][i] = hidGrad[h][i] - lastHidGrad[h][i];
                        hiddenTraces[h][i] = hiddenResGrad[h][i] + hiddenTraces[h][i] * lambda;
                        lastHidGrad[h][i] = hidGrad[h][i];
                }
        }
}
```

**Figure 5.6.1 Update traces**

The output trace that corresponds to the most recently selected action is generally the only trace to be updated. This trace is identified by the output ID. All existing hidden and output traces are decayed as shown below in Figure 5.2.

```
else // otherwise just decay the traces
{
        for (int h=0; h<numOutputWeights; h++)
        {
                outputTraces[h] = outputTraces[h] * lambda;
        }
                for (int h=0; h<numHidden; h++)
                {
                        for (int i=0; i<numHiddenWeights; i++)
                        {
                                hiddenTraces[h][i] = hiddenTraces[h][i]* lambda;
                        }
                }
}
```

**Figure 5.6.2 Decaying traces**

Two separate dot products need to be calculated for the direct and residual gradient algorithms to be used in Equation 5.1 to calculate the value of phi. This means both the hidGrad[][] and outGrad[] arrays of the direct algorithm and the hiddenResGrad[][] and resGrad[] arrays of the residual gradient algorithm need to be merged. This forms two new separate arrays for the direct and residual gradient algorithms, these were called dirGrad[] and resGrad[].

Both arrays have the same size which allowed a new variable to be created called resDirSize. It holds the weights for the dirGrad[] and resGrad[] arrays. This variable is used when forming the two dot products, a for loop is used to pass through every value in resDirSize which enables every value in dirGrad[] to be multiplied by a corresponding value in resGrad[] thus creating the dot products.

```
//for every value in dirGrad array times by corresponding value in resGrad array to
find angle
for(int e=0; e<resDirSize; e++)
{
        dot_dr += (dirGrad[e] * resGrad[e]);
        dot_rr += (resGrad[e] * resGrad[e]);
}

float denom = dot_dr - dot_rr;
if (denom==0.0f)
{
        phi = 0.0f;
}
else
{
        phi = dot_dr / denom + 0.001f; // from equation 11 plus the small positive
        constant mentioned in Baird's text

        if (phi<0.0f || phi > 1.0f)
        {
                phi = 0.0f;
        }
}

        for(int h = 0; h<resDirSize; h++)
        {
                residual[h] = (1-phi)*dirGrad[h] + phi*resGrad[h]; //residual algorithm
        }
```

**Figure 5.6.3 Calculating phi and adding it to the residual algorithm**

The two dot products, dot_dd and dot_rr are applied in Equation 5.2 to find the value of phi. The equation also checks for a zero denominator, if the denominator is zero then phi is set to zero. As illustrated in Figure 5.3 the residual gradient dot product is divided by the subtraction of the direct dot product from the residual gradient dot product, a small positive constant ε with the value of 0.001 is added to the Equation to convert a right angle to an acute angle.

$$\Delta W_r \cdot \Delta W_{rg} = 0$$

$$((1 - \phi)\, \Delta W_d + \phi\, \Delta W_{rg}) \cdot \Delta W_{rg} = 0$$

$$\phi = \frac{\Delta W_d \cdot \Delta W_{rg}}{\Delta W_d \cdot \Delta W_{rg} - \Delta W_{rg} \cdot \Delta W_{rg}}$$

**Equation 5.6.4 Calculating value of phi**

The value of phi is evaluated to see whether it is less than 0 or greater than 1, if this evaluates to true, phi is set to zero. Once a suitable value of phi has been found i.e. between 0 and 1, phi will be added to Equation 5.1 to calculate the residual algorithm. If the residual algorithm holds then the hidden and output traces will be updated using the result of the residual algorithm times by the value of lambda.

## *6.4 ε – Greedy Selection*

The ε-greedy method was used for all experiments, this was to allow for exploratory moves in addition to exploitation of paths by performing both exploratory and exploited actions the agent will achieve a more optimal result. A small amount of non - greediness equalling 0.2 was used in all the experiments. This means the majority of actions taken by the agent were exploited with the occasional action being exploratory.

## 6.5 Problem Environments

The benchmark problems used to test the algorithms against each other were Gridworld, Mountain Car, Acrobot and Puddleworld. Gridworld is the simplest problem out of the four. Gridworld is made up of a 10x7 grid with the agent beginning in a starting square on the grid; the aim is for the agent to reach the goal square in which case it receives a reward of 1.

The rewards' the agent receives gives an indication of the agent's situation in relation to the goal square hence if the agent becomes stuck in a corner, it would receive a reward of zero. The agent can move up, down, right and left.

Mountain Car and Acrobot are slightly more challenging. In Mountain Car, illustrated in Figure 5.4, the car must get enough momentum in order to reach the top of the mountain on the right, to obtain a positive reward. The reward is -1 for all time steps until the car has reached the goal on top of the mountain. The difficulty of the problem is that gravity is stronger than the car's engine, even with the car in full throttle it still can not make it up the mountain. Hence, the only solution is for the car to first move away from the goal, up the opposite mountain on the left.

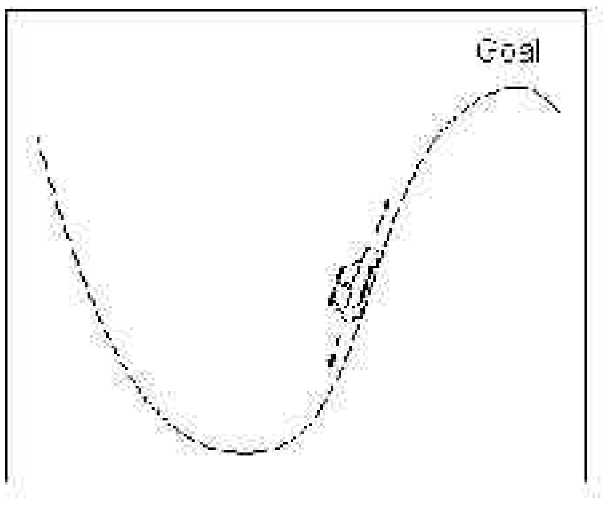


**Figure 5.6.4 Mountain Car Problem**

Acrobot consists of two rods that are attached in the centre with the whole system having four continuos state variables which are two joint positions and velocities. The goal is for the two rods to gain enough momentum to swing to 180 degrees getting the tip of the second rod over a line as illustrated below in Figure 5.5.

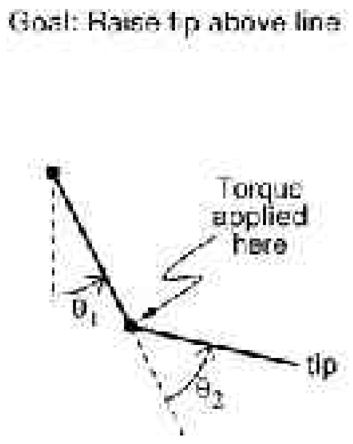


**Figure 5.6.5 The Acrobot**

The last problem is Puddleworld, the aim is for the agent is to reach the goal without getting stuck in a puddle. The agent can move left, right, up and down, and checks if it is in or near a puddle by measuring the distance it is from each puddle, each time it moves. There are two puddles in the world and if the agent lands in a puddle a penalty is given. A 2-D continuous gridworld with puddles is illustrated in Figure 5.6, the further the agent goes into a puddle then the higher the penalty will be. This is shown in Figure 5.6 where the darker areas in the centre of the puddle indicate the centre of the puddle, therefore higher penalty.

