**A. Braithwaite**

# Reinforcement Learning Demolition Teams

Computer Science Tripos – Part II

Fitzwilliam College

May 12, 2016

# Proforma

| | |
|---|---|
| Name: | **Alex Braithwaite** |
| College: | **Fitzwilliam College** |
| Project Title: | **Reinforcement Learning Demolition Teams** |
| Examination: | **Computer Science Tripos – Part II, 2016** |
| Word Count: | **11,998**[1] |
| Project Originator: | Alex Braithwaite |
| Supervisor: | Dr Sean Holden |

## Original Aims of the Project

This project aims to develop a game derived from the Demolition Game mode in various popular first person shooter games. This will be used as a test-bed for evaluation of a reinforcement learning algorithm. The effect of modifying hyperparameters for this algorithm will be tested, alongside methods to automatically find the optimal hyperparameters for the algorithm.

## Work Completed

The success criteria were met, and a variety of extensions were implemented.

A modification to the Sarsa($\lambda$) algorithm was proposed and tested. Two further games were implemented, and the learning module developed successfully implemented in both. A new approach to learning with groups was developed and shown to be effective.

Learning was shown to occur in all games. A general method to find optimal values for the learning hyperparameters was implemented. The effect of each hyperparameter was assessed with suggestions on selection of these for novel problems.

## Special Difficulties

None.

[1] Counted using TeXcount

# Declaration

I, Alex Braithwaite of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

# Contents

# List of Figures

# Acknowledgements

Thanks to my supervisor, Dr Sean Holden, for giving me advice in weekly meetings throughout term time, and my Director of Studies, Dr Robert Harle, for making me feel guilty about taking time off. Further thanks to my parents for helping fund my computer, education and accommodation, making this work possible.

# Chapter 1

# Introduction

## 1.1 Motivation

First person shooters are a genre of computer game in which a player views the world from a first person perspective, shooting enemy players, and potentially coordinating with team-mates to achieve some objective. First person shooter games are a firmly established industry, with games such as Call of Duty: Black Ops II selling over 7.5 million copies in just 11 days [1], meaning improving game-play for these games could be very valuable.

Reinforcement learning is an area of machine learning in which agents observe an environment, perform actions, and receive rewards. The agent learns which actions are preferable in a certain environment to maximise the future reward. This is arguably the most general form of learning, and one which aims to mimic the manner in which humans learn to solve real world problems. A particularly useful version of this for video games is online reinforcement learning, in which the agent can learn as it plays, with no game reset and feedback required to learn.

One issue with first person shooter games is that the AI used for the computer controlled players is too often distinguishable from true human players, partly due to deterministic scripted behaviours. This can clearly detract a player from enjoying the game, also requiring significant effort from developers to script all the potential behaviours.

Agents teaching themselves to play could remove predictability associated with a prescripted AI, as their behaviour changes as the game progresses. This would also reduce the necessity for a team to write the whole script themselves, improving the quality of a game and reducing production costs.

## 1.2 Related Work

Reinforcement Learning was first used by Samuel in teaching a machine to play Checkers [2], with good results. Arguably the first major success of reinforcement learning was TD-Gammon [3], where a Backgammon player was trained using TD learning [4], a form of reinforcement learning, managing to defeat the world champion and change the theory of backgammon play. A more recent, and possibly even greater, success in the area of board games is Google's AlphaGo [5], recently defeating the European and later the

World Champion at Go. This was based around large convolutional neural networks trained using supervised learning on expert games, followed by reinforcement learning approaches explored in this project.

Reinforcement learning has also been used in video game playing [6, 7, 8], with another recent success from DeepMind's project on training a General Atari game player [9], applying the Q-learning algorithm [10] to deep neural networks.

The same algorithms have been used in real life problems as well, such as training robots to perform various tasks, including juggling [11], box pushing [12] and collecting/-transporting disks [13]. All of these have been successful, showing reinforcement learnings applications are not limited to simple games.

## 1.3 Aims

This project first aims to develop a first person shooter game, involving teams of agents, obstacles and complex objectives. A reinforcement learning algorithm will be built into this game, allowing for the AI to improve as the game progresses.

Furthermore, I will look into methods to find optimal hyperparameters for the algorithm. Extensions will include development of Backgammon and Stock Market Simulator APIs to test learning in these environments. The effect of giving the learner memory, modifying the learning algorithm slightly, changing the value of certain hyperparameters over time and varying exploration strategies will be evaluated.

# Chapter 2

# Preparation

## 2.1 Theoretical Background

This project is based around reinforcement learning, so a brief understanding of the field is important. The primary sources of information for this chapter have been Sutton and Barto's Reinforcement Learning: An Introduction [14] and Kaelbling, Littman, and Moore's Reinforcement Learning: A Survey [15].

### 2.1.1 Reinforcement Learning

In reinforcement learning, the learner is called the *agent*, which interacts with the *environment*. The agent provides actions to the environment, which then provides rewards to the agent. The agent must learn a strategy to maximise the reward received. This differs from supervised learning, as no labelled examples are provided; instead the agent must learn how to label states itself by exploration of the state space and observing rewards.

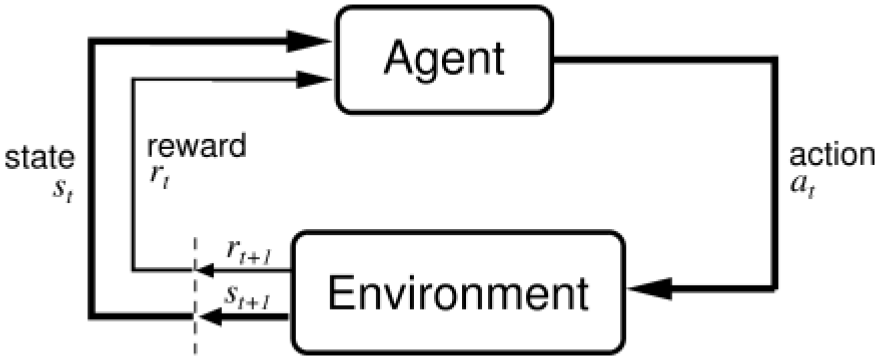


Figure 2.1: The Reinforcement Learning Problem (from [14])

An agent must learn how its action in a current time step can influence not only the reward immediately after taking that action, but the future expected reward too. This long term reward can be measured in a variety of ways. The most commonly used way is a discounted reward, where rewards in the future decay by a discount factor $\gamma$. The agent learns a strategy to maximise this discounted reward, $\sum_{n=0}^{\infty} r_{t+n}^{\gamma}$, where $r_n$ is the reward at time step $n$ and $t$ is the current time step.

According to [14], a reinforcement learning problem consists of four main elements; a policy, a value function, a reward function, and, optionally, a model of the environment.

The following paragraph is paraphrased from [14].

The policy defines how an agent behaves in any given state, mapping an observed state to a probability distribution over all the actions. The value function is the expected future reward in a given observed state, or the expected future reward when taking a given action in an observed state. The reward function then defines the reward given in each state. The entire state is not necessarily observable by the agent, meaning the agent will have to learn to deal with uncertainty.

Reinforcement learning algorithms fit into two broad categories: model-based and model-free algorithms. A model-based algorithm will learn a model for the world, which, for example, may try to predict a future state given a state-action pair. This can then be used to help define the value function and plan the optimal policy, deriving a controller from the model. Model-based algorithms will normally use the learning data available more effectively, requiring less data to achieve good performance, but every update will require more computation. Model-free algorithms do not try to learn a model for the world, only learning a controller. These generally need more experience to perform well, but require less computation before converging to a solution [15].

#### Choice of Algorithm

For this project I will be using a model-free algorithm, as my experiments will be on computer simulations where training data can be generated quickly. The main model-free reinforcement learning algorithm is Q-learning [10].

Q-learning is an off-policy learning algorithm. This means the learner will learn an optimal strategy, but its actual policy when learning could differ from this, as exploratory actions may be taken as opposed to exploitative[1]. An on-policy learning algorithm will instead learn the optimal strategy given its current policy. This means it will learn to make allowances for mistakes made due to exploratory actions.

Off-policy learning can often lead to divergence, and less 'sensible' behaviour than on-policy learning due to not making allowances for the exploratory actions taken whilst learning. For this reason, an on-policy algorithm should be used.

Sarsa($\lambda$) [16] is the on-policy version of Q-learning. The on-line version of Sarsa($\lambda$) will be used for this project, as on-line updates potentially allow for a faster learning rate [16], and there is no need for a trial to end before learning can take place. This is particularly useful in longer games with incremental rewards (as opposed to a single reward signal at the end of a game), and necessary in games which never end.

### 2.1.2 Sarsa($\lambda$)

At the core of the algorithm is a $Q$-table. This is addressable by a state $s$ and action $a$. $Q(s, a)$ is the expected future discounted reward received from taking action $a$ in state $s$.

[1] An exploratory action is an apparently suboptimal action according to the current estimate of its reward. It is useful to take these as the initial estimate of the future discounted reward from taking each action will be incorrect.

This table can be updated by a simple update equation as experience is gained by taking actions and exploring the environment.

$$
\begin{aligned}
Q_{t+1}(s,a) &= Q_t(s,a) + \alpha\delta_t e_t(s,a) \quad \text{for all } s, a \\
\delta_t &= r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \\
e_t(s,a) &= \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases}
\end{aligned}
$$

Here, $\alpha$ is the learning constant, which can be varied to control the rate at which values in the $Q$-table change. The Sarsa($\lambda$) algorithm incorporates the discounted reward factor, $\gamma$, into it's update equation in $\delta$, the error in our estimate of the $Q$-value.

Eligibility traces allow for the use of a temporal difference method [4] to help update the $Q$-value. These traces are incremented whenever a state-action pair is encountered (accumulating traces [17][2]), and decay over time. $\lambda$ is the discount factor, determining how much an error at the current time step will feed back to previous $Q$-function estimates. This effectively gives the learner a form of memory, where unexpected new experiences will be used to update the $Q$-value for past state-action pairs accordingly.

```
initialise $Q$ table arbitrarily
initialise the state, $s$
initialise action $a$ from policy
initialise all eligibilities to 0
for every time step do
    take action $a$
    observe reward $r$, state $s'$
    select $a'$ using policy and $s'$
    $\delta = r + \gamma Q(s', a') - Q(s, a)$
    $e(s, a) = \gamma\lambda e(s, a) + 1$
    for all s”,a” do
        if $a” \neq ors'' \neq s$then
            $e(s'', a'') = \gamma\lambda e(s, a)$
        end
        $Q(s, a) = Q(s, a) + \alpha\delta e(s, a)$
    end
    $s = s'$
    $a = a'$
end
```

**Algorithm 1:** Tabular Sarsa($\lambda$)

[2] An alternative method is to set eligilities to one whenever the pair is encountered. This is called resetting traces [18], not used in this project.

### 2.1.3 Function Approximation

In most real life problems, the state space is likely to be either very large, or, more often, continuous. This can be discretised and stored in a table, however this results in huge $Q$-tables (requiring impractical storage space), and no generalisation between similar states. This table needs to be compressed somehow, and ideally experience in one state should carry over to similar states. In the majority of cases where reinforcement learning is used, most states encountered will not have been experienced before in exactly the same form, meaning generalisation is completely necessary.

If the the $Q$-table is thought of instead as a $Q$-function, with a state-action pair as its input and returning an estimated future discounted reward, then it can be seen that a function approximator can be used instead of a table, allowing the use of supervised learning to update the estimates. An effective form of function approximation is a backpropagation artificial neural network. This can be used then to represent the $Q$-function in the Sarsa($\lambda$) algorithm.

Two options available are to either have one single neural network, taking the state as an input vector, and producing an output vector of $Q(s, a)$ for every $a \in A$, where $A$ is the space of all actions. Alternatively, there can be a large number of networks, each taking the state as an input vector and producing a single output, corresponding to the $Q(s, a)$ value for a single $a \in A$. This approach may increase the computation required for every time step, as many networks will need to be evaluated as opposed to one. However, it means each network can be smaller, as they only need to work for a single action each, and interference between actions will be removed. This is the approach used in this project, as shown in figure 2.2.

The update equations for Sarsa($\lambda$) are changed to update equations for the weight vectors of the networks, and every edge in each network maintains its own eligibility trace.

$$\begin{aligned}
\delta_t &= r_t + \gamma Q_t(s', a') - Q_t(s, a) \\
w_{t+1} &= w_t + \alpha \delta_t e_t \\
e_t &= \gamma \lambda e_{t-1} + \nabla_w
\end{aligned}$$

The calculation of $\nabla_w$ follows the normal backpropagation algorithm for an artificial neural network. For completeness, its calculation is shown in Appendix A.

### 2.1.4 Policy

The policy, given a state, selects an action. It is important to include an element of randomness to ensure exploration occurs, such that the agent will not get stuck on a single path (resulting in the agent learning a strategy optimal for that path, but not for the whole environment).

The $Q$-learning algorithm will converge to a correct value function if every state is visited an infinite number of times [20]. This proof may not directly apply to the problems in this project (due to usage of a function approximator, non-deterministic environments and incomplete knowledge), but the basic idea will still apply. Some state-action pairs

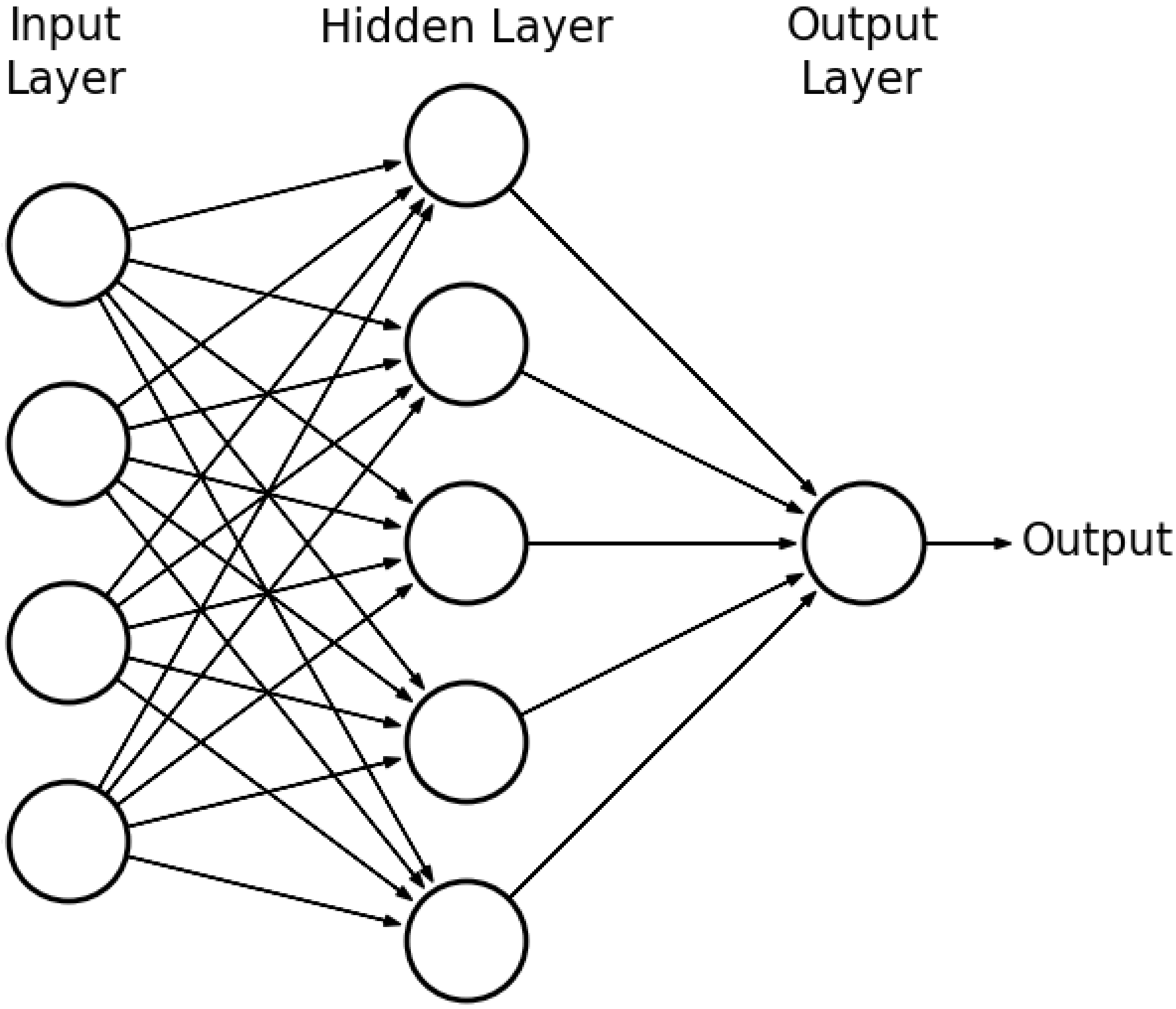


Figure 2.2: Fully connected feedforward neural network with a single hidden layer and output (from [19])

will be encountered frequently when following an optimal policy, therefore we care about the $Q$-function values for these more than that for those which would be less frequently visited. This means the policy should visit some state-action pairs more frequently than at random, so a purely random exploration policy is undesirable.