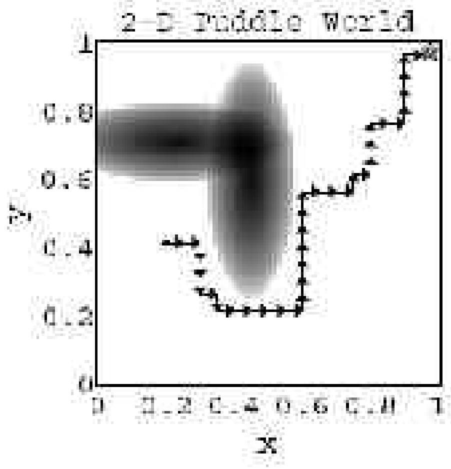


**Figure 5.6.6 2-D Puddleworld**

These different environments were chosen as they are recognised as standard problems in the reinforcement learning area.

Lambda was varied from 0.5 to 1 with 0.1 increments. For each value of lambda twenty trials were ran for each of the six learning rates α. The example in Figure 5.7 illustrates the varying values of lambda and alpha used in the experiments. Figure 5.7 also shows an example result of 1 for 0.5 lambda with 0.0001 alpha which is the average over 20 trials. In order to find a suitable set of parameters for each algorithm, varied values for alpha and lambda were used to provide consist and standard set of results.

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| Lambda | 0 | | | | | | |
| | 0.1 | | | | | | |
| | 0.2 | | | | | | |
| | 0.3 | | | | | | |
| | 0.4 | | | | | | |
| | 0.5 | 1 | | | | | |
| | 0.6 | | | | | | |
| | 0.7 | | | | | | |
| | 0.8 | | | | | | |
| | 0.9 | | | | | | |
| | 1 | | | | | | |

**Figure 5.6.7 Example of results table**

The number of hidden nodes used throughout all experiments was 12. The reason for using 12 hidden nodes for all the experiments was due to the fact that when similar previous experiments have been trialled using 12 hidden nodes, (Vamplew and Ollington, 2005) they outperformed other experiments using alternative values of hidden nodes.

## 6.6 MLP_Sarsa

MLP_Sarsa is the driver class that was used for training the Multilayer Perceptron (MLP) through reinforcement learning. It uses multiple MLP's but only one per action to implement Sarsa(λ) algorithm was employed to alter the values of lambda and alpha during experiments and to also change between the different environments. Two temp arrays were created to allow for the monitoring of the last 50 rewards. This was done to enable a comparison to be made between the last 50 rewards and the preceding rewards, in an attempt to measure the merit of the final policy learnt by the system.

This was achieved by first checking if numEpisodes was less than maxEpisodes, if this was true and numEpisodes greater than maxEpisodes – 50 held true which means the episodes were entering the last 50 rewards then place the last 50 rewards in trialReward[1] and the preceding rewards in trialReward[0].

The chapter has described in detail the implementation of the neural network and the updateTraces() method as well as the algorithms used in order to achieve the aim outlined in the introduction. The following chapter will discuss the results obtained from carrying out the experiments outlined in the chapter and also indicate which algorithm is more superior.

# Chapter 7 Results

The original purpose of this project was to investigate whether the residual algorithm would outperform direct application of the TD algorithm when training a neural network on reinforcement learning tasks. The previous chapter discussed the implementation of the MLP network and algorithms and also how the eligibility traces were updated. The parameter values were also mentioned, stating the reasons behind choosing them.

Four benchmark problems were run for each algorithm to allow them to be compared to one another, to examine which algorithm outperformed the other on each experiment. Another reason was to compare the results to optimal policies achieved previously in other research, to see how they performed in relation to the optimal policy that was received from conducting these experiments. The quality of the final policy was also an important factor in comparing the algorithms performances. To achieve this, a variable called the last 50 rewards was created, this allows the last 50 rewards to be examined for each algorithm to determine whether the agent learnt a sub-optimal or near-optimal policy.

This chapter focuses on the results obtained from the experiments and indicates which algorithm performed better with each problem; this is given by the amount of reward received for each trial. The algorithms performances are depicted by graphs with the amount of reward received plotted against the value of lambda. The last 50 rewards were gauged indicating the quality of each algorithms final policy. It was predicted that the residual algorithm would perform better and hence produce a more favourable final policy.

## 7.2 Gridworld

The following tables illustrate values in each square; these values are an average of the rewards over 20 trials of the Gridworld problem, which is a deterministic environment, meaning the environment is predictable rather than random. For every value of lambda there were six corresponding learning rates, the values of lambda were between 0.5 and 1 inclusive. The highest reward received during the average rewards was 0.704 shown below in Table 6.1; this result may not seem overly high since the maximum reward is 1. However, over a period of 20 trials, many instances occurred when the agent learnt near perfect (0.98) but within the 20 trials there were also a few trials where the agent learnt nothing (0). This means the average of all the trials is brought down. If agent learnt, it generally learnt well thus, if the agent only learnt a small amount, then the majority of the time it learnt nothing.

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | 0.358 | 0.420 | 0.467 | 0.465 | 0.462 | 0.380 |
| | 0.6 | 0.356 | 0.440 | 0.466 | 0.506 | 0.498 | 0.488 |
| | 0.7 | 0.354 | 0.468 | 0.522 | 0.544 | 0.597 | 0.571 |
| | 0.8 | 0.352 | 0.505 | 0.521 | 0.596 | 0.666 | 0.582 |
| | 0.9 | 0.342 | 0.525 | 0.593 | 0.704 | 0.701 | 0.501 |
| | 1 | 0.355 | 0.656 | 0.639 | 0.033 | 0.006 | 0.001 |

**Table 6.7.1 Average Residual Rewards for Gridworld**

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | 0.527 | 0.470 | 0.559 | 0.453 | 0.529 | 0.13 |
| | 0.6 | 0.460 | 0.531 | 0.554 | 0.468 | 0.568 | 0.227 |
| | 0.7 | 0.458 | 0.539 | 0.616 | 0.590 | 0.714 | 0.259 |
| | 0.8 | 0.459 | 0.591 | 0.582 | 0.648 | 0.751 | 0.489 |
| | 0.9 | 0.435 | 0.585 | 0.662 | 0.779 | 0.773 | 0.449 |
| | 1 | 0.497 | 0.838 | 0.816 | 0.702 | 0.000 | 0.000 |

**Table 6.7.2 Average Residual Last 50 Rewards for Gridworld**

Table 6.2 illustrates the average of last 50 rewards. The last 50 rewards was used a measurement to indicate the quality of learning to keep it separate from the actual learning, which would be biased towards the quicker learning algorithms (TD algorithm) as they would have a lower overall mean, whereas the slower learning algorithms (residual algorithm) would have a higher mean overall. This means that although the TD algorithm will learn quickly to begin with and converge, it will most likely only converge to a sub-optimal policy. The residual algorithm however, may learn slowly but it learns a more accurate policy, hence the last 50 rewards for the residual algorithm should indicate a near-optimal policy has been learnt. Therefore, the residual algorithm is more superior in performance than the TD algorithm. It is important to see how the agent's learning is affecting the average of the rewards over the 20 trials. For each trial the agent may have learnt quickly at the beginning producing a high reward but then learnt badly towards the end of the trial. A more ideal approach would be for the agent to learn poorly at the beginning and learn from its mistakes thus improving in its rewards towards the end of a trial.

Using the residual algorithm, the agent learnt more in the last 50 rewards, this is evident when the average rewards in Table 6.1 are compared with the last 50 rewards in Table 6.2. It is clear from these figures that the majority of the time for the residual algorithm the agent learnt a near-optimal policy, as the last 50 rewards were always an improvement on the average reward. This means it took longer for the agent to learn hence, producing a more accurate policy. For example, when the learning rate is 0.0005 and lambda is 1 the average of the last 50 rewards is equal to 0.838 which is a considerable improvement on 0.656 for the average rewards for the residual algorithm.

**Table 6.7.3 Average Temporal Difference Rewards for Gridworld**

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | 0.146 | 0.140 | 0.148 | 0.110 | 0.105 | 0.137 |
| | 0.6 | 0.143 | 0.138 | 0.147 | 0.103 | 0.106 | 0.165 |
| | 0.7 | 0.142 | 0.137 | 0.146 | 0.107 | 0.109 | 0.171 |
| | 0.8 | 0.141 | 0.137 | 0.144 | 0.107 | 0.125 | 0.186 |
| | 0.9 | 0.141 | 0.137 | 0.143 | 0.117 | 0.138 | 0.195 |
| | 1 | 0.173 | 0.173 | 0.181 | 0.187 | 0.214 | 0.170 |

| | Alpha | | | | | |
|---|---|---|---|---|---|---|
| Lambda | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| 0.5 | 0.120 | 0.120 | 0.165 | 0.118 | 0.094 | 0.158 |
| 0.6 | 0.120 | 0.122 | 0.163 | 0.102 | 0.125 | 0.215 |
| 0.7 | 0.120 | 0.120 | 0.164 | 0.141 | 0.127 | 0.195 |
| 0.8 | 0.120 | 0.120 | 0.155 | 0.138 | 0.130 | 0.198 |
| 0.9 | 0.120 | 0.120 | 0.150 | 0.159 | 0.159 | 0.130 |
| 1 | 0.156 | 0.157 | 0.189 | 0.240 | 0.207 | 0.171 |

**Table 6.7.4 Average Temporal Difference Last 50 Rewards for Gridworld**

Tables 6.3 and 6.4 are the results achieved for the Temporal Difference algorithm when applied to Gridworld. As the tables show the TD algorithm performed substantially worse than the residual algorithm, the highest reward obtained overall was 0.240 in the last 50 rewards. Where the residual algorithm achieved a reward of 0.838 for learning rate of 0.0005 and lambda 0.9, the TD algorithm only managed a reward of 0.157. Overall the temporal difference algorithm doesn't appear to have learnt much at all, as the rewards are so low. This indicates that the agent did not come very close to reaching the goal.

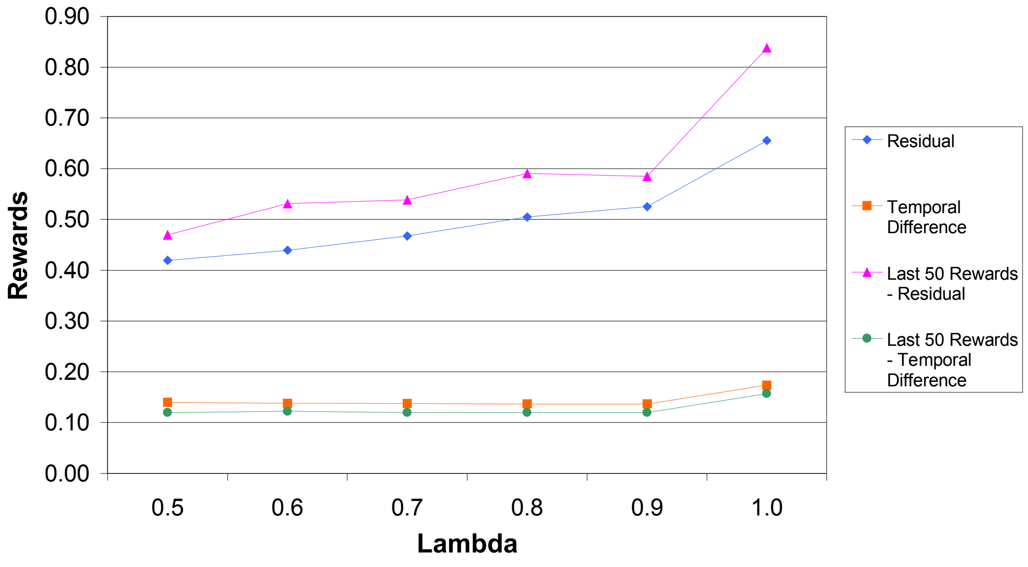


**Figure 6.7.1 Gridworld - Residual vs. Temporal Difference**

Figure 6.1 illustrates the performance of the residual algorithm against the TD algorithm. The values plotted on the graph include the best value of alpha for that algorithm. As lambda approached 1 both the residual and TD algorithms improved in performance. The higher rewards obtained when lambda is equal to 1, is possibly due to the amount of randomness within an environment. If an environment is fairly deterministic then the highest rewards will be achieved when lambda is close to 1 compared with an environment that is completely random then the agent will learn better when a lower value of lambda is used.

## *7.3 Mountain Car*

The Mountain Car problem is slightly more difficult compared to Gridworld and as a result the algorithms do not perform as well. Mountain Car is different to Gridworld, in that the rewards received were negative, the aim of Mountain Car is still to obtain the highest reward possible (-100 is higher than -900).