In this project, two methods of exploration will be implemented. The first is called the $\epsilon$-greedy policy. With probability $(1 - \epsilon)$, the action to maximise the value of the $Q$-function in the given state is chosen, but with a probability of $\epsilon$ a random action is chosen.

$$a = \begin{cases} \arg\max_a Q(s, a) & \text{with probability } (1 - \epsilon) \\ \text{random } a \in A & \text{otherwise} \end{cases}$$

The second is the Boltzmann distribution strategy. Actions are selected from a distribution based on respective $Q$-function values; actions with a higher $Q$-function value in the current state are more likely to be selected. This distribution uses the parameter $T$; $T = 0$ corresponds to selecting the action to maximise the $Q$-function with probability 1.

$$Pr(a|s) = \frac{e^{Q(s,a)/T}}{\sum_{a' \in A} e^{Q(s,a')/T}}$$

Different combinations of these methods will be explored by varying the constants $\epsilon$ and $T$. An action will be selected from the Boltzmann Distribution first, and then with probability $\epsilon$ will be changed to a random action.

## 2.2 Software Engineering

### 2.2.1 Development Model

As the project has a few well defined and independent modules, an Object Oriented design model will be followed. Encapsulation is also a large aim of this project, aiming to make the Learner module possible to use with no need to understand its precise implementation.

The design process will follow a slightly modified version of the Waterfall Model, as shown in figure 2.3. The requirements are known to a reasonable degree of accuracy in advance so there should be no need to re-evaluate these after each design iteration. This model also stresses the importance of the requirements analysis and design stage, important here as a thorough understanding of the theory is required, and a good plan needs to be made to ensure the implementation is correct.

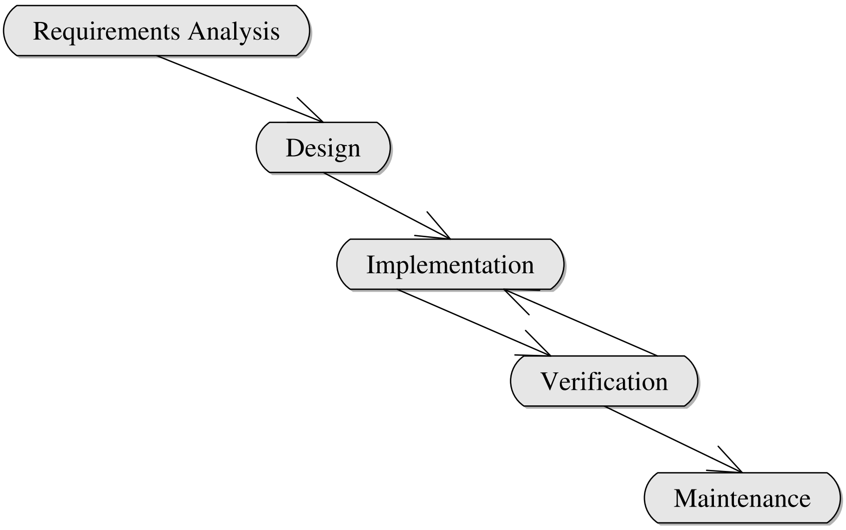


Figure 2.3: Modified Waterfall Model

The implementation and verification stages within the Waterfall Model will take inspiration from the Spiral Model. Incrementally more complete, complex and effective prototypes will be created for each major module, to allow unit testing of each set of features to be performed whilst the project is under development. This is important as the final product will be complex and bugs would be difficult to diagnose without an incremental approach to testing.

Once the initial implementation and testing of the core project is complete, the maintenance stage will consist of using an Agile development approach to push updates, changing or adding small parts of the project to explore the project extensions.

### 2.2.2 Testing Strategy

Testing the mathematical models will be done by comparing results with Matlab functions. For the geometric modules representing shapes, a separate module will be designed to draw them to the screen. Edge cases will then be automatically generated and drawn for manual inspection. For example, to test collision of a line and a circle, lines will be generated and rotated such that they only just intersect with a circle, and this will be drawn, with the intersection points highlighted. The Neural Network will be tested by comparison of feedforward and backpropagation results with those calculated from a few hand drawn networks.

The Demolition Game module will only need Bomb logic to be tested beyond this, which will be done by adding in manual controls for an agent in the Game. This will also allow further testing of the collision detection and other features in the game. The Backgammon game module will be tested by manually generating a series of board states and their possible successors with a given dice roll, and checking that all of these (and only these) are returned as potential future board states. The test for the Stock Market simulator will check for correct reading of data.

The final test of the Reinforcement Learner will be trying to learn some boolean functions. If these functions can be learnt to a high degree of accuracy, the learner is likely to be correct and can be deployed into the full games.

## 2.3 Requirement Analysis

This project aims to produce a system which is capable of demonstrating learning using the Sarsa($\lambda$) algorithm. This system will be used inside a variety of games, and methods to maximise the rate of learning will be explored.

The project limits itself to only exploring the Sarsa($\lambda$) algorithm inside discrete time and action domains. A feedforward neural network with only a single hidden layer will be used as the function approximator in the final implementation. No testing will be done for the effects of multiple hidden layers or recurrent neural networks. Due to time and computational constraints no testing of the limits of performance will be done, only tests of the rate of learning.

### 2.3.1 Functional Requirements

The learner must be able to run in any arbitrary discrete time step game with a discrete action space. All the hyperparameters in the Sarsa($\lambda$) algorithm must be adjustable. Methods must be available for automatically searching for hyperparameters to maximise the learning rate.

### 2.3.2 Non-functional Requirements

The learner will be designed to be as simple to integrate into a game by minimising the method calls necessary for learning. At construction time the hyperparameter values will be specified. Games will implement an interface to allow for a method of searching the hyperparameter space for an arbitrary game, and a class must be made to search the hyperparameter space.

## 2.4 Project Design

The overall structure of the project can be seen in figure 2.4. From this, three main components to the project can be identified; the learner, the games, and a manager for running multiple games. This section will describe each of these components in detail.

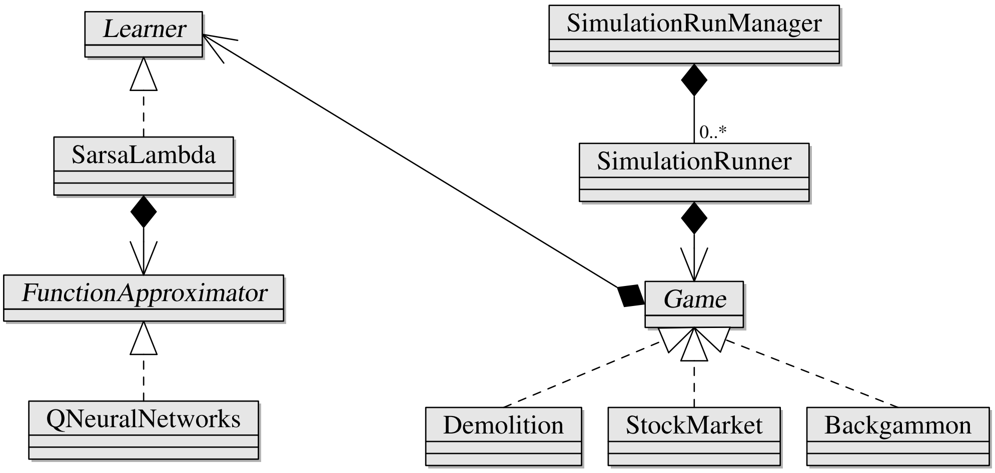


Figure 2.4: High Level Full Project UML Class Diagram

Two major interfaces will be implemented; one for the Learner, and a second for a generic game. This will ensure there is a clear definition of what the Learner should and can do (allowing for easy integration with any new game), and allow for easy development of further games compatible with the management module. A third interface will define the function approximator to be used by the Reinforcement Learner in the most generic way possible. This will allow for substitution of different function approximators depending on the problem to be solved (for example, using a table based learner for a simple, discrete environment, or a recurrent neural network for a problem where memory is useful).

### 2.4.1 Learner

The Learning module will provide functionality to take an input vector and produce an action based on the policy. In doing this, the Sarsa($\lambda$) update equations should be performed, and any memory should be updated and used. For the Backgammon game, it is also necessary to allow the learner to be given a state and produce an output vector of the expected reward when taking each action (with this representing the win chance for each player). This can then be used to choose the move the player will take, and learning should only be done on the sequence of actual board states. There will be the option to have learning take place from forced actions, as opposed to from an action selected by the policy, which is needed for training the predictor in Backgammon.

Implementation of the Learner Module will require understanding of the Sarsa($\lambda$) algorithm and how to integrate feedforward neural networks with it. The design of this module will be aimed towards maximising the ease of integration into a game.

### 2.4.2 Games

#### Demolition Game

In the Demolition Game an attacking team is tasked with arming a bomb at a bomb site, whilst the defending team must defend the bomb site for a set time limit. This leads to teamwork being required for success, with different strategies being beneficial for each team.

Firstly, basic vector maths and geometric shapes will need to be implemented (circles and lines). In the game each agent will be modelled as a circle with a vector direction. When an agent is killed, they will re-appear at their team's respawn area. The game map will be randomly generated, containing a few obstacles (modelled as lines, blocking movement and shooting), with a bomb site on the central line across the map. Each team's respawn area will be in a corner of the map. The bomb will initially be placed inside the attacking team's respawn area.

The outputs available to the agents will be: turn left and right, move forward, and shoot. Multiple actions can be performed at every time step. Shooting is done in the current direction of the agent, with a small random deviation. The Laser will not pass through obstacles or agents, and will cause damage if it hits an enemy agent.

The bomb can be picked up, armed and defused. These actions will all be performed automatically when an attacking team's agent (defending team for defusing) stands within a certain distance of the bomb for a sufficient time. Dying whilst carrying the bomb causes it to be dropped.

After the bomb is armed by the attacking team, the defending team can try to defuse it. If successful, the defending team wins. If the defending team fails to defuse the bomb in a certain time, the bomb explodes and the attacking team wins. If the attacking team fails to arm the bomb within a time limit, the defending team wins.

Each agent will have access to basic sensory inputs in the form of an array of sensors. These will give information about the distance to, and identity of, objects in their field of view. Additional inputs will tell the agent the current state of the bomb (dropped, carried by ally or enemy, and fuse set), the team the agent is on, its current health, bearing and distance to the bomb and bomb site, and whether the agent is carrying the bomb or not.

#### Stock Market Simulator

A second game to be implemented will be a Stock Market Simulator. The Learner will be fed Open-High-Low-Close (OHLC) data, and make predictions for the price at a fixed number of days in the future. Rewards will be given if these predictions are correct, allowing online training to take place.

For this, a source of OHLC data must be found. I will use Yahoo Finance, as they have large archives of data freely available. Training will take place with a large number of companies, aiming to avoid ever repeating experiences.

#### Backgammon

A third game is Backgammon, posing a very different learning problem from the previous two. Here, all board positions possible after a single move, given a current board state and a dice roll, will be presented to the agent, which will learn to predict the probability of each player winning given a current board state. A controller can be derived from this predictor (choosing the move to maximise win probability), and this can generate training sequences of board states from which the predictor can learn.

### 2.4.3 Simulation Manager

The Simulation Manager module must be able to perform two simulations, each given minimum and maximum values for the hyperparameters.

The first is an exploration simulation. A grid search should be performed between these minimum and maximum values, and the results output in an easy to manage format. Secondly, an optimisation simulation should be available. This will find the optimal values by first performing a coarse grid search, then recursing into the optimal sector of the grid until the desired level of precision is reached.

## 2.5 Tools used

### 2.5.1 Java

The project will be carried out in Java, chosen as I am already familiar with the language. Java abstracts away any memory management, which reduces the number of possible bugs, in particular harder to diagnose bugs. This is important as my project will involve a significant amount of programming.

Java is known to be slower than a lower-level language such as C++. This difference is significantly less than one order of magnitude for most problems [21]. I did not consider this an issue, as reinforcement learning has been used on far slower, older computers with plenty of success [3, 16], and computers today are far faster. This project also does not aim to train a world champion player, therefore such long simulations will not be necessary.

#### Libraries

This project will use the Apache Commons Maths library for its efficient implementation of some statistical functions (Student's t-distribution and the Normal Distribution modules).

Google's Guava library will be used in one iteration of the Learner, but is not used in the final product. This library is used to implement a cache to limit the size of the $Q$-table.

#### Integrated Development Environment

I will be using the Eclipse (Mars) IDE to help in my development, as organisation in this project will be very important due to the extent of the programming which needs to be done. No existing code base will be used. I will aim to include sufficient JavaDoc documentation to make the code I produce easy to use, modify and apply by anyone else.

### 2.5.2 Latex

Graphs and diagrams will be produced using the LaTeXpackages pgfplot, tikz and metaUML.

### 2.5.3 Version Control

Git will be used for version control. Branches are made whenever a new feature is being added, and merged back to the main project whenever the feature is complete and fully working. Version control is important as it can also allow easy restoration to any back up point should new features not work as well as intended, or break other parts of the project.

### 2.5.4 Back up

The project will be backed up onto a Dropbox account daily, with weekly checkpoints onto an External HDD.

# Chapter 3

# Implementation

## 3.1 Interface Design

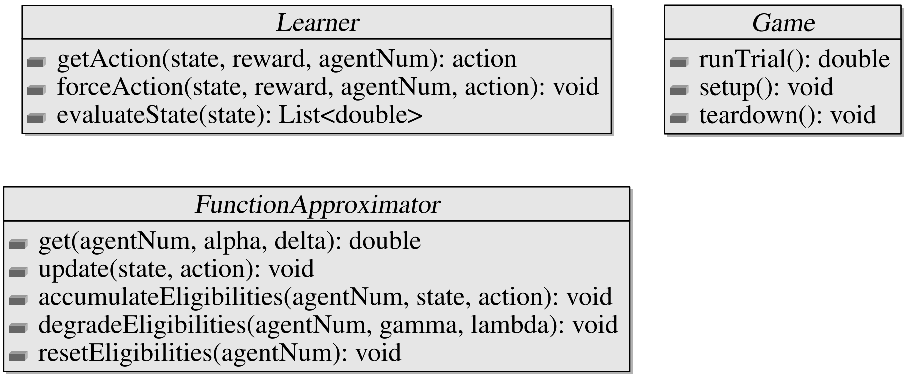

Learner
getAction(state, reward, agentNum): action
forceAction(state, reward, agentNum, action): void
evaluateState(state): List<double>
Game
runTrial(): double
setup(): void
teardown(): void
FunctionApproximator
get(agentNum, alpha, delta): double
update(state, action): void
accumulateEligibilities(agentNum, state, action): void
degradeEligibilities(agentNum, gamma, lambda): void
resetEligibilities(agentNum): void

Figure 3.1: Interface UML Class Diagram

This project will be based around the three interfaces shown in figure 3.1. The Game and Learner interfaces define the majority of functionality inside this project, and therefore must be designed carefully to offer all the functionality needed and avoid any unnecessary complexity. In addition, there will be an interface for a generic function approximator used inside the Learner module, allowing easy substitution and testing of different approximator modules.

### 3.1.1 Learner Interface

An agent using the Sarsa($\lambda$) algorithm will implement this, with the intention that other forms of reinforcement learner should also be able to do so equally easily.

This interface defines the only method used when training a controller. Designed to be used every time step of the game, the `getAction` method takes an input vector representing the state, alongside the reward received in the previous time step. An action is returned based on the policy of the learner, and the necessary update equations should be performed.

To train a predictor, a different method is needed. Reinforcement learning can be used to train a predictor by forcing certain actions to be taken every time step with no reward. When a given event occurs, the action is forced and a reward is given. This way, the $Q$-function becomes a mapping of state to a (possibly discounted) probability of an event happening in the future, each action corresponding to a different event. For this, the `forceAction` method is added. This will perform the same updates as the `getAction` method, but for a given action as opposed to one chosen using the agent's policy.

To utilise the predictor's result requires a method to access the results of the $Q$-function directly. This will take an input vector, and return a vector of the output of the $Q$-function for every available action in the given state. These results can then be used to derive a controller by, for example, choosing the next state to maximise the predicted probability of a win.

### 3.1.2 Game Interface

This interface will allow the Simulation Runner module to interact with games in a generic manner. This allows for easy expansion of the project scope to analyse the effectiveness of reinforcement learning algorithms in different situations.

A game implementing this interface will provide methods to set the learner to be used, set-up and tear-down, and run a single game. Some measure of the performance of the learning agent in the game is returned (for example, the total reward accumulated, or whether the agent won or not). This can then be used by the Simulation Runner module to find optimal values for hyperparameters.

### 3.1.3 Function Approximator

The Function Approximator handles the performing of the Sarsa($\lambda$) update equations (as described in Chapter 2). It also allows for retrieval of $Q$-function values, and manages eligibilities. This interface gives a method to get the $Q$-function value for a given state-action pair, alongside a method to perform the Sarsa($\lambda$) update equations. Methods are also provided for managing eligibilities (accumulating, degrading and resetting).

## 3.2 Learner Module

The learner needs to handle calls to the function approximator managing the $Q$-function, and also implement the policy.

The `getAction` method, used for training a controller, will first select an action using the policy described in Section 2.1.4. This action will be fed into the `forceAction` method, which will use the Function Approximator to update the eligibilities for the given state-action pair and perform the Sarsa($\lambda$) update equations.

### 3.2.1 Function Approximators

Several iterations of Function Approximators were used, each iteration aimed at fixing shortcomings of the previous.

First was a simple table-based approach using a Hash Map (one to store the $Q$-table, another to store eligibilities). The state space was discretised to enable storage in a table. Getting a $Q$-value is a lookup operation, and updates are to the specific state-action pair to which they are applied, along with all the stored eligibilities of previous state-action pairs experienced. Updates for this could take a significant amount of time (due to every visited state ever having an eligibility, often becoming vanishingly small). For this reason, eligibilities below a cut off point were set to zero. The $Q$-table would become huge over time, eventually using up all available RAM.

In response to this, Google's Guava library was used to use a Loading Cache to represent the $Q$-table. Whenever a given element in the $Q$-table is not accessed for a large number of steps it is removed from the cache. This results in only the more commonly experienced state-action pairs being stored. Surprisingly, there is very little drop in performance from using this method compared with storing the whole table, however learning is still unacceptably slow due to the lack of generalisation.

To speed up the learning rate function approximation was used. A fully connected feedforward neural network with a single output and one input fixed at one (removing the need for a bias in each neuron) was implemented. The feedforward and backpropagate equations were implemented as described in Appendix A, alongside management of eligibilities associated with each edge using the backpropagation results. The Function Approximator class has one of these networks for each action available, and either selects the appropriate network to get a value from or accumulate eligibities for, or performs the update equation on all of the networks. This resulted in far faster learning rates as experience from each state could transfer to others, and no artificial restrictions have to be added to the eligibilities.

To implement the neural network, a simple matrix class was needed. I looked into library implementations, but none were worthwhile as they were generally optimised for very large matrices, whilst the matrices in use here were all relatively small. A simple matrix class was implemented, able to be initialised from arrays of arrays in column or row major order. All the common matrix operations were implemented, making the network relatively simple to implement once the correct matrix equations were derived.

Each layer in the network was represented by a unique set of matrices for all the required features of the network: weighted input ($z^l$), activations (neuron output, $a^l$), edge weights ($w^l$, using the convention described in Appendix A), edge eligibilities ($e^l$), edge gradients with respect to output ($\nabla_w^l$) and error in each neuron ($\delta^l$). The output was decided to be the weighted input to the neuron in the final layer, $z^2$. This allows it to take the value of any real number, as opposed to using $a^2$, which is limited to the range -1 to 1.

### 3.2.2 Memory

A simple way to store memory whilst staying agnostic to the specific function approximator being used is to multiply the size of the state space, such that the current state indexed by the $Q$-table contains information about the current state alongside some previous states. The total size of the memory can be controlled, alongside its 'coarseness', ie the number of time steps between each snapshot of state which the $Q$-function sees, is controlled by two parameters (the memory length, and memory interval).

Implementation of this is straightforward, and particularly useful for problems similar to the Stock Market Simulator problem. The major disadvantage of this approach is the massive increase of state space size, making learning far more difficult.

### 3.2.3 Shared Learning

The Demolition Game in particular is a team based game in which two teams of agents must aim to achieve different objectives. The optimal policy for the agents on a given team should be the same, so their learning should converge to the same $Q$-function. It would be wasteful for each to learn this $Q$-function independently.

The experience of all team members can be shared, allowing faster learning to take place. The method with which this can be done is by storing unique eligibilities for every agent using a single learning module, alongside storing the previous state-action pair seen by each agent. The update equation can use each of these to update itself based on the experience of all agents using the learner, and all can benefit from each other's experience. If the agent has its current team as one of its inputs as well, all agents in the game can use a single Sarsa($\lambda$) learner.

This adjustment should allow for accelerated learning rates, however it may also lead to less specialisation of each individual agent (which may occur if each learn separate $Q$-functions), which could potentially harm performance.

### 3.2.4 Hyperparameter Decay

In gradient descent problems, reducing the rate of descent as the minimum is approached may lead to better results [16]. As Sarsa($\lambda$) performs gradient descent to find the correct values for the $Q$-table (or parameters for the function approximator), reducing the value of $\alpha$ (the learning rate) over time may improve the outcome.

At the initial stages of learning, more exploration is normally required. However, as the value function converges, less and less exploration should be done to ensure the agent experiences the more important states more frequently, allowing the $Q$-function for these states to be more accurate. Due to this, decreasing the values of $T$ and $\epsilon$ over time could be beneficial for performance of the network. This can allow for evaluation of the effects of decaying each form of randomness, and possibly of decaying each at different rates, potentially leading to a different form of randomness dominating at earlier and later stages of learning.

Finally, Tesauro suggested that reducing the value of $\lambda$ over time may be beneficial [22], but did not experiment with this in his trials of TD-Gammon. The reasoning for this is that a large initial value for $\lambda$ would help the algorithm move beyond complete random guessing relatively quickly, and later reducing the value would help allow for more fine-tuning of the $Q$-function with less disturbance from temporally distant state-action pairs.

There is no decay of $\gamma$, as this is a fundamental property of the system to be learnt, and adjusting $\gamma$ would change the $Q$-function which is to be learnt. For the same reason, parameters relating to the memory of the agents will not change with time either.

## 3.3 Games Modules

### 3.3.1 Demolition Game

#### Geometry

The Demolition Game is based in a 2D world, so the first step required is to implement the basic shapes representing the objects in that world. Everything will be formed from circles and lines.The main method to implement here is intersection, using the maths described in Appendix B.

#### Pathfinding Game

The original iteration of the full game involves creating a simple pathfinding game with a single agent being able to move. The game ends when a goal location has been reached, and includes obstacles blocking the agent's line of sight and movement.

Obstacles are represented by lines, and the agent and goal are circles. When the agent and an obstacle intersect, the agent is moved away from the centre of the two intersection points such that it is no longer intersecting with the obstacle. Obstacles are placed randomly inside the map, and four more enclose the map.

The inputs given to the agent are based on what the agent can directly see, alongside the bearing and distance to the goal. For sight, there are 9 sensors arranged as shown in figure 3.2, covering a 180° field of view in front of the agent. Sensors towards the front are separated by a smaller angle than those at the edges of the field of view, giving more detail towards the front. Distance for each sensor is measured by drawing a line from the agent at a given angle from its direction, and checking for intersections with this line, returning the distance to the closest intersection point. This was then transformed into three shifted values, each passing zero when the distance was low, medium or high. Transforming inputs in this way makes it far easier for a network to learn to use them effectively.

There are another three inputs for the distance to the goal, transformed in the same manner. Four more inputs inform the agent of the bearing to the goal. These are calculated from the angle between the vector connecting the agent to the goal, and the agent's direction rotated by a multiple of $\pi/2$. This allowed for each to 'flip' past the zero point at a different angle, again making these inputs easier for the network to use.

Agents have three actions available; turn left, right and move forward. Any combination of these can be performed in a single time step, leading to eight actions in total. As performing a left and right turn together would have the same effect as no turn, the action space can be reduced to six actions.

For testing purposes, manual controls are added to the agent. When activated, the agent turns towards the cursor and a key press causes the agent to move forward.

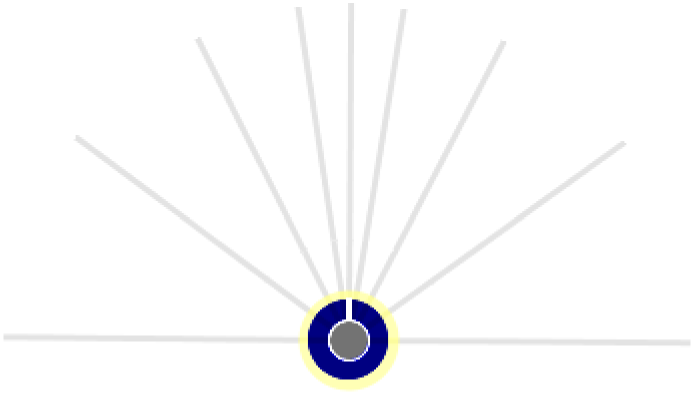


Figure 3.2: Agent Sensors

The agent is rewarded for approaching the goal, and punished for moving away by a slightly larger amount. When the reward for approaching was the same as the punishment for moving away the agent managed to find a way to have moving in circles lead to a net positive reward. It appeared the agent managed to learn to abuse rounding errors, so the rewards were adjusted to prevent this. Experiments were also performed with the agent only receiving a reward for reaching the goal, with similar results to those described later in Section 4.4.1.

### 3.3.2 Shooting Game

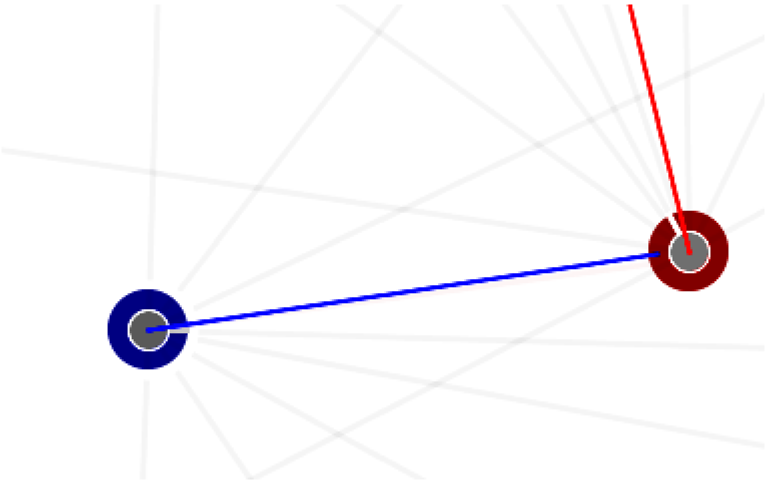


Figure 3.3: Shooting Game

The second iteration involves adding multiple agents and a Laser object. Agents have an additional action available to fire a Laser (doubling the size of the action space), and can die (after being hit by multiple Lasers), after which they respawn at a random location. Lasers are shot at a small random deviation from the current direction the agent is pointing, as seen in figure 3.3.

Additional inputs are added such that the agent can determine what type of object is seen by each sensor (whether the sensor detects another agent or an obstacle). Rewards are the same as before for approaching the goal, with additional rewards added for damaging or killing another agent, alongside punishments for dying.

### 3.3.3 Full Demolition Game

Teams are added, with each team only able to damage agents on the enemy team, and each team respawning in a set area. The goal was turned into a bomb site.

A bomb was added, with inputs added to the agents for their bearing and distance in an analogous way to the sensors for the goal (now the bomb site). Attacking agents can pick up the bomb, dropping it on death. When the bomb reaches the bomb site in the hands of an agent, it will be armed after a given time. This begins a countdown to victory for the attacking team, during which the agents on the defending team can defuse the bomb by standing nearby the bomb site. Additional inputs are added to allow the agents to determine the current state of the bomb.

A game ends with defender victory if the bomb is defused or not armed within a given time, or with attacker victory should the bomb be armed and not defused within the time limit.

Rewards are now specific to the phase of the game. Attacking team agents are rewarded for approaching a dropped bomb, carrying the bomb, approaching the bomb site whilst carrying the bomb, or arming the bomb. Defenders are rewarded for approaching an armed bomb and defusing the bomb. All agents still receive the same rewards for killing other agents.

### 3.3.4 Additional Features

As the sensor beam is a single line, it can sometimes miss objects by a small margin, causing the agent to be completely unaware of something in its field of view. A fix for this is each sensor consisting of multiple individual lines being shot at slight deviations from the exact angle of the sensor. The closest object to intersect with any of these lines is the object detected by the sensor.

With the game in its most basic version, agents tend to shoot and run non-stop, leading to very basic strategies. In order to try and introduce a cost to this strategy, fatigue and laser heat are added. Fatigue builds up as an agent runs, decreasing when the agent stands still. If fatigue rises too high, the agent becomes exhausted and can only walk until fatigue drops. Laser heat is similar, with the laser overheating if fired too frequently, preventing any shots being fired for a reasonably long period of time. The fatigue, laser heat and agent health are visually represented by the colour of various elements of the agent, as can be seen in figure 3.4.