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -761 | -511 | -391 | -371 | -478 | -590 |
| | 0.6 | -746 | -467 | -373 | -415 | -498 | -602 |
| | 0.7 | -733 | -428 | -320 | -425 | -517 | -623 |
| | 0.8 | -676 | -356 | -296 | -449 | -574 | -654 |
| | 0.9 | -526 | -297 | -261 | -573 | -539 | -687 |
| | 1 | -451 | -439 | -404 | -288 | -287 | -286 |

**Table 6.7.5 Average Residual Rewards for Mountain Car**

| Lambda | Alpha 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
|---|---|---|---|---|---|---|
| 0.5 | -668 | -274 | -194 | -399 | -585 | -529 |
| 0.6 | -616 | -249 | -192 | -495 | -632 | -557 |
| 0.7 | -590 | -249 | -167 | -637 | -652 | -645 |
| 0.8 | -407 | -187 | -149 | -596 | -697 | -676 |
| 0.9 | -269 | -157 | -142 | -736 | -624 | -707 |
| 1 | -459 | -465 | -433 | -301 | -301 | -301 |

**Table 6.7.6 Average Residual Last 50 Rewards for Mountain Car**

| Lambda | Alpha 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
|---|---|---|---|---|---|---|
| 0.5 | -663 | -686 | -680 | -741 | -720 | -668 |
| 0.6 | -667 | -681 | -688 | -734 | -714 | -654 |
| 0.7 | -673 | -673 | -706 | -741 | -701 | -680 |
| 0.8 | -679 | -668 | -727 | -756 | -729 | -678 |
| 0.9 | -685 | -725 | -767 | -771 | -699 | -698 |
| 1 | -678 | -602 | -590 | -514 | -547 | -613 |

**Table 6.7.7 Average Temporal Difference Rewards for Mountain Car**

**Table 6.7.8 Average Temporal Difference Last 50 Rewards for Mountain Car**

| Lambda | Alpha 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
|---|---|---|---|---|---|---|
| 0.5 | -677 | -643 | -682 | -746 | -585 | -529 |
| 0.6 | -678 | -636 | -734 | -619 | -632 | -557 |
| 0.7 | -680 | -629 | -748 | -702 | -652 | -645 |
| 0.8 | -681 | -705 | -784 | -786 | -697 | -676 |
| 0.9 | -681 | -823 | -786 | -535 | -624 | -707 |
| 1 | -728 | -614 | -614 | -528 | -301 | -301 |

Illustrated below in Figure 6.2 is a graph of the residual algorithm plotted against the Temporal Difference algorithm. The graph shows that the residual algorithm overall learnt better than the Temporal Difference algorithm by approximately a factor of 3. The last 50 rewards of the Temporal Difference algorithm were slightly poorer than the average rewards. This could be due to the TD algorithm diverging instead of converging. Since the TD algorithm learns so well to begin with, the weights may then diverge causing the agents performance of the last 50 episodes to be poor.

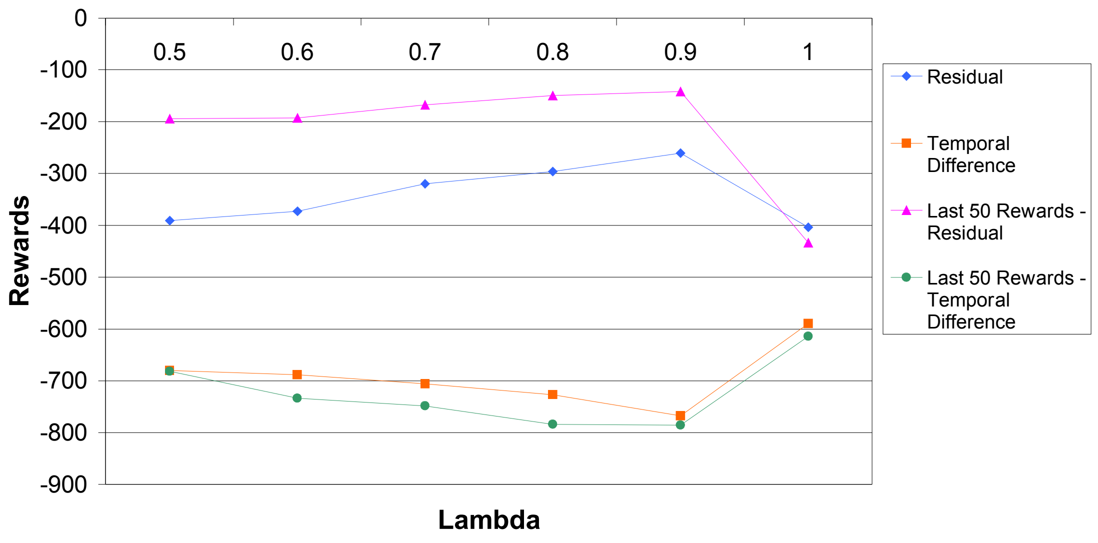


**Figure 6.7.2 Mountain Car - Residual vs. Temporal Difference**

Figure 6.2 shows that as lambda approached 1 using the TD algorithm it improved slightly, compared with the residual algorithm which declined as lambda approached 1, especially in the last 50 rewards in which case the average reward was better. Unlike the previous experiment Gridworld, where the residual algorithm improved considerably from lambda 0.9 to 1, the residual algorithm instead gradually improved and peaked as lambda reached 0.9 then steeply declined as lambda equaled 1. As lambda performs well at 0.9 lambda and poor at lambda 1, it was considered that a better reward would be achieved if lambda were equal to 0.95.

| **Residual** | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | Alpha | | | | | |
| | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| Lambda | 0.95 | -356 | -214 | -212 | -525 | -539 | -663 |
| **Last 50 Rewards** | | | | | | | |
| | | Alpha | | | | | |
| | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| Lambda | 0.95 | -182 | -134 | -200 | -650 | -608 | -723 |

**Table 6.7.9 Residual Rewards using 0.95 Lambda for Mountain Car**

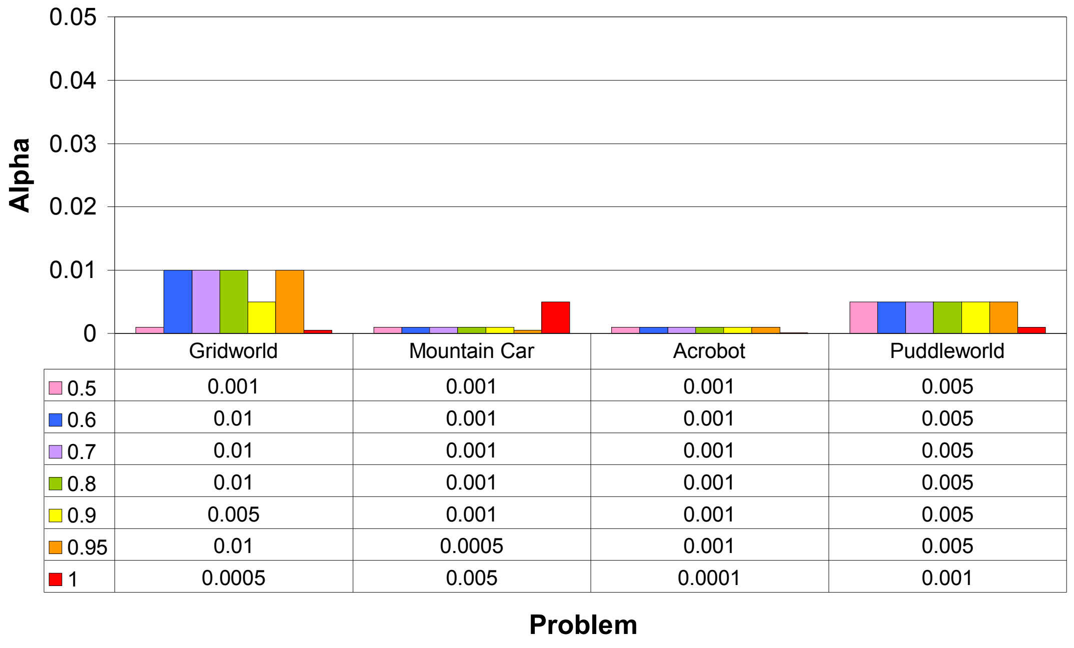
| **Temporal Difference** | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | Alpha | | | | | |
| | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| Lambda | 0.95 | -647 | -743 | -766 | -770 | -611 | -615 |
| **Last 50 Rewards** | | | | | | | |
| | | Alpha | | | | | |
| | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| Lambda | 0.95 | -649 | -805 | -827 | -703 | -457 | -597 |

**Table 6.7.10 Temporal Difference Rewards using 0.95 Lambda for Mountain Car**

As the results show in Table 6.9 the average rewards did improve for the residual algorithm when lambda is at 0.95 but only for smaller learning rates (below 0.005). The last three average rewards for lambda 0.95 for the residual algorithm are better than when lambda is 0.9 but they are worse than when lambda is 1. The same occurred for the residual algorithms last 50 rewards in Table 6.9. The highest rewards received when lambda is 0.95 for both the averages for the residual algorithm are the highest rewards received when lambda is 0.95.