### 3.3.5 Stock Market Simulator

This problem is based on training a predictor for trends in market data, however it can be rephrased to train a controller, allowing usage of the standard `getAction` method from the Learner module. For example, a car should turn left when there is a wall approaching to its right, and in an analogous way the learner should predict a rise in stock value when

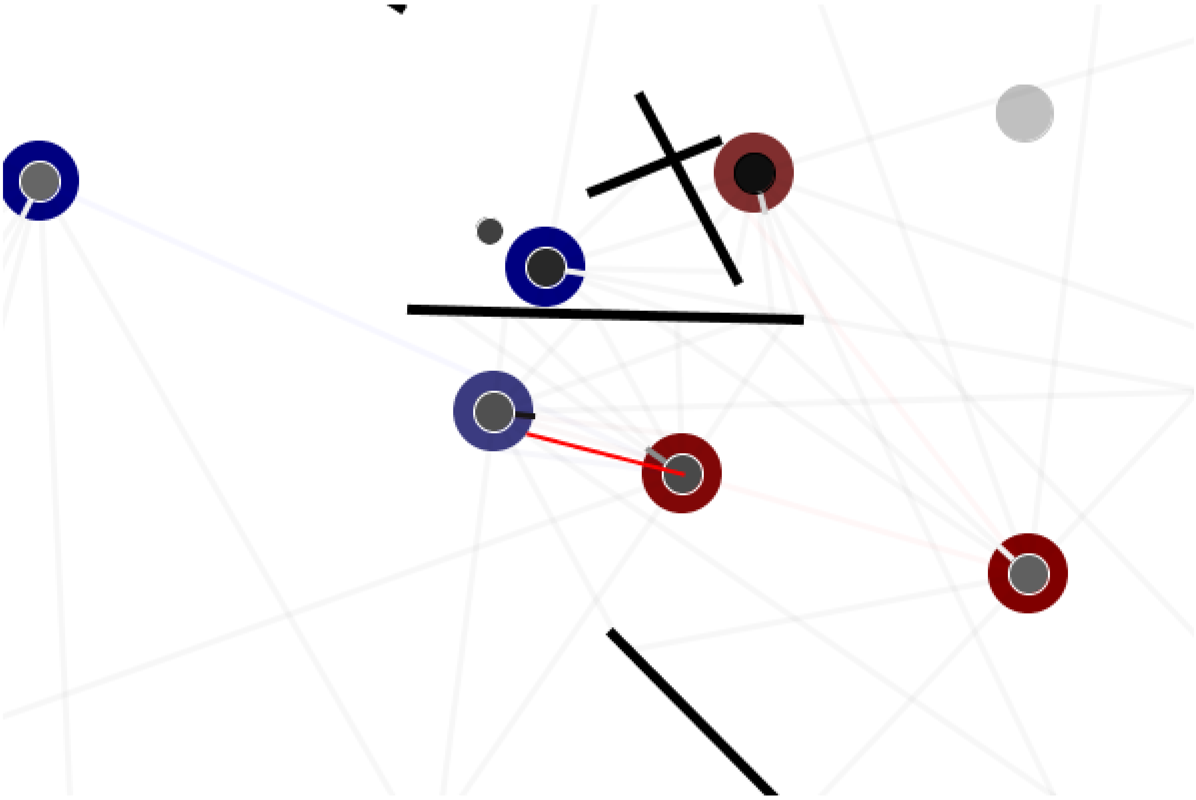


Figure 3.4: Full Demolition Game with all features

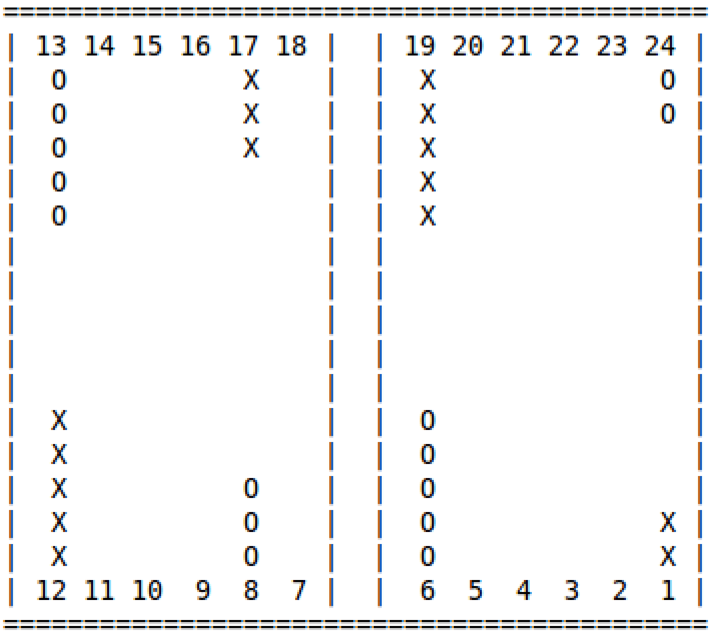
signs of a rise are present. In this simulation the outputs of the learner do not affect the state at all, however it can still learn to maximise its reward, as the ideal ‘action’ should still depend on the observed state.

The trading game uses market data from Yahoo Finance for a number of companies. This is given to the agent one day at a time, with one ‘game’ lasting for a year, using data for a random company and a random starting date. Data is formatted as fractional changes from the previous day’s value (as opposed to raw data), allowing for easier learning. The output is interpreted as a prediction, checked instantly, with a reward given to the agent if the prediction proves correct. Rewards are not delayed until the prediction proves correct or not; this would complicate the learning problem by adding in a difficult temporal credit assignment problem. The learner’s in-built memory system is used to allow for access to information about the past.

### 3.3.6 Backgammon Game

The Backgammon Game works by storing a board state, and producing all the next possible board states from this and a given dice roll. This is done with a basic approach of checking all possible moves, removing illegal moves, and finally returning the set where the two dice rolls are used in a legal manner.

In every trial, one game is played between two learning agents, and another is played with one side learning and the other taking random moves. The win rate against a random player is treated as the performance indicator to maximise when searching the hyperparameter space.



```
============================================
| 13 14 15 16 17 18 |  | 19 20 21 22 23 24 |
|  O           X    |  |  X              O |
|  O           X    |  |  X              O |
|  O           X    |  |  X                |
|  O                |  |  X                |
|  O                |  |  X                |
|                   |  |                   |
|                   |  |                   |
|                   |  |                   |
|                   |  |                   |
|                   |  |                   |
|  X                |  |  O                |
|  X                |  |  O                |
|  X           O    |  |  O                |
|  X           O    |  |  O              X |
|  X           O    |  |  O              X |
| 12 11 10  9  8  7 |  |  6  5  4  3  2  1 |
============================================
```

Figure 3.5: Backgammon Game drawn in ASCII

Algorithm 2 describes the process of using the Learner module to play a game of Backgammon against itself, and learn from the process. The Sarsa($\lambda$) update equations are used inside the learner, however this algorithm is different from Sarsa($\lambda$), which is meant to train a controller. With the use of forced actions, the same Learning module can be used for the Backgammon game.

```
reset learner eligibilities
initialise Backgammon board
while game not over do
    learner.forceAction(state = current, action = 0, reward = 0)
    learner.forceAction(state = current, action = 1, reward = 0)
    generate set of next possible board states
    output = whiteTurn ? 0:1
    current = $\arg\max_{\text{board} \in \text{nextBoards}} Q(\text{board}, \text{output})$
end
learner.forceAction(state = current, action = 0, reward = whiteWin ? 1:0)
learner.forceAction(state = current, action = 1, reward = blackWin ? 1:0)
```

**Algorithm 2:** Training and using a Backgammon Predictor

It was decided not to follow the exact Sarsa($\lambda$) algorithm here, as it is not clear how the unmodified algorithm can be used to train a predictor separate from a controller. The need to force actions could be avoided by only having a single action, and using this to predict one player's win probability, however this approach was found to give poorer game play than having two outputs (one for each players win probability), and is less extensible, for example adding prediction of the probability of a Gammon or Backgammon win and use these to help derive the controller (as used in Tesauro's TD-Gammon [22]).

An alternative approach would be to use the original Sarsa($\lambda$), allowing the learner to take moves directly (as opposed to presenting it with legal next board states). The problem with this approach is that the learner will have to teach itself which moves are legal before any meaningful learning can take place, which could take a very long time, especially if no intermediate reward signals are given. For this reason, the algorithm given above was used.

## 3.4 Simulation Runner Module

### 3.4.1 Simulation Run Manager

The Simulation Runner module manages the running of a large number of games and statistical analysis of the results. Two forms of simulation are available; one for generally exploring the hyperparameter space, and a second for performing a directed search for optimal hyperparameter values.

Each simulation searches between given minimum and maximum values for each hyperparameter, but typically only two or three will be varied due to the exponential increase in search space size for each additional hyperparameter to search for. The results from each simulation can be any numerical value representing performance of the learning agent, for example the total reward accumulated throughout a game.

As each game is completely independent of every other game, this search is embarrassingly parallel. Every game is run in a separate thread, and the search can be scaled up to utilise all of these. In the program, a ThreadPoolExecutor is used to manage all the simulations to be run.

#### Explore

The Explore method performs a grid search of a given coarseness between the minimum and maximum values. Repeats are run for every combination of hyperparameters, and the averages, standard deviations and 95% confidence intervals (calculated as shown in Appendix C)are returned for every trial for every combination. Example output can be seen in Appendix D.

#### Optimise

This method recursively runs until stopped. A coarse grid search is performed between the minimum and maximum values of hyperparameters given. Initially ten repeats are done, and the average and 90% confidence intervals are determined. The best (mean) result is compared with the second best (mean) result, and should the second best be outside the confidence interval for the best, then the search recurses into the section as shown in figure 3.6, setting new minimum and maximum values for each hyperparameter. The best combination of hyperparameters up to this point is output to a file, alongside the uncertainty in each. Example output can be seen in Appendix E.

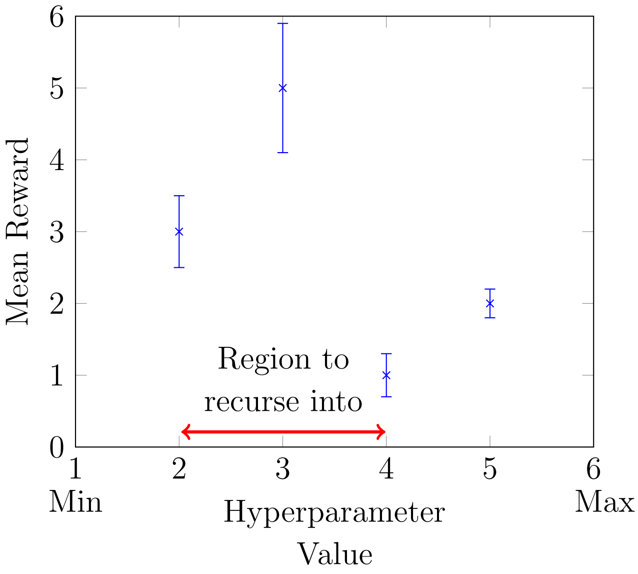


Figure 3.6: Splitting the search space in 4 between a minimum and maximum

Should the second best be within the confidence interval of the best, more repeats are run to obtain a more precise estimate of the mean. This repeats indefinitely until the second best falls outside the confidence interval of the best combination of hyperparameters.

### 3.4.2 Simulation Runner

A Simulation Runner is created by the Simulation Run Manager with a given combination of hyperparameters and a given game type. This will run a series of games of the given type, and a list containing all the results from these trials is stored inside the object for later examination by the Simulation Run Manager object.

# Chapter 4

# Evaluation

## 4.1 Comparison with Aims

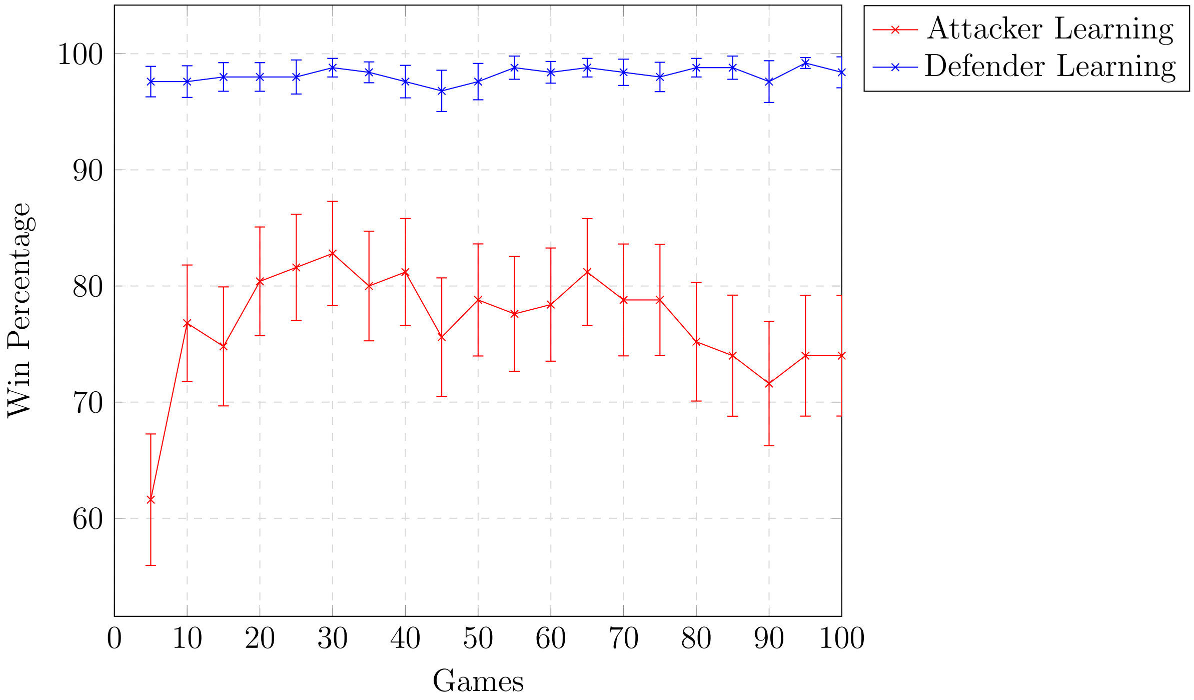


Figure 4.1: Moving average mean win rate over time for attackers or defenders learning against random opponents in the Demolition Game

The success criterion of the project was to train a team of agents to win the Demolition Game consistently when playing against a team of random agents.

Using the optimisation search method described in section 3.4.1, values for hyperparameters were determined. Games were then played with the learner controlling one team, and agents on the other team taking random actions.

Figure 4.1 shows the moving average mean win percentage. For the defending agents learning, this remains around 100% throughout, as for the attackers to win requires complex behaviour which random actions are very unlikely to cause. For the attackers learning, the moving average mean win rate never drops below 50%, showing sufficient learning to win can occur within a very small number of games, passing 80% within 20 games. This is still lower than the win rate for a defending team learning as the attacking team has more complex objectives to win, and some maps may be very difficult for them to win due to obstacle placement, meaning reaching the bomb or bomb site is complex.

## 4.2 Unit Testing

```
m1 = [1.0, 2.0 ; -3.0, 4.0]
m2 = [5.0, 7.0 ; -6.0, -8.0]
m3 = m1 * m2
m3 = [-7.0, -9.0 ; -39.0, -53.0]
PASS
m4 = m2 * m1
m4 = [-16.0, 38.0 ; 18.0, -44.0]
PASS
m5 = m2 * m1
m5 = [-16.0, 38.0 ; 18.0, -44.0]
PASS
m6 = m1’
m6 = [1.0, -3.0 ; 2.0, 4.0]
PASS
m7 = m1 + m2
m7 = [6.0, 9.0 ; -9.0, -4.0]
PASS
```

Figure 4.2: Matrix operations test

```
100000: 51% correct
200000: 52% correct
300000: 53% correct
400000: 55% correct
500000: 60% correct
600000: 68% correct
700000: 80% correct
800000: 95% correct
900000: 99% correct
Function Learnt
```

Figure 4.3: Sarsa($\lambda$) learning test

All tests were passed. Figure 4.2 shows the output for a matrix test, where matrices are printed in a format which can be copied directly into Matlab to check results. Figure 4.3 shows a test for the Learner module, here learning the XOR function (seemingly the hardest boolean function to be learnt by this algorithm). The inputs are randomly generated, and the network output (0 or 1) is checked against the correct result, with a reward given if the output is correct. A 99% correct rate is reached after 900,000 test examples here, demonstrating the agent is learning the boolean function to a good degree of accuracy (a significantly faster learning rate would probably be achieved if optimal values for the hyperparameters were used).

## 4.3 Hyperparameter Analysis

The Sarsa($\lambda$) algorithm has a number of hyperparameters which need to be fine-tuned in order to work effectively, or even at all. This section will analyse the effect of each in turn, providing insight into how these vary depending on the type of problem, and the importance of each. For all hyperparameters not specified, optimal values are used. Error bars all represent 95% confidence intervals.

### 4.3.1 Learning Rates

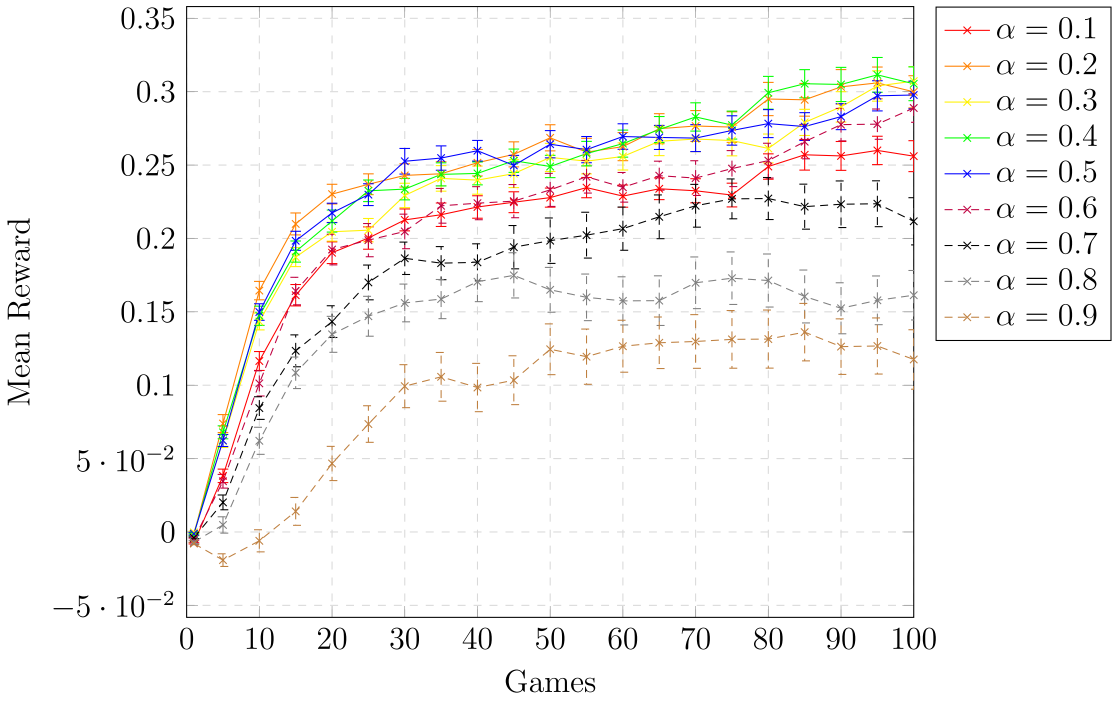


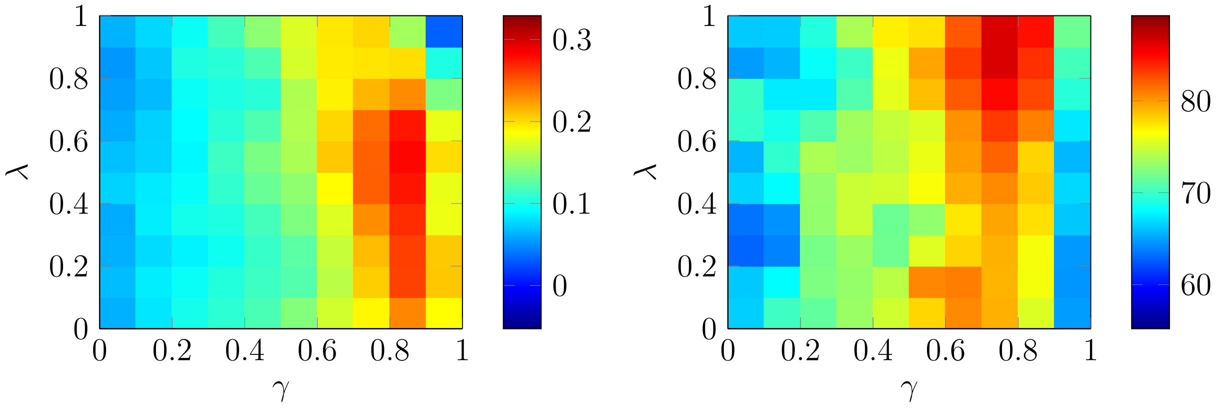
Figure 4.4: Mean reward over time with varying learning rate in the Demolition Game

All games are reasonably insensitive to learning rate variations. Figure 4.4 shows the effect of varying $\alpha$, the learning rate, on the mean rewards agents on the attacking team earn per time step in the Demolition Game, averaged over 50 repeats.

Values of $\alpha$ between 0.2 and 0.6 all showed remarkably similar learning rates. The further $\alpha$ is from this range the lower the performance and the higher the variation between trials. This variation can be attributed to, for lower values of $\alpha$, a higher reliance on the (random) initial weights of the network alongside insufficient response to learning stimuli. For higher $\alpha$, the high variance will be due to overshooting, causing large swings in behaviour as the $Q$-function changes significantly, causing some games to be played far more poorly than others.

This plot also shows that the number of games spent learning is unlikely to significantly alter the optimal value of $\alpha$ to use. This is useful as it means long simulations are not necessary to find a good approximation for the optimal value of $\alpha$. It appears to be sufficient to find a value for $\alpha$ to only one significant figure, as the performance varies so little in the 0.2 to 0.6 range. Finer granularity tests were run within this range and revealed no large peaks, supporting this.

### 4.3.2 Discounted Rewards and Temporal Difference



(a) Demolition Game (after 20 games)

(b) Backgammon (after 100 matches)

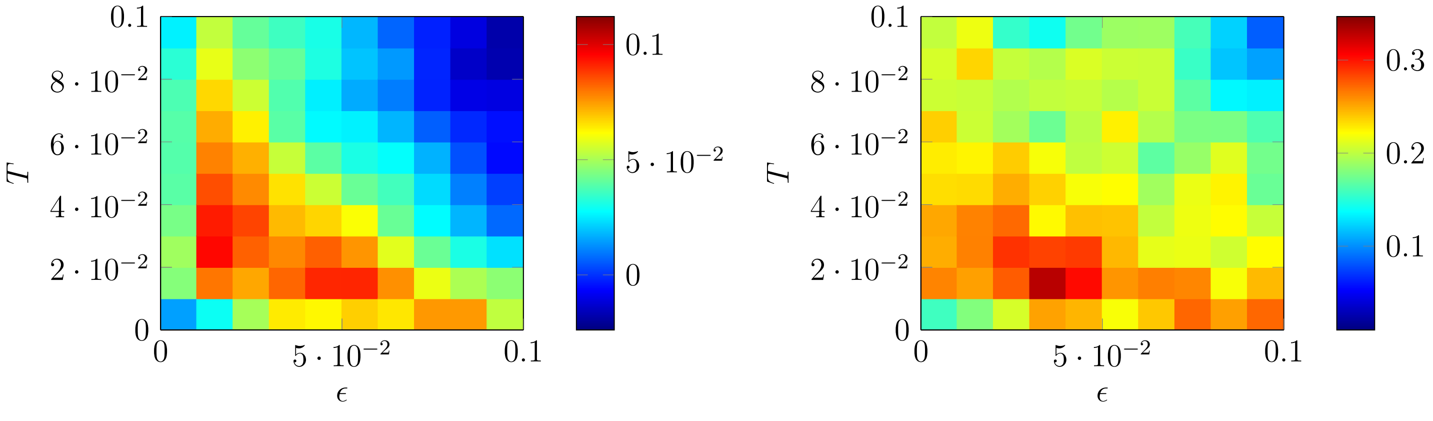
Figure 4.5: Effect of varying $\gamma$ and $\lambda$ on mean reward (a) and win rate against a random opponent (b)

In both the graphs in figure 4.5 it can be seen that higher values of $\gamma$ are optimal for both problems, however $\gamma = 1$ is poor for both. This is the case as the Sarsa($\lambda$) algorithm breaks down when $\gamma = 1$; the future discounted rewards will approach being infinite, meaning at $\gamma = 1$ almost all $Q$-values should be the same.

Interestingly, figure 4.5b shows performance is worse with $\gamma = 1$ than for $\gamma = 0.8$. This is unexpected here, as the learner is reset after each game, so the discounted future rewards should not approach $\infty$. $\gamma = 1$ will lead to the predictor trying to estimate the win probability with no notice for how far from the end of a game a given move is, whilst $\gamma = 0.8$ will take predicted time from the end of the game into account when selecting moves alongside win probability. It appears the latter strategy is in fact better for maximising overall win rate against a random opponent, possibly due to the lack of strategy from the random opponent.

In both problems TD learning is very useful, as can be seen from the significant performance improvements when $\lambda \neq 0$. Lower values of $\lambda$ are optimal for the Demolition Game due to the incremental rewards given, so there is rarely a long period of time between reward signals. Meanwhile, in Backgammon a reward is only given at the end of a possibly very long match. This means assigning credit to individual actions is difficult, and so the method of temporal differences is very useful here for assigning credit to moves possibly a long way back in time, as shown by the general trend of higher $\lambda$ values leading to greater performance. $\lambda = 1$ and $\lambda = 0.9$ have nearly equivalent performance, showing the TD method can still work even with $\lambda = 1$, in contrast with $\gamma = 1$ significantly harming performance in both games.

### 4.3.3 Exploration



(a) After 5 games

(b) After 25 games

Figure 4.6: Effect of varying exploration parameters on mean reward in Demolition Game

Figure 4.6 displays the trade-off between exploration and exploitation in a reinforcement learning problem. Both graphs clearly show too low or too high exploration parameters consistently lead to poor learning results.

In figure 4.6a it can be seen that only a single form of exploration policy (one parameter set to zero) leads to significantly poorer results than a mixture of both policies, with clear 'hot' lines where one parameter remains low (but non-zero), whilst the other varies. Having both parameters equal appears to give poorer performance than one low and another higher.

After more training time, it appears the exact choice of exploration parameters becomes less important, as shown in figure 4.6b. The general pattern visible in figure 4.6a can still be seen, with hot spots again where $T$ is low and $\epsilon$ slightly greater. There is now far less drop off for higher exploration parameters, however having no exploration occur still leads to significantly less learning taking place.

This suggests exploration parameter choice is important for ensuring optimal performance during the early stages of learning, where a mixture of action selection policies should be used, possibly with one dominating over the other(s). However, due to Sarsa($\lambda$) being an on-policy learning algorithm, as more training time passes the exact values of the exploration parameters become less important as the value function learns to accommodate for the randomness in its policy.

### 4.3.4 Network Complexity

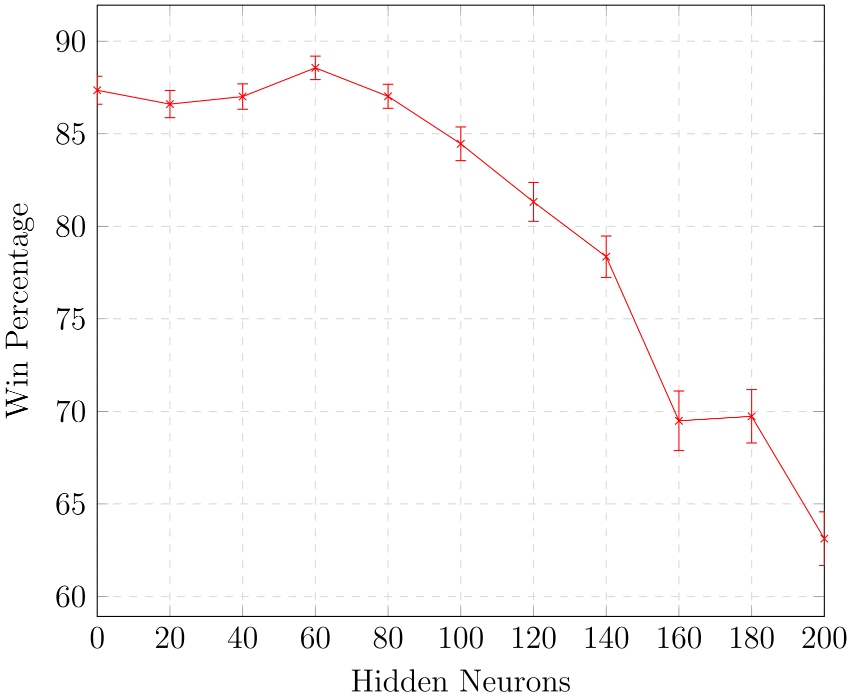


Figure 4.7: Effect of varying hidden nodes on Win Percentage after 250 matches

In reinforcement learning, where all new learning data is fresh, overfitting is unlikely to occur. However, there is still the related problem that a larger network will inherently learn a more complex function. This will require more data to learn, with Tesauro suggesting an exponential increase in learning time for larger networks [22].

Figure 4.7 shows the average win rate for a learning agent playing against a random player in Backgammon after 250 training games. A single hidden neuron shows remarkably good performance, despite only allowing for a simple linear function. This is likely due to the simplicity in strategy necessary to defeat an opponent making random moves. Performance increases up to 60 hidden neurons, before decreasing rapidly. This could be due to the increase in learning time required from adding further neurons, effectively as adding more neurons will reduce the generalising effect of the network. Adding training time did not change this trend, with the peak remaining around 60 hidden neurons. This suggests that for this level of problem complexity further neurons are not necessary, and is close to the number Tesauro used in TD-Gammon [22]. However, for a game of Backgammon against an intelligent opponent, more may be useful as the optimal strategy is expected to be more complex.