Again the rewards received for the TD algorithm in Table 6.10 when lambda is 0.95 were poor. It is only when the learning rate is equal to 0.0001 that the rewards achieved were higher in both the average rewards and the last 50 rewards, the remaining rewards were all higher.

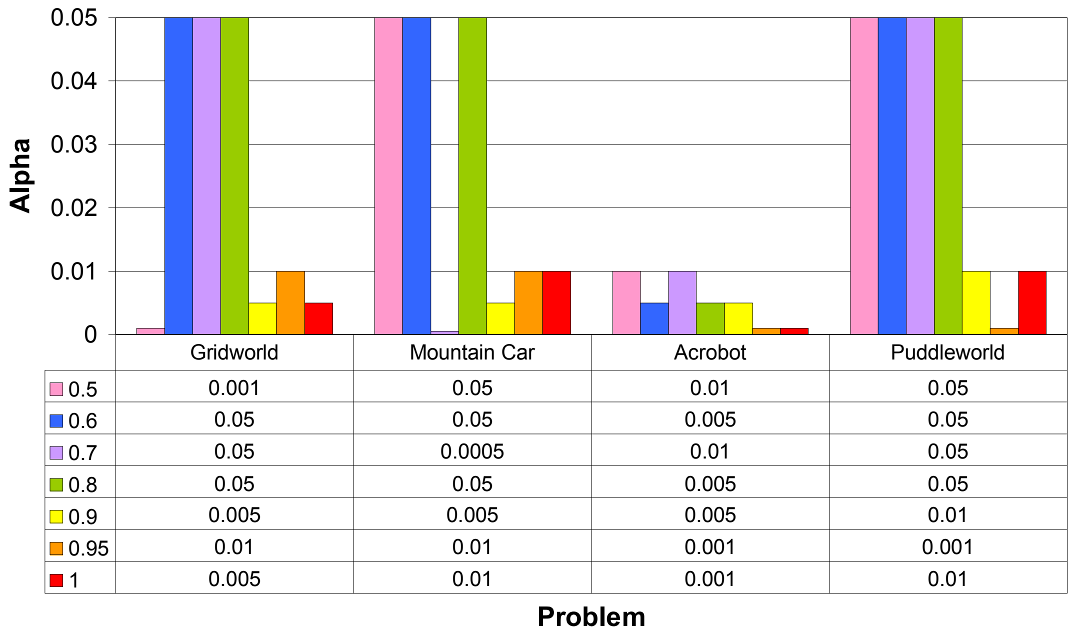
In order to gain a better understanding of the relationship between values of lambda and alpha, for example, does a higher lambda yield better results with higher or lower value of alpha? A graph was made showing for each problem, the best alpha value for each lambda value for both algorithms.



| | Gridworld | Mountain Car | Acrobot | Puddleworld |
|---|---|---|---|---|
| ■ 0.5 | 0.001 | 0.001 | 0.001 | 0.005 |
| ■ 0.6 | 0.01 | 0.001 | 0.001 | 0.005 |
| ■ 0.7 | 0.01 | 0.001 | 0.001 | 0.005 |
| ■ 0.8 | 0.01 | 0.001 | 0.001 | 0.005 |
| ■ 0.9 | 0.005 | 0.001 | 0.001 | 0.005 |
| ■ 0.95 | 0.01 | 0.0005 | 0.001 | 0.005 |
| ■ 1 | 0.0005 | 0.005 | 0.0001 | 0.001 |

**Figure 6.7.3 Indicates the Best Alpha Value for Each Value of Lambda for Residual Algorithm**

Figure 6.3 shows a graph of each problem with the best alpha value obtained for each lambda value for the residual algorithm. As the graph illustrates a high lambda value is more likely to achieve good results with lower alpha rates using the Mountain Car and Acrobot problems. Gridworld obtained the best results with higher alpha values and mid-range lambda values. Puddleworld worked well with all values of lambda using an alpha value of 0.005, except for lambda 1 which obtained the best result using a slightly lower alpha value of 0.001. Overall the best results are achieved when alpha is equal to 0.001 for any lambda value.



| | Gridworld | Mountain Car | Acrobot | Puddleworld |
|---|---|---|---|---|
| ■ 0.5 | 0.001 | 0.05 | 0.01 | 0.05 |
| 🟦 0.6 | 0.05 | 0.05 | 0.005 | 0.05 |
| 🟪 0.7 | 0.05 | 0.0005 | 0.01 | 0.05 |
| 🟩 0.8 | 0.05 | 0.05 | 0.005 | 0.05 |
| 🟨 0.9 | 0.005 | 0.005 | 0.005 | 0.01 |
| 🟧 0.95 | 0.01 | 0.01 | 0.001 | 0.001 |
| 🟥 1 | 0.005 | 0.01 | 0.001 | 0.01 |

**Figure 6.7.4 Indicates the Best Alpha Value for Each Lambda Value for Temporal Difference Algorithm**

Figure 6.4 also illustrates a graph of each problem with the best alpha value for each corresponding lambda value but for the TD algorithm instead. Overall a higher value of lambda achieves better results with a higher value of alpha (above 0.001). As the graph shows, the TD algorithm is more likely to achieve good results with lower values of lambda and higher alpha values.

The optimal policy for Mountain Car is 88 steps (Dietterich and Wang, 2001). Using the residual algorithm the agent was able achieve 134 steps from the goal which means 46 steps from the optimal policy. This means the agent is actually 52% off the optimal policy so for every step the agent takes using the optimal policy the agent takes 1.52 steps using the policy obtained from these experiments.

## 7.4 *Acrobot*

Acrobot is more challenging than Mountain Car which could explain the lower rewards; there is also little change between the average rewards and last 50 rewards which suggests the agent is not learning much overall. For example, the reward received for lambda 0.9 and alpha 0.005 for the residual algorithm is -450 and the last 50 rewards for this lambda and alpha value is equal to -434.