The win rate at 60 hidden neurons is close to the win rate expected when a high level player plays Backgammon against an opponent taking random moves[1]. Obviously win rate against a player making random moves isn't an excellent measure of performance

[1] On the First Internet Backgammon Server (FIBS), the lowest Elo rating is a little above 700, apparently achieved through a player deliberately attempting to minimise rating. Assuming random moves would give this rating, an 87% win rate corresponds to that expected from a player of rating 2350, greater than the top ranked player

(especially when the player is aware of the opponent's lack of strategy), but this can give a rough idea of the ability which may be achievable from a very small number of training matches.

## 4.4 Learning Rates

Besides varying the hyperparameters for the Sarsa($\lambda$) algorithm, there are also other choices to make which can significantly affect the performance of the learning agent. This section will discuss a few of these choices.

### 4.4.1 Reward Function

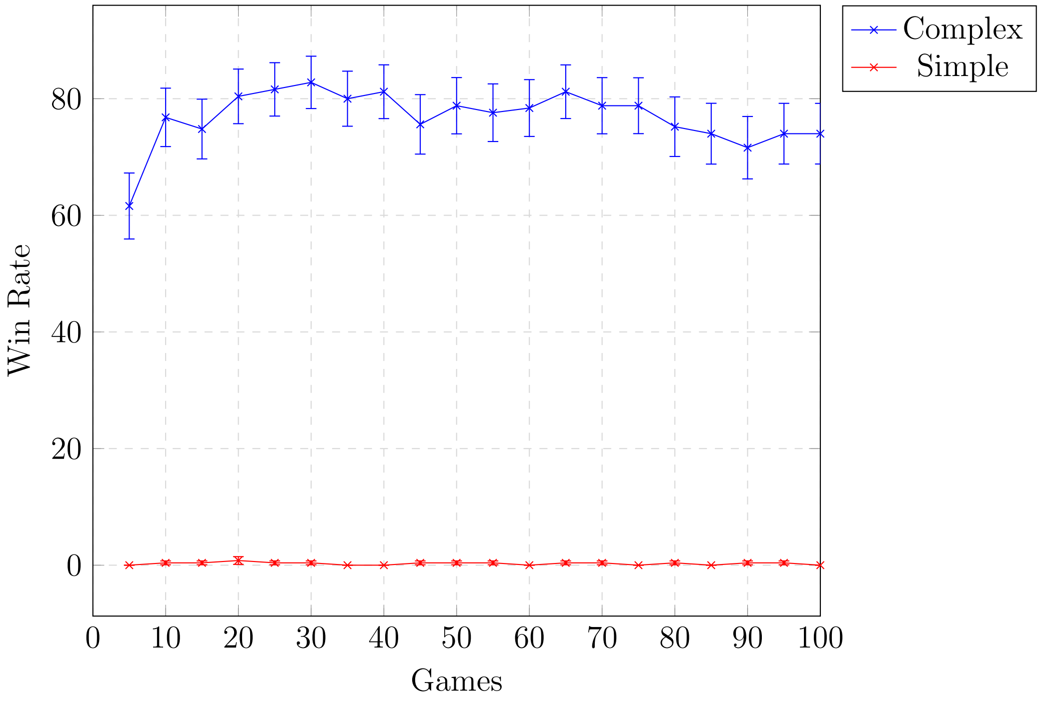


Figure 4.8: Attacker mean win rate with simple and complex reward functions

Normally, a reward signal will be given when the agent achieves some objective. Giving incremental rewards can help direct exploration in some path towards that objective. However, this can also potentially prevent the optimal behaviour, as certain pre-scripted and potentially sub-optimal behaviours will be rewarded artificially. For more complex problems, however, this is absolutely necessary, such as in DeepMind's attempt to train an agent to play the Montezuma's Revenge Atari game [9], where no successful strategy was learnt due to the long time before any reward is given.

Figure 4.8 shows this very clearly. When only a reward is given for winning the game (the Simple reward function), no learning appears to take place, with win rate remaining around 0%. On the other hand, when a reward is given for smaller tasks which are expected to be part of a successful strategy for winning the game (such as shooting an enemy or moving towards the bomb), learning occurs very effectively, quickly achieving high win rates. If at all possible, incremental rewards should always be given to help with the initial stages of learning. Once a basic strategy has been learnt it may be possible to gradually remove some of these incremental rewards to avoid imposing artificial limitations on the agent's strategy.

### 4.4.2 Shared Learning

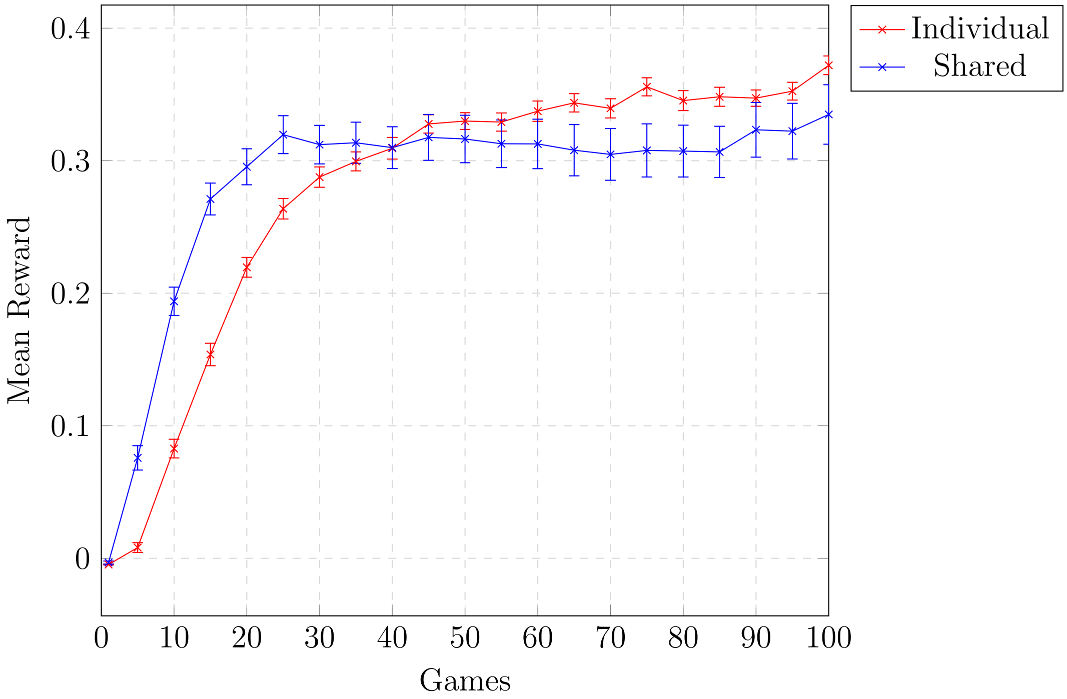


Figure 4.9: Mean Reward over time with team and individual learning in the Demolition Game

This project introduces a method of allowing experience to be shared among a group of agents; *shared eligibilities*. Figure 4.9 shows this helps dramatically with the initial stages of learning. This is because often a single agent will only experience a very small region of the state space, and may not experience any rewards at all, or learn a policy only useful for this small region and not for exploring the rest of the state space.

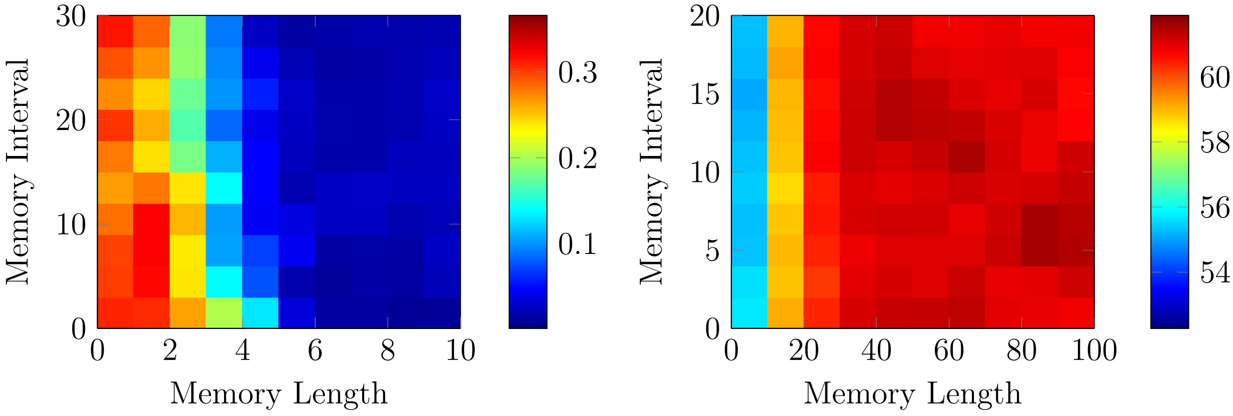
When experience is shared, lots of different states will be experienced simultaneously, meaning such dead ends are far less likely to occur. This will also ensure no agent's learning is left behind in a game, which could lead to some not learning optimal strategy. For example, never learning to shoot and instead always running away from enemies, or vice versa even when running would be a better choice should the enemy be likely to shoot back.

Shared learning can be seen to lead to a far faster initial rate of learning, even for a very small game with only six agents (this effect increases for larger number of agents). However, at later stages, this appears to be more of a hindrance, as the games with no shared learning tend to train more effective agents after 30 to 40 games. This is likely due to the potential for specialisation when the $Q$-function is not shared between agents.

In the Demolition Game, specialising each agent into specific roles (which shared learning will be very unlikely to achieve) could be very beneficial.

This suggests that *shared eligibilities* is a very effective method to kick start learning in games with a group of agents, but should be deactivated once the improvement rate slows down. Shared Learning can also be implemented in games with a single agent by running many games simultaneously, where the game must either be non-deterministic or having varying initial states.

### 4.4.3 Memory



(a) Demolition Game (after 20 games)

(b) Stock Market Simulator (after 15 years)

Figure 4.10: Effect of varying memory parameters on mean reward

Here, memory length refers to the number of previous time steps recalled, and memory interval refers to the inverval between each time step remembered.

Figure 4.10a shows that this method of giving the learner memory proved to be useless in the Demolition Game. Learning performance is consistently better for learners with no memory, showing a very clear drop off as the size of memory increases. For this problem memory would be useful (knowledge of the environment is incomplete and the game is non-deterministic), and the agent should benefit from using past experience to help it understand its current situation better. However, this naive approach to memory does not work here due to the huge expansion of the state space caused by duplicating all inputs.

For the trading game, figure 4.10b shows there is a clear benefit to having memory. Very short or no memory hinders performance, as predictions based only on a small amount of information are unlikely to be correct. Interestingly, here the memory interval appears to have no significant effect on performance. This could suggest some form of randomness of stock market data, meaning distant data is just as useful as recent data. The drastically smaller state space for the Stock Market Simulator (4 inputs, compared with nearly 100 for the Demolition Game) will also lead to this memory approach working better, as the expansion of the state space with duplication from memory will be far less extreme.

Increasing the length of the memory does not lead to any significant performance drop or improvement beyond a memory length of 20. This is likely due to a very simple function being learnt, as it is very difficult for complex strategies to be learnt for such a difficult task as stock market prediction with only raw data available, and this approach to memory does not appear to help with this. Adding many additional years of training leads to very little performance improvement, with the limit appearing to be a 65% correct prediction rate.

This shows giving the learner memory by having previous states alongside the current state as input to the agent is not effective for more complex problems, and alternative approaches should be investigated.

### 4.4.4 Hyperparameter Decay

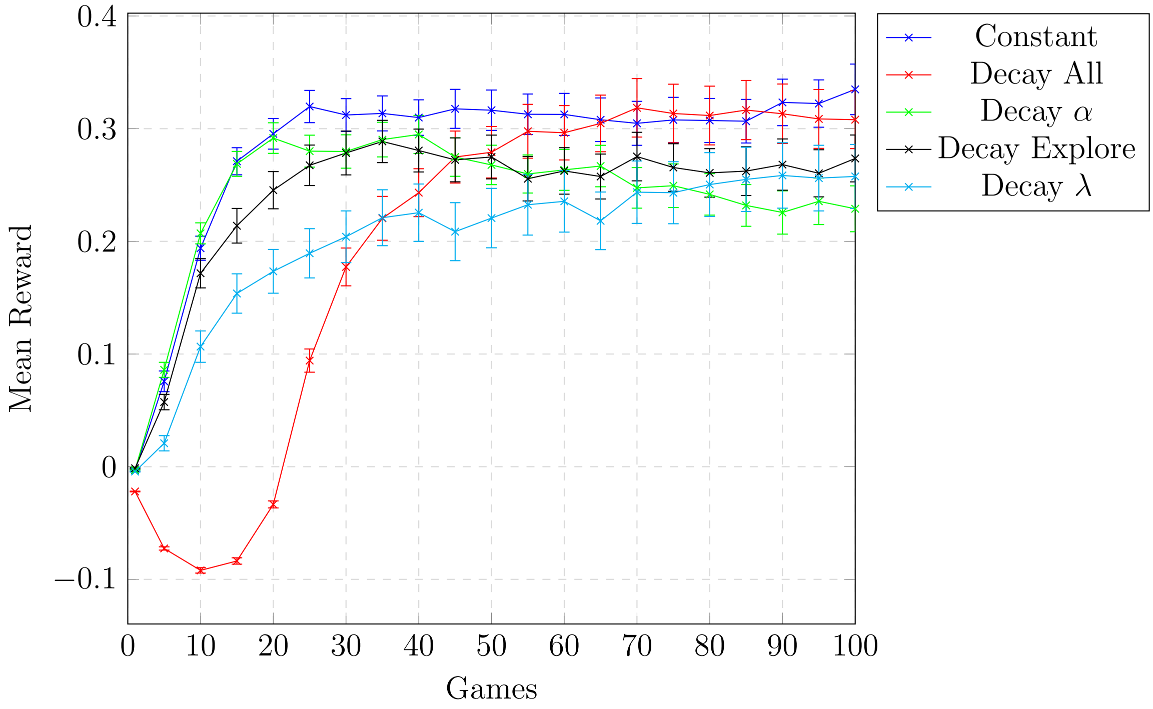


Figure 4.11: Mean Reward over time with constant and decaying hyperparameters in the Demolition Game

The overall trend in figure 4.11 is that constant values for hyperparameters lead to better performance than exponentially decaying values.

In the case of decaying $\alpha$, we can even see a negative learning rate. This is likely due to the environment being dynamic; as other agents learn and modify their behaviour the optimal strategy will change, and a decayed $\alpha$ may not enable effective adaptation to these changes. The poor performance when decaying $\lambda$ is also likely due to similar reasons, although performance can still be seen to improve gradually right until the end of these simulations. This suggests Tesauro's intuition that a decaying $\lambda$ could lead to better learning in the later stages [22] could still be correct, but may only be noticeable over longer learning periods.

Decaying the exploration hyperparameters ($\epsilon$ and $T$) appears to lead to a similar initial learning rate, but a reduced limit in performance. This is likely due to the fact the agents operate with incomplete information and no memory, where more determinism can actually harm performance, for example leading to getting stuck at dead ends through repetitive behaviours. As Sarsa($\lambda$) is an on-policy algorithm, a value function applicable to the non-deterministic policy will be learnt, so modifying the policy over time may also prevent Sarsa($\lambda$) from learning the value function effectively, as a different policy will lead to a different value function.