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -682 | -621 | -652 | -590 | -641 | -768 |
| | 0.6 | -662 | -589 | -557 | -588 | -544 | -749 |
| | 0.7 | -631 | -580 | -578 | -656 | -592 | -727 |
| | 0.8 | -592 | -529 | -453 | -474 | -562 | -696 |
| | 0.9 | -630 | -558 | -502 | -450 | -567 | -720 |
| | 1 | -548 | -797 | -854 | -899 | -899 | -899 |

**Table 6.7.11 Average Residual Rewards for Acrobot**

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -671 | -572 | -561 | -565 | -729 | -780 |
| | 0.6 | -624 | -492 | -392 | -607 | -670 | -739 |
| | 0.7 | -566 | -470 | -461 | -667 | -772 | -689 |
| | 0.8 | -539 | -402 | -361 | -488 | -698 | -662 |
| | 0.9 | -607 | -427 | -402 | -434 | -560 | -682 |
| | 1 | -327 | -688 | -896 | -921 | -921 | -921 |

**Table 6.7.12 Average Residual Last 50 Rewards for Acrobot**

Out of all three environments Acrobot performed the worst. Both the residual and TD algorithms improved in learning once lambda was nearer to 1, as seen in Tables 6.11, 6.12 and 6.13. The optimal policy for Acrobot is about 30 steps which means when calculated the agent is 900.90% off the optimal policy or for every step an agent takes using the optimal policy the other agent takes 10.9 steps. This is using the best value achieved with the residual algorithm which is -327. For the TD algorithm the agent is 1510% off the optimal policy. This shows that overall the residual algorithm learnt the better policy.

| Lambda | Alpha 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
|---|---|---|---|---|---|---|
| 0.5 | -901 | -888 | -885 | -719 | -715 | -906 |
| 0.6 | -901 | -879 | -879 | -679 | -646 | -904 |
| 0.7 | -900 | -907 | -898 | -679 | -688 | -898 |
| 0.8 | -897 | -909 | -904 | -768 | -765 | -920 |
| 0.9 | -897 | -910 | -895 | -851 | -867 | -931 |
| 1 | -646 | -588 | -570 | -751 | -869 | -920 |

**Table 6.7.13 Average Temporal Difference Rewards for Acrobot**

| Lambda | Alpha 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
|---|---|---|---|---|---|---|
| 0.5 | -911 | -903 | -870 | -564 | -514 | -972 |
| 0.6 | -915 | -895 | -890 | -579 | -879 | -964 |
| 0.7 | -917 | -931 | -853 | -618 | -483 | -967 |
| 0.8 | -917 | -921 | -951 | -750 | -821 | -964 |
| 0.9 | -939 | -944 | -899 | -874 | -930 | -964 |
| 1 | -586 | -564 | -552 | -883 | -661 | -952 |

**Table 6.7.14 Average Temporal Difference Last 50 Rewards for Acrobot**

Generally the last 50 rewards the agent receives are slightly better than the average reward. For example, the average last 50 rewards for the residual algorithm in Table 6.11 with lambda value 1 and alpha equaling 0.0001 is obviously an improvement on the -548 this indicates clearly that the residual can take slightly longer to learn but does eventually converge to a more optimal solution than the TD algorithm.

Using a lambda value of 0.95 and an alpha value of 0.001, the residual algorithm managed the best reward of -261. The TD algorithm didn't improve with lambda at 0.95 instead it got slightly worse.

While TD also improves in the last 50 rewards this does not occur as often, as you can see if comparing Tables 6.13 and 6.14, many of the rewards are worse than the average rewards. This may be due to the TD algorithm converging quickly then diverging causing the last 50 rewards to be poorer.

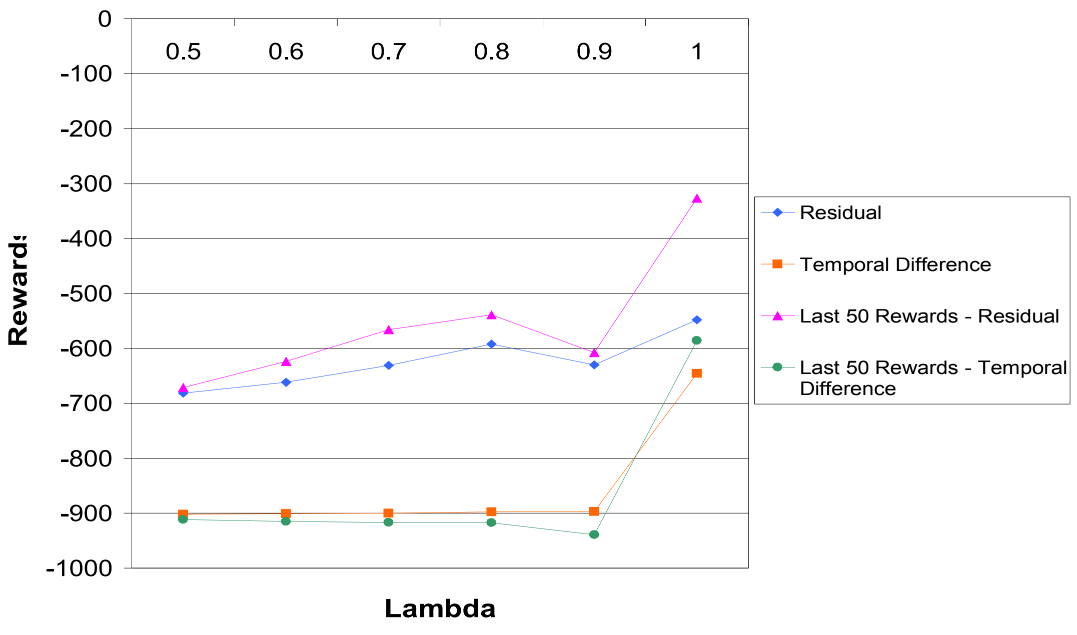


**Figure 6.7.5 Acrobot - Residual vs. Temporal Difference**

Figure 6.5 illustrates the residual algorithm against the TD algorithm, as illustrated on the graph with rewards gained are never very high. For the first time however, the last 50 rewards for the TD algorithm actually learn better than the average rewards. This suggests that towards the end of the trial when lambda is close to one the agent starts learning a more optimal policy. As this environment is fairly deterministic better rewards are achieved at lambda 1.

## 7.5 Puddleworld

Out of all four problems Puddleworld is the most challenging as the results illustrate, Puddleworld performed the worst especially the TD algorithm, with only one of its average rewards being above -1000 (-998). The residual algorithm however, had many rewards that were below -1000 (-1015). This means it is possibly harder for the agent to achieve an optimal solution thus resulting in the agent learning very little.

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -1007 | -864 | -614 | -302 | -266 | -1022 |
| | 0.6 | -1009 | -703 | -519 | -252 | -233 | -1020 |
| | 0.7 | -1010 | -648 | -415 | -232 | -214 | -1017 |
| | 0.8 | -1011 | -538 | -386 | -212 | -208 | -1008 |
| | 0.9 | -951 | -469 | -321 | -241 | -363 | -998 |
| | 1 | -1015 | -982 | -932 | -1003 | -1002 | -1036 |

**Table 6.7.15 Average Residual Rewards for Puddleworld**

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -1042 | -413 | -296 | -107 | -113 | -1028 |
| | 0.6 | -1038 | -386 | -257 | -118 | -121 | -1042 |
| | 0.7 | -1049 | -269 | -246 | -122 | -128 | -1035 |
| | 0.8 | -1024 | -300 | -211 | -119 | -126 | -1029 |
| | 0.9 | -733 | -283 | -204 | -186 | -427 | -1025 |
| | 1 | -1076 | -1021 | -984 | -1035 | -1034 | -1047 |

**Table 6.7.16 Average Residual Last 50 Rewards for Puddleworld**

As shown in Tables 6.15 and 6.16 the agent learnt a considerable amount using the residual algorithm. The rewards received for the highest and lowest learning rates were very low being in the -1000 range. The agent learnt well in between these learning rates, however, when lambda was 1, the reward became even less. The temporal difference algorithm performed the worst out of all the experiments by far, as the highest reward received was -998.

This clearly indicates that the agent did not learn a near-optimal policy, instead it is suggested that the agent continued to get stuck in one of the puddles moving further and further into the puddle thus receiving a greater penalty.

The optimal policy for Puddleworld is 39 steps (Dietterich and Wang, 2001). This means the agent is 39 steps from the goal. The results for these experiments suggest the agent was not near the optimal policy especially with reference to the TD algorithm where the best result was 998. When calculated the agent is actually 2458% off from optimal policy using the TD algorithm. Using the best result obtained from the residual algorithm which is 107, the agent is 174% off from the optimal policy. Another way to look at it is, for every step the agent takes towards the goal using the optimal policy; the agent using this policy takes 2.74 steps.

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -1096 | -1250 | -1240 | -1050 | -1038 | -1022 |
| | 0.6 | -1131 | -1274 | -1196 | -1078 | -1035 | -1020 |
| | 0.7 | -1270 | -1304 | -1200 | -1054 | -1023 | -1017 |
| | 0.8 | -1295 | -1285 | -1226 | -1045 | -1018 | -1008 |
| | 0.9 | -1419 | -1291 | -1233 | -1015 | -1006 | -998 |
| | 1 | -1412 | -1143 | -1102 | -1109 | -1087 | -1036 |

**Table 6.7.17 Average Temporal Difference Rewards for Puddleworld**

| | | Alpha | | | | | |
|---|---|---|---|---|---|---|---|
| Lambda | | 0.0001 | 0.0005 | 0.001 | 0.005 | 0.01 | 0.05 |
| | 0.5 | -1181 | -1362 | -1542 | -1153 | -1052 | -1028 |
| | 0.6 | -1286 | -1353 | -1346 | -1120 | -1046 | -1042 |
| | 0.7 | -1886 | -1522 | -1378 | -1085 | -1092 | -1035 |
| | 0.8 | -1879 | -1431 | -1326 | -1061 | -1058 | -1029 |
| | 0.9 | -1750 | -1415 | -1269 | -1057 | -1023 | -1025 |
| | 1 | -1768 | -1333 | -1068 | -1125 | -1045 | -1047 |

**Table 6.7.18 Average Temporal Difference Last 50 Rewards for Puddleworld**

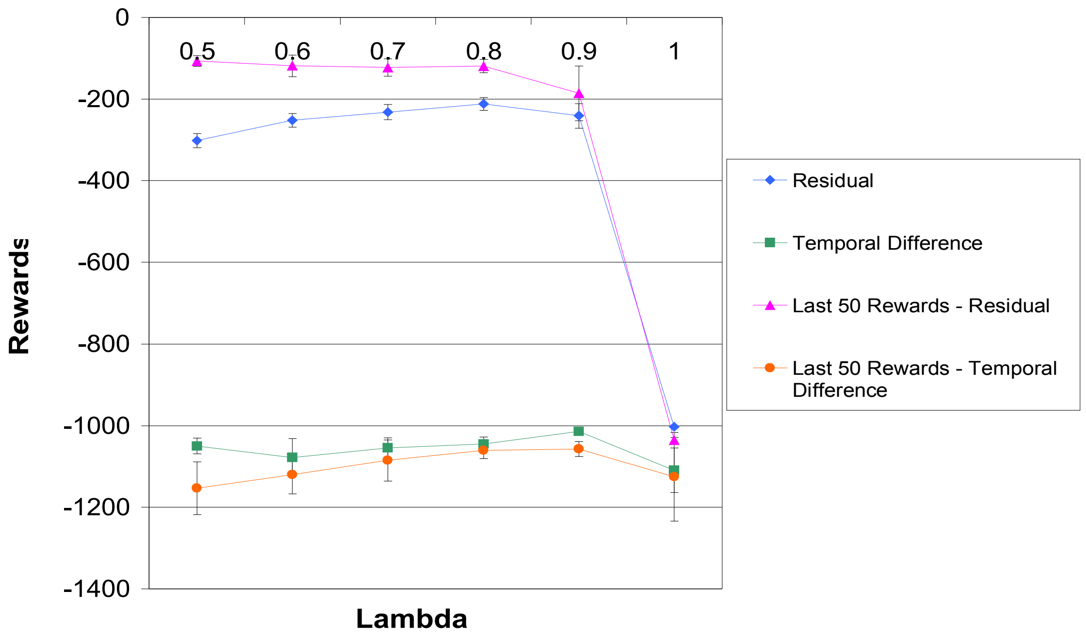


**Figure 6.7.6 Puddleworld - Residual vs. Temporal Difference**

Figure 6.4 illustrates the performance of the algorithms in the Puddleworld problem. The lines coming from each point on the graph are error bars, these indicate the amount of confidence that the resulting reward will be within a certain range. For example, there is a 95% confidence that the reward value for 0.8 lambda with a learning rate of 0.005 for the average residual rewards will be in 15.84 range of the reward -211. For certain points the range is so small it can not be seen e.g. there is a 95% confidence that the reward for lambda 1 with a learning rate of 0.005 for the average residual rewards will be within 1.3 range of the -1003 reward. This means there is 95% confidence the reward will always be between -1001.7 and -1004.3.

# Chapter 8 Conclusion

Over the past decade reinforcement learning has become increasing popular and well known by creating agents that are capable of learning a behaviour that is similar to that of a human with little or no knowledge of its surrounding environment. Although previous work has been successful in many applications of reinforcement learning using neural networks, there have also been some unsuccessful cases. When neural networks are used as a form of function approximation for large problem domains with reinforcement learning, it has been shown that the system may not learn a suitable policy when using direct application of function approximation techniques.

This thesis has investigated the performance of the residual algorithm against the well known TD learning algorithm when training a MLP using backpropagation on four different reinforcement learning problems. The aim was for the residual algorithm to outperform direct application of the TD learning algorithm in all problems.