When all these hyperparameters are set to decay together (as opposed to all being fixed besides one or two), performance appears to improve beyond that of any individual one decaying. These simulations used rates of decay found optimal in isolation, as opposed to in combination (one large optimise simulation was not performed due to the curse of dimensionality; searching an 8-dimensional space is not practical, whilst two 2-dimensional and one 4-dimensional space searches are). This result is interesting as despite this the decay rates appear to complement each other, effectively making the whole more than the sum of its parts. Performance still, however, does not surpass that with no hyperparameter decay, and so exponential decay of hyperparameters is not a recommended method for optimising the performance of Sarsa($\lambda$).

# Chapter 5

# Conclusion

## 5.1 Achievements

A first person shooter game, alongside Backgammon and a Stock Market simulator, have been implemented fully and successfully. The Sarsa($\lambda$) algorithm has also been implemented and incorporated into all these games to train agents.

The success criterion of training teams of agents to play the Demolition Game effectively has been achieved. In addition to this, a successful Backgammon player has been trained, able to defeat an opponent taking random moves approximately as effectively as an expert level human player. Stock market prediction accuracy of around 65% on a large range of different companies has been achieved; significantly higher than that of random guessing.

A method to optimise hyperparameters based on recursive grid search and confidence intervals has been used to select the hyperparameters to achieve these results. In addition to this, each hyperparameter has been analysed in detail, helping provide intuition to guide selection of these hyperparameters for future problems, and guidance on the precision needed for each hyperparameter.

An extension to the Sarsa($\lambda$) algorithm to allow shared experiences has been evaluated and shown to lead to definite performance increases. A basic model for memory has been evaluated, and its weaknesses and strengths have been evaluated. An exponential decay model for hyperparameter values over time has been analysed, and shown to be insufficient for achieving performance improvements.

## 5.2 Further Work

This project was focused primarily on the learning rates of different combinations of hyperparameters; performance limits were not examined. Future experiments could look into how the choice of hyperparameters affects the limits of performance, and how the optimal choice for the best final performance differs from the optimal choice for the best learning rate over a set time period.

The Backgammon player was not trained using the exact Sarsa($\lambda$) algorithm due to the complexity this would give during the initial learning stage. It would be interesting to see what performance could be gained from allowing the player to make moves directly, assisted using incremental rewards as in the Demolition Game. The advantage of this approach is that there is no need to present and analyse every next possible game state, so this method would be more general and allow for faster game play, or a more complex network for the same speed of game play.

An incremental reward function was shown to be greatly superior to a more simple one, however no evaluation was done for modifying the reward function over time (reducing incremental rewards gradually to encourage the agent finding its own optimal strategy). This could be a promising approach to helping solve problems where no rewards are available for a long period of time, where an inefficient series of incremental rewards can be given at first to help the agent reach the first true reward signal, followed by reducing incremental rewards to encourage the agent to optimise its strategy.

Memory used in this project was all uniformly spaced; the detail available for events long ago was just as coarse as that for recent events. A better model could be some form of exponential backoff, with more fine grained temporal detail available for more recent memories (ie a smaller interval between recent memories, and larger intervals for distant ones). This approach should be compared with a uniformly spaced memory model. In addition to this, the memory approach used in this project was not at all successful for the Demolition Game, as shown in Section 4.4.3. An alternative approach to giving memory to a neural network is to allow for connections to form a directed cycle, giving a recurrent neural network as opposed to a feedforward network. A successful approach for this is the Long Short-Term Memory architecture [24]. This could be useful for reinforcement learning problems with large state spaces and similar behaviour to the Demolition Game.

Deep learning networks were not used in this project due to the increased number of hyperparameters necessary to specify them. It would be useful to evaluate the exact improvement received from having a deeper network with an equally complex shallow network, and comparisons of different topologies for different classes of problems would be useful.

Manually specifying the network complexity and architecture for the problem imposes artificial limitations on the performance achievable, and adds additional hyperparameter that need to be optimised. It has been shown that allowing the network topology to adjust itself can lead to better performance [25], so perhaps these methods could be adapted for use here in a way similar to that demonstrated by Whiteson and Stone [26]. This would also allow for deep neural networks to be trained, as opposed to only those with a single hidden layer.

Human assisted grid search is the widely used approach for solving the hyperparameter optimisation problem, this Dissertation improving on this by implementing a recursive grid search approach guided by confidence intervals. It has been suggested random search performs better than a brute force grid search [27], suggesting taking uniform sampling of the search space is a very inefficient method. Simulated annealing accommodates randomness into its probabilistic search strategy, and is a method which has been shown to perform well on various problems. It appears simulated annealing has not been applied to hyperparameter optimisation, despite having been shown to perform well in high-dimensional space [28] and for noisy cost functions [29], both characteristic of the hyperparameter optimisation problem. Evaluating the effect this has on time to find optimal hyperparmeter values compared with the approach used in this Dissertation, or a grid/random search, could be useful.

# Bibliography

[1] S. Nunneley. Black Ops sold 7.5 million copies in November - npd. *http://www.vg247.com/2012/12/07/black-ops-moved-7-5-million-copies-in-11-days-according-to-november-npd-data/*, 2012.

[2] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[3] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[4] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[5] D. Silver et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484489, 2016.

[6] D. Wang and A.H. Tan. Creating autonomous adaptive agents in a real-time first-person shooter computer game. *IEEE Transactions on Computational Intelligence and AI in Games*, 2015.

[7] J. Ivanovic, F. Zambetta, X. Li, and J. Rivera-Villicana. Reinforcement learning to control a commander for capture the flag. *IEEE Conference on Computational Intelligence and Games (CIG)*, 2014.

[8] Y. Liao, K. Yi, and Z. Yang. Reinforcement learning to play Mario. Cs229, Stanford University, 2012.

[9] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[10] C.J.C.H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

[11] S. Schaal and C Atkeson. Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, 14, 1994.

[12] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Proceedings of the Ninth National Conference on Artificial Intel ligence*, 1991.

[13] M. J. Mataric. Reward functions for accelerated learning. *Proceedings of the Eleventh International Conference on Machine Learning*, 1994.

[14] S. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* 1998.

[15] L.P. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artifcial Intelligence Research*, 4:237–285, 1996.

[16] WG. A. Rummery and M. Niranjan. On-line learning using connectionist systems. Technical report, CUED/F-INFENG/TR 166, Engineering Department, Cambridge University, 1994.

[17] R. Sutton. *Temporal credit assignment in reinforcement learning.* PhD thesis, University of Massachusetts Amherst, 1984.

[18] S. Singh and R. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 2:123–158, 1996.

[19] J. Kwak et al. Bivariate drought analysis using streamflow reconstruction with tree ring indices in the Sacramento Basin, California, USA. *Water*, 8(4):122, 2016.

[20] C.J.C.H. Watkins and P. Dayan. Technical note Q-Learning. *Machine Learning*, 8: 279–292, 1992.

[21] The computer language benchmarks game. `https://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html`. Accessed: 2016-05-09.

[22] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8: 257–277, 1992.

[23] B.P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[24] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[25] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2004.

[26] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *Machine Learning Research*, 7:877–917, 2006.

[27] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Machine Learning Research*, 13:281–305, 2012.

[28] D. Easterling, L. Watson, and M. Madigan. Direct search versus simulated annealing on two high dimensional problems. *Proceedings of the 19th High Performance Computing Symposia*, pages 89–95, 2011.

[29] W. Gutjahr and G. Pflug. Simulated annealing for noisy cost functions. *Global Optimization*, 8:1–13, 1996.

[30] V. A.H. Tan. Integrating temporal difference methods and self-organizing neural networks for reinforcement learning with delayed evaluative feedback. *IEEE Transactions on Neural Networks*, 19(2), 2008.

[31] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), 1995.

# Appendix A

# Feedforward and Backpropagation

The values $a^l$ represent the activations (outputs) of the neurons in the $l^{th}$ layer. $z^l$ is the weighted input for each neuron in the $l^{th}$ layer. $w^l_{i,j}$ is the weight of the edge from neuron $j$ in layer $(l-1)$ to neuron $i$ in layer $l$.

Feedforward happens by updating the $z$ and $a$ values. $z^0$ is initialised to the network inputs (not $a^0$). $a^0 = \sigma(z^0)$, where $\sigma(t) = \frac{1}{1-e^{-t}}$. $z^l = w^l a^{l-1}$ and $a^l = \sigma(z^l)$ for $l = 1, 2$. $z^2$ is the output of the network.

For a simple network with a single hidden layer and output neuron (as used in this project), $\nabla_w$ is calculated as follows:

$$\begin{aligned} \delta^2 &= \sigma'(a^2) \\ \delta^1 &= ((w_2)^T \delta^2) \odot \sigma'(a^1) \\ \nabla^l_w &= \delta^l (a^{l-1})^T \qquad \text{for } l = 1, 2 \end{aligned}$$

Where $\odot$ represents the Hadamard product of two matrices and $\sigma'(t)$ is the derivative of $\sigma(t)$.

For all networks representing the $Q$-function for the actions not selected, $\nabla_w = 0$.

# Appendix B

# Intersection

For circles this is a simple check of the distance between centre points against their radius. For two lines with equations $P_x^1 = x_1 + \lambda_1 dx_1$, $P_y^1 = y_1 + \lambda_1 dy_1$ (and analogously for the second line), the following equation can be derived from equating $P^1$ and $P^2$:

$$d_1 = \frac{y_2 dx_2 + x_1 dy_2 - y_1 dx_2 - x_2 dy_2}{dy_1 dx_2 - dx_1 dy_2}$$

$d_1$ refers to the distance along the first line at which the intersection occurs. An intersection occurs if $0 < d_1 < \text{len}_1$, where $\text{len}_1$ is the length of line 1.

The equation of a circle with radius $r$ and centre $c$ is $r^2 = (P_x - c_x)^2 + (P_y - c_y)^2$. Substituting the equation of a line ($P = l + \lambda d$) gives the following quadratic to solve:

$$(d_x^2 + d_y^2)D^2 + 2(l_x d_x - d_x c_x + l_y d_y - d_y c_y)D + l_x^2 + l_y^2 + c_x^2 + c_y^2 - 2c_x l_x - 2c_y l_y - r^2 = 0$$

Solving for $D$ gives distances along the line at which intersections (if any) occur with the given circle.

# Appendix C

# Confidence Interval Calculation

In this project confidence intervals were calculated for small sample sizes. I made the assumption that the data being averaged was normally distributed, which is reasonable as learning performance is likely to be randomly spread around a mean value, with the majority of results being close to the mean. Based on this, it is possible to use Student's t-distribution to calculate a 95% confidence interval as follows:

Firstly, choose $A$ such that $P(T < A) = 0.975$, where $T$ follows a t-distribution with $(n-1)$ degrees of freedom.

$$\begin{aligned} \bar{x} &= \frac{1}{n}\sum_{i=0}^{n} x_i \\ s_n^2 &= \frac{1}{n-1}\sum_{i=0}^{n}(x_i - \bar{x})^2 \\ CI &= A\frac{s_n}{\sqrt{n}} \end{aligned}$$

A 90% interval would use an $A$ such that $P(T < A) = 0.95$.

The Explore simulation calculated the means and confidence intervals as above once all games have finished. The Optimise simulation performed these updates online (using the algorithm described by Welford [23]) to remove unnecessary recalculations after every repeat.

# Appendix D

# Example Explore Output

```
EXPLORE SIMULATION
Initial Min:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=1.0E-5, TDecayRate=0.0, epsilonDecayRate=0.0,
          numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Initial Max:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=1.0E-5, TDecayRate=0.0, epsilonDecayRate=0.0,
          numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Simulation length:
      100
Divisor:
      21
Repeats:
      50

Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
   =0.95255647698944, lambda=0.3238726864076799, T
   =0.037311385459533594, epsilon=0.008779149519890258,
   alphaDecayRate=1.0E-5, TDecayRate=0.0, epsilonDecayRate=0.0,
   numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
   memoryInterval=15]
Means:
      [-0.0035676907508013595, 0.0030419849816865742,
         0.020672098825167094, 0.045465560157885075,
         0.07139881047220514, 0.098766538384027, 0.12449450204044267,
          0.1508398562278928, 0.18550900088857752,
         0.20193137451394944, 0.21798764133172172,
         0.23287670640832872, 0.23966559136368026,
         0.25137718421875144, 0.26377638826167954,
         0.2622866590422602, 0.2668548552892973, 0.27017957858732566,
          0.2642789503021676, 0.2674870543901067, 0.2624582737525582,
          0.2642430991931191, 0.264486835543637, 0.266787730579933,
         0.26322224180519077, 0.26447272518480214,
         0.2688603956992469, 0.26595667634389175,
         0.26485336942197074, 0.2646294686713962,
         0.26171251384229893, 0.26353121626705883,
         0.26036627163336606, 0.25410554559599985,
         0.2532485413732559, 0.25293080055105693, 0.2521162141116098,
          0.2437728952417177, 0.25006518537353495,
         0.24325952060081335, 0.24319888162071177,
         0.2375211380825303, 0.24065694751995598, 0.2405888507509907,
          0.24071181288158874, 0.23598630406888674,
         0.23879034272622046, 0.23500198587462848, 0.232481549317124,
          0.23203966890914252, 0.236224536128368, 0.2317252846275677,
          0.2338262995857396, 0.23350638616972433,
         0.23981732399207267, 0.24175312579051364,
         0.23917227016465933, 0.24040939185620036,
         0.24504426251218028, 0.24287782740491629,
         0.24090566525218918, 0.23952318744076295,
         0.23358580737774454, 0.23183121740951088,
         0.2351236228021707, 0.2323678874714675, 0.23610117268826328,
          0.24127155114828777, 0.24027310826345666,
         0.23851710522158517, 0.22831241382141826,
         0.23086213224279156, 0.2203664592123614, 0.2164156653958651,
          0.21838162821001172, 0.21578314886847316,
         0.21527775787583586, 0.21030186620680508,
         0.21103451406353874, 0.21168314611315794,
         0.21193406937522127, 0.21265308965262503,
         0.21201256245662356, 0.20486103845950468,
         0.2130996140234805, 0.20709636359227068,
         0.20866951578815982, 0.20595513433162554,
         0.20552415165576665, 0.2058742814619455, 0.2076961073126081,
          0.20449318741406142, 0.20394363205831972,
         0.20230230923499445, 0.1995937374361071,
         0.20785530695099916, 0.2110883903544163,
         0.20943644285831556, 0.20830150417712126,
         0.2063630901677245]
Standard deviations:
      [0.004406415493179091, 0.0155240271530308, 0.03238192309944599,
          0.04853368036867417, 0.057497962218300475,
         0.0673856275715896, 0.07167776888008145, 0.0764334156594904,
          0.08105608977987103, 0.07236398291879945,
         0.07120120026275294, 0.07297740740427382,
         0.06780904464168565, 0.06547275624270563,
         0.06751729076965056, 0.06636903296747955,
         0.0666164217567047, 0.07337341628568952,
         0.07913510350717047, 0.08301515764921484,
         0.08271210229726415, 0.08769379907438263, 0.090108516458431,
          0.0920710049491812, 0.09725052951697169,
         0.10076657538229902, 0.09790210743857697,
         0.0988189626632648, 0.10092917982412808, 0.099016343465495,
         0.10036989864330764, 0.1044298222238478,
         0.10625274527826276, 0.10937468483306685,
         0.1154229452780035, 0.11584199684522532, 0.1177057423966047,
          0.11475447883584732, 0.12242467594836039,
         0.12169946953913766, 0.12139760121584488,
         0.11934274756497858, 0.12502410831921373,
         0.12562346618875045, 0.12926969316759993,
         0.13157263994439347, 0.13326050380582966,
         0.13368836904887355, 0.13732793156417777,
         0.14061829539562412, 0.14130027284535224,
         0.13899820223338896, 0.14047135490177126,
         0.14339598957794294, 0.14366601648264907,
         0.14694114707073996, 0.1460533911874459,
         0.14772933088422124, 0.15097384003450798,
         0.1477511935475108, 0.14649931490439558,
         0.14713150676362927, 0.14456887991633452,
         0.1442346274890662, 0.1439241036794558, 0.14223212404118077,
          0.14410968632382873, 0.1482497988820308,
         0.1463349856226945, 0.14744636802910924,
         0.14238442660087486, 0.14729482171632827,
         0.14253743520609855, 0.14213036714577096,
         0.14540251120434478, 0.14554635635826282,
         0.14749686980812743, 0.14495180492740847,
         0.14712089011155308, 0.1485779948066528,
         0.14984804090690662, 0.1472215551013315,
         0.14782385723324895, 0.14153688397376077,
         0.1495057556399097, 0.14296818637484462,
         0.14324664027748008, 0.14193008386962605,
         0.1411905953683236, 0.14387899936607795,
         0.14692192560308154, 0.14531122850828043,
         0.14507859556200178, 0.14614216749413972,
         0.14472598228919892, 0.14939272953044574,
         0.15005858065353944, 0.15162032297745295,
         0.15278059672688493, 0.15190492798350977]
Confidence intervals:
      [5.189845007519424E-4, 0.001828408940134379,
         0.003813920003496759, 0.005716262552812103,
         0.006772069329891924, 0.00793663156305455,
         0.008442156931144005, 0.009002273645826186,
         0.009546728934760773, 0.008522978735362039,
         0.008386027016404656, 0.008595227437472798,
         0.007986501326147432, 0.007711334930642041,
         0.007952138761415406, 0.007816897769476104,
         0.007846035046127927, 0.008641869083484847,
         0.009320476530004097, 0.00977746656304036,
         0.00974177291799581, 0.010328513641555017,
         0.010612917347460026, 0.010844057854109935,
         0.011454098594958863, 0.011868215990574983,
         0.01153084098179659, 0.01163882754179825,
         0.011887367426746474, 0.011662074913105388,
         0.01182149568476995, 0.01229967061307602З,
         0.012514373201329268, 0.012882073034387632,
         0.013594432872512772, 0.013643788469764231,
         0.013863299102841967, 0.013515701367667422,
         0.014419091759533221, 0.014333677461490932,
         0.014298123623842293, 0.01405610441395073,
         0.014725251065961325, 0.014795843011994278,
         0.015225293047102337, 0.01549653248992442,
         0.0156953279019369, 0.01574572156769401 4,
         0.01617438666699855, 0.0165619233922571 6,
         0.016642246214019905, 0.016371109965271944,
         0.016544616844801404, 0.01688907824878029,
         0.016920881826664725, 0.017306624391293247,
         0.0172020651311431, 0.01739945612347409 6,
         0.01778159211681226, 0.01740203109249597,
         0.01725458571118663, 0.01732904482130725 3,
         0.01702722044341654, 0.01698785242890236 3,
         0.016951279154196034, 0.016751998988897985,
         0.016973136946813912, 0.017460756476202675,
         0.017235231124594748, 0.017366128958498626,
         0.016769937076676688, 0.017348279940946565,
         0.01678795832199858, 0.01674001413371488 4,
         0.017125404947008836, 0.017142346927515966,
         0.017372077022318622, 0.017072321080432107,
         0.017327794399533376, 0.017499411160118078,
         0.017648996291681932, 0.017339650650843363,
         0.01741058937001746, 0.01667011410540226 3,
         0.01760868217498501, 0.0168386918900440 9,
         0.01687148799378542, 0.0167164249110825 8,
         0.016629328478334843, 0.016945966800062297,
         0.017304360500425067, 0.017114653735617723,
         0.01708725439171342, 0.01721252114176656 5,
         0.01704572384979621, 0.0175953700397397 3,
         0.017673793514155347, 0.01785773442066634,
         0.017994390642376856, 0.017891254997018827]
```