The algorithms were tested on four problems these were Gridworld, Mountain Car, Acrobot and Puddleworld. Each problem environment was run for 20 trials for each algorithm and for each value of lambda from 0.5 to 1 inclusive there were six corresponding alpha values. The algorithms were compared in the rewards they achieved for each problem and the last 50 rewards were recorded for examining and comparing the quality of each policy achieved by each algorithm which was further compared to optimal policies achieved through previous research. The best alpha for each lambda value for each problem and algorithm were also examined to determine which combination of lambda and alpha produced the best results.

## 8.1 Discussion

In chapter 6 the results of all experiments were discussed. Section 6.1 discussed the results for Gridworld, it was shown that overall the residual algorithm outperformed the TD algorithm. Many instances occurred over a period of 20 trials when the agent learnt near perfect (0.98) but because there were also instances where the agent learnt near nothing (0). This brought the average down so the highest reward achieved for Gridworld was 0.838 for the residual algorithm. The TD algorithm performed much worse than the residual algorithm using the Gridworld environment. The highest reward the TD algorithm achieved was 0.241 with lambda at 0.95, thus indicating the agent did not learn a near optimal policy.

The last 50 rewards demonstrate if the agent has learnt an optimal policy. For Gridworld using the residual algorithm the agent appears to have learnt a near-optimal policy as the average rewards are poorer than the last 50 which means the agent took longer to learn but has learnt a more accurate policy.

Again for Mountain Car problem the residual algorithm performed better than the TD algorithm by approximately a factor of 3. For both algorithms the reward was better when lambda was close to the value of 1 (0.9 and above). In order to see if better results could be obtained, the algorithms were tested on lambda 0.95 for all values of alpha. In the majority of cases 0.95 lambda did provide a slightly better result. To compare the results even more the best value for alpha was taken for each value of lambda for each problem and algorithm. The graphs showed mixed results, using the residual algorithm

The results for Acrobot were average. By comparing the optimal policies for the two algorithms it showed that once again the residual algorithm converged to a better solution overall than the TD algorithm.

Puddleworld results achieved the best and worst in terms of rewards, the majority of the TD algorithms results were in the -1000 range. The best reward achieved by the TD algorithm was -998 which is far more than the residual algorithm's best reward of -107.

In all four experiments for the TD algorithm had lower last 50 rewards than the average rewards. This means that the TD algorithm could be diverging, as it learns well in the beginning, but the weights may be diverging meaning the last 50 rewards the agent performs quite poorly.

Overall the residual algorithm appeared to perform best on the Mountain Car problem as it was closet to the optimal policy. The TD algorithm performed a lot worse especially with Puddleworld with it being 2458% off the optimal policy compared with the residual algorithm which was only 174% off the optimal policy.

In conclusion the residual algorithm can clearly perform better at training a MLP neural network on this group of reinforcement learning tasks. Further work that could be done in order to test this theory even further is discussed below.

## *8.2 Future Work*

In order to extend and prove further that the residual algorithm is more superior to the well known TD algorithm, the residual algorithm will need to be tested on more challenging networks and problems.

The network used in all experiments was the MLP network. Although the MLP network was more complicated than the linear function approximation system Baird used, it is a fixed network hence it doesn't change. Therefore examining how the residual algorithm would perform on a constructive neural network would be more challenging. Examples of possible constructive neural networks would be Cascade and RAN. Both networks are constructive meaning they start with no hidden nodes and add one at a time whilst learning. Due to these types of networks learning iteratively, they may be able to recognise more complicated problems that a fixed network could not. Cascade has also been proven to converge faster than the MLP network.

The four problems used to compare the algorithms were fairly simple deterministic problems. A problem domain with a larger number of states and parameters would be the next step since the residual algorithm performed well on these simpler problems. It only seems logical to test its performance on an even harder set of problems, where a function approximation system is needed to a much greater extent.

If using a different network to MLP such as Cascade or RAN then it might be reasonable to try other parameters. The parameters that could be altered would be whether using a Cascade network would still perform best with 12 hidden nodes or would it perform better using a different number of nodes e.g. 30. Slightly better results could also be obtained using different learning rates or action selection, for example, if using a Cascade network the agent may perform better using softmax or ε-greedy selection. These changes in the parameters are not significant future work, just an extra suggestion or consideration.

# References

- Baird, L 1995, *'Residual Algorithms: Reinforcement Learning with Function Approximation',* Machine Learning: Proceedings of the Twelfth International Conference, Morgan Kaufman Publishers, San Francisco.
- Beale, R & Jackson, T 1990, *Neural Computing: An Introduction*, Institute of Physics Publishing, Bristol and Philadelphia.
- Hagan, M T., Demuth, H. B. and Beale, M 1996, *Neural Network Design*. USA, PWS Publishing Company.
- Hebb, DO 1949, *The Organization of Behaviour*, John Wiley & Sons, New York.
- Kaelbling, L. P., Littman, M. L. and Moore, A. W 1996, 'Reinforcement Learning: A Survey', *Journal of Artificial Intelligence Research*, Vol. 4, pp. 237-285.
- McClelland, JL & Rumelhart, DE 1986, *Parallel Distributed Processing*, MIT Press, Cambridge, MA.
- McCulloch, WS & Pitts, W 1943, 'A logical calculus of the ideas immanent in nervous activity', *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-33.
- Minsky, M & Papert, SA 1969, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA.
- Parker, DB 1985, *Learning logic*, 47, Center for Computational Research in Economics and Management Science, MIT.
- Randlov, J & Alstrom, P 1998, *'Learning to Drive a Bicycle using Reinforcement Learning and Shaping',* paper presented to Fifteenth International Conference in Machine Learning.
- Rosenblatt, F 1962, *Principles of Neurodynamics*, Spartan, New York.

- Rumelhart, DE, Hinton, GE & Williams, RJ 1986, 'Learning internal representations by error propagation', *Computational models of cognition and perception*, vol. 1, pp. 319-62.

- Rummery, GA & Niranjan, M 1994, *On-line Q-Learning Using Connectionist Systems*, CUED/F-INFENG/TR 166, Cambridge University Engineering Department, Cambridge.

- Sondak, N E. and Sondak, V K 1989, ‘Neural networks and artificial intelligence’, pp. 1-5.

- Sutton, R & Barto, A 1998, *Reinforcement Learning: An Introduction*, Cambridge, Massachusetts: A Bradford Book, The MIT Press.

- Tesauro, G 1995, *‘Temporal Difference Learning and TD-Gammon’*, *Communications of the ACM*, Vol. 38, No. 3.
- Thrun, S and Schwarz, A 1993, *Issues in using Function Approximation for Reinforcement Learning*, Lawrence Erlbaum Publisher, pp. 1-8.

- Vamplew, P & Ollington, R 2005, *Global versus Local Constructive Function Approximation for On-line Reinforcement Learning,* The Australian Joint Conference on Artificial Intelligence

- Werbos, PJ 1974, 'Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences', PhD thesis, Harvard University.

- Widrow, B & Hoff, ME 1960, 'Adaptive switching circuits', *IRE WESCON Convention Record*, pp. 96-104.