# Appendix E

# Example Optimise Output

```
OPTIMISE SIMULATION
Initial Min:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=0.0, TDecayRate=0.0, epsilonDecayRate=0.0,
         numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Initial Max:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=1.0E-4, TDecayRate=0.0, epsilonDecayRate=0.0,
          numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Simulation length:
      20
Divisor:
      4

14 Mar 12:46:52
40 trials: Finished Round 1. 10 repeats this round.
Best Settings:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=2.0E-5, TDecayRate=0.0, epsilonDecayRate=0.0,
          numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Uncertainty:
      Settings [hiddenNodes=0, alpha=0.0, gamma=0.0, lambda=0.0, T
         =0.0, epsilon=0.0, alphaDecayRate=2.0E-5, TDecayRate=0.0,
         epsilonDecayRate=0.0, numAgents=0, numInputs=0, numOutputs
         =0, memoryLength=0, memoryInterval=0]
```

```
Next Round Min:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=0.0, TDecayRate=0.0, epsilonDecayRate=0.0,
         numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Next Round Max:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=4.0E-5, TDecayRate=0.0, epsilonDecayRate=0.0,
          numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]



14 Mar 13:16:02
120 trials: Finished Round 2. 20 repeats this round.
Best Settings:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=8.000000000000001E-6, TDecayRate=0.0,
         epsilonDecayRate=0.0, numAgents=6, numInputs=82, numOutputs
         =12, memoryLength=0, memoryInterval=15]
Uncertainty:
      Settings [hiddenNodes=0, alpha=0.0, gamma=0.0, lambda=0.0, T
         =0.0, epsilon=0.0, alphaDecayRate=8.000000000000001E-6,
         TDecayRate=0.0, epsilonDecayRate=0.0, numAgents=0, numInputs
         =0, numOutputs=0, memoryLength=0, memoryInterval=0]
Next Round Min:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=0.0, TDecayRate=0.0, epsilonDecayRate=0.0,
         numAgents=6, numInputs=82, numOutputs=12, memoryLength=0,
         memoryInterval=15]
Next Round Max:
      Settings [hiddenNodes=20, alpha=0.2578875171467764, gamma
         =0.95255647698944, lambda=0.3238726864076799, T
         =0.037311385459533594, epsilon=0.008779149519890258,
         alphaDecayRate=1.6000000000000003E-5, TDecayRate=0.0,
         epsilonDecayRate=0.0, numAgents=6, numInputs=82, numOutputs
         =12, memoryLength=0, memoryInterval=15]
```

# Appendix F

# Project Proposal

*A. Braithwaite*
*Fitzwilliam College*
*ab2153*

Computer Science Project Proposal

## Reinforcement Learning Demolition Teams

$23^{\text{rd}}$ October 2015

**Project Supervisor:** Dr S. Holden

**Director of Studies:** Dr R. Harle

**Overseers:** Prof R. Anderson & Prof J. Bacon

## Introduction and Description of the Work

First person shooters are a genre of computer game in which a player views the world from a first person perspective, shooting enemy players, and potentially coordinating with team-mates to achieve some objective. First person shooter games are a very large industry, with games such as Call of Duty: Black Ops II selling over 7.5 million copies in just 11 days [1], meaning improving gameplay for these games could be very useful.

Reinforcement learning is an area of machine learning in which agents observe an environment, perform actions, and receive rewards. The agent learns which actions are preferable in a certain environment to maximise future reward. A particularly useful version of this for video games is online reinforcement learning, in which the agent can learn as it plays, not requiring a game reset to learn.

One issue with first person shooter games is the AI used for the computer controlled players is too often distinguishable from true human players, partly due to deterministic scripted behaviours. This can clearly detract from a players enjoyment of the game, also requiring significant effort from developers to script all the potential behaviours.

Agents teaching themselves to play could remove predictability associated with a pre-scripted AI, alongside reducing the necessity for a team to write the whole script themselves. This could improve the quality of a game and reduce production costs.

Q-learning [10] is one approach to reinforcement learning, having been applied successfully in many domains, including board games [3], video game playing [6, 7, 9, 8], and AI in modelled real world problems [16, 30]. In Q-learning, a Q-function is approximated, telling the agent the expected future reward from taking any action in a given state.

In this project I will aim to use Q-learning to train a team of agents to play both the attacking and defending team effectively in the Demolition game mode featured in some popular first person shooter games.

## Starting Point

I will be using Watkins' Q-learning algorithm [10], modified as described by Rummery and Niranjan [16]. I intend to implement this algorithm myself.

The game will be created from scratch, inspired by the Demolition game mode featured in the games Counter Strike and Call of Duty.

# Substance and Structure of the Project

## Implementation

The aim of this project is to use Q-learning [10] to teach a team of agents to play the Demolition game mode in a simple 2D first person shooter game.

In the Demolition game mode an attacking team is tasked with arming a bomb at a bomb site, whilst the defending team must defend the bomb site for a set time limit. This leads to team work being required for success, with different strategies beneficial for each team.

The project will be carried out in Java. I have experience with Java and I don't feel the efficiency gain from using a lower level language will be justified with the extra development time required.

In the game each agent will be modelled as a circle with a direction. The game map will be symmetrical around the bomb site, including a few obstacles (modelled as lines, blocking movement and shooting) and each team's respawn area. The bomb will initially be placed inside the attacking team's respawn area. When an agent is killed, they will appear at their team's respawn area after a short time.

After the bomb is armed by the attacking team, the defending team can try to defuse it. If successful, the defending team win. If the defending team fail to defuse the bomb in a certain time, the bomb explodes and the attacking team win. If the attacking team fail to arm the bomb within a time limit, the defending team win.

Each agent will have access to basic sensory inputs in the form of an array of sensors. These will give information about the distance to and identity of objects in their field of view, and possibly give extra information about the object seen (e.g. relative bearing). The exact method of giving these inputs to the agents will be experimented with to ensure sufficient simplicity to allow learning at a reasonable rate and enough complexity for reasonable gameplay.

The agents will also have inputs telling them the current state of the bomb (dropped, carried by ally or enemy, and fuse set), the team the agent is on, it's current health, bearing and distance to the bomb and bomb site, and whether the agent is carrying the bomb or not.

The outputs available to the agents will be turn left and right, move forward, backward, left and right, and shoot. Movement is relative to current direction, not the game world, and shooting is in the current direction faced, possibly with a small random alteration.

The bomb can be picked up, armed and defused. These actions will all be performed automatically when an agent stands within a certain distance of the bomb for a sufficient time without shooting. The attacking team agents cannot defuse the bomb and the defending team agents cannot arm the bomb. Shooting or dying whilst carrying the bomb causes it to be dropped.

The algorithm I plan to implement is MCQ-L [16]. Here, the Q-function is approximated by neural networks and updated on-line. I will use 1 artificial neural network for every output, with the maximal output deciding the action to be taken (with some small chance a random action is taken). Rewards in the full game will be given by damaging or killing enemy agents, and winning the game. Negative rewards may be given for shooting, damaging or killing allied agents, being shot, dying, and losing the game.

## Evaluation

For the evaluation of the success of the learning algorithm, initially it is best to test on a simple scenario. The game can be set such that there is only 1 agent with obstacles placed randomly every trial, and the agent is given a reward for reaching a destination. This is similar to the scenario used by Rummery to compare different approaches to Q-learning [16]. From this, training curves can be plotted, showing reward plotted against trial number. An increase in this shows learning has taken place. A comparison can be made with the average reward received by an agent making random moves.

Once this has been tested, analysis can be performed as to whether learning is still successful in more complex environments. The game can be modified to differing complexity levels, e.g. remove the bomb (meaning the only game goal is to damage the enemy), and/or have teams consist of 1 player each (removing any team dynamics from the game). Games should be played against a constant opponent to ensure the results of learning can be seen, so for these games the agents on the opposing team will take a random action every time step.

In each level of complexity a moving average reward can be plotted against time, with an overall increase in this showing the algorithm is successfully optimising the Q-function.

Next, it should be analysed whether this change leads to an improvement in the game-play of the agents. For this, different measurements can be taken and possibly combined. The rate of damage inflicted on enemy agents, rate of damage received by learning agents, rate of winning games and rate of losing games could all be used as indicators for improved gameplay.

The behaviour of the agents after learning has taken place could also be analysed. In the case of TD-Gammon, a Q-learning Backgammon playing agent, this analysis has even lead to new strategies in human Backgammon playing [31]. A learning agent could also compete against a human player (e.g. myself) as a fun example of how it's gameplay can improve.

## Structure

The project has the following main sections:

1. A study of Q-learning and MCQ-L, focusing on the details of implementation.
2. Planning the details of the game.

3. Design of the interfaces between the agents and the game world, ensuring to make it simple enough to allow reasonable learning speeds, but sufficiently complex such that agents are capable of playing effectively.
4. Development of the game, with a simple GUI and a random-action player/team to test the learning agents against.
5. Implementation of the MCQ-L algorithm and integration into the game.
6. Evaluation of single agent learning to find it's way to a destination.
7. Evaluation of learning in different game complexities.
8. Dissertation write-up.

## Success Criterion

The project will be a success if a team of learning agents is able to consistently defeat an opposing team of random-action agents as both the attacking and defending team in the Demolition game mode.

## Possible Extensions

If my main result is achieved early I will then look into:

- Balancing the game fairly between teams (either giving handicaps to one team, or making the map asymmetrical in some way).
- Modifications to the learning algorithm used and effects on learning rate.
- Effect of team size on learning rate.
- Effect of different reward functions on rate of game-play improvement.
- Effect of including a (learning?) commander to coordinate a team of learning agents, where the commands could be sent directly (e.g. extra inputs to each agent), or indirectly (e.g. modifying reward functions, effecting future learning).

## Timetable and Milestones

1. **23rd Oct - 1st Nov** Proposal finished. Study Q-learning and MCQ-L, looking at implementation details. Design game.
2. **2nd Nov - 15th Nov** Implement a simple working neural network. Begin programming of game.

3. **16th Nov - 29th Nov** Produce working version of game. Implement Q-learning demo with simple discrete state space.
4. **30th Nov - 10th Jan** *(Michaelmas holiday)* Implement and test MCQ-L in game. Apply to single agent finding path to destination and produce data from trials.
5. **11th Jan - 29th Jan** Write progress report. Begin to run main trials with simple versions of full game.
6. **30th Jan - 14th Feb** Continue running trials with different complexities of full game. By end of this period the Success Criterion should have been achieved.
7. **15th Feb - 6th March** Analyse learning data using different metrics. Experiment with methods to ensure game is fair between teams. Experiment with modifying learning algorithm and parameters.
8. **7th March - 17th April** *(Easter holiday)* Finish analysis of main learning data. Experiment with a commander and methods to communicate with team. Finish Preparation and Implementation chapters of dissertation. Draft Evaluation chapter.
9. **18th April - 1st May** Evaluate learned behaviour. Complete dissertation.
10. **2nd May - 13th May** Proof reading and then an early submission so as to concentrate on examination revision.

# Resource Declaration

I plan to use my own computer:

- **Processor** AMD A6-4400M APU with Radeon(tm) HD Graphics x 2
- **Memory** 5.4 GiB
- **Operating System** ubuntu 14.04 LTS

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure, which are holding everything under git on Dropbox, with weekly checkpoints to an external HDD.

Should my own computer fail I will continue work on the MCS computers in the Computer Laboratory (and may consider purchasing a new computer).

I require no other special resources.