IMPROVING REINFORCEMENT LEARNING AGENTS

By

PATRICK EVANS

A report submitted in partial fulfillment of
the requirements for the degree of

COMPUTER SCIENCE BSC.

UNIVERSITY OF MANCHESTER
School of Computer Science

MAY 2020

IMPROVING REINFORCEMENT LEARNING AGENTS

Abstract

by Patrick Evans,
University of Manchester
May 2020

: Supervisor: Konstantin Korovin

The need for reinforcement learning implementations to perform as optimally as possible is one that is universally accepted. The methodologies for reinforcement learning are vast and varied and their abilities to converge to an optimal policy also differ greatly. The purpose of this project is to examine concepts that are known to cause changes in performance, to implement these concepts in the context of RL agents to play Tic Tac Toe and compare their efficacy.

Specifically, the concepts that are focused on in this project relate to:

- Differing implementations of the theoretical ĂŸvalue functionĂŹ with increasing approximation e.g. no approximation up to high levels of approximation.
- Different policies for training RL agents

In the end, of the value function implementations, the neural network implementations performed most optimally. In regards to the different training policies, inconclusive results were found with the agents trained using a random policy performing to around the same level as those trained using an epsilon greedy policy or using Monte Carlo Tree Search.

TABLE OF CONTENTS

Part I

Introduction

Chapter One

Introduction

Reinforcement learning is an approach of machine learning and, like other machine learning approaches, is used to produce software agents that learn to complete tasks; often it is used to train agents to play game-like scenarios. However, unlike with other approaches, in reinforcement learning, this learning takes place not by explicit instruction, but instead by making its own actions and learning from the results of these actions.

This is a concept which has been receiving vast amounts of attention recently and has been used to implement everything from chess agents to self driving cars. The ability of an RL to perform its objective as optimally as possible is one that is of utmost importance for these important tasks and is what this report is concerned with.

RL is a concept that lends itself to implementations that are vastly varied, each with vastly varied performance. In this project, experiments will be performed to understand why this differing performance occurs and how we can maximise our performance in RL tasks. These experiments will consist of implementing different agents to play 4x4 Tic Tac Toe, each with different properties.

To this end, 2 main concepts common to RL implementations and how they can influence performance will be focused on. These concepts being: the value function implementation of the agent, as well as the training data used.

This first concept - the value function - can be implemented in myriad different ways and

generally, each implementation alters the ability of the RL agent to be able to 'generalise' based on the states it's previously seen to different extents. This ability is generally referred to how well a value function can be approximated. Implementations that approximate the value function to different extents and how this relates to performance have been studied.

The second concept - the training data - refers to the data that is used to inform an RL agent's understanding of an environment and thus, used to inform its decisions. This training data (unlike that in other types of machine learning) is not produced by any external agents but instead must be only made using our agent's current understanding. The way this training data is produced is known as a training data methodology and the impact of changing the methodology upon our RL agent's performance will also be studied.

For each concept, different implementations have been implemented as agents trained play 4x4 Tic Tac Toe. Experiments are then conducted using these agents and finally, conclusions are drawn concerning how these different concepts can affect the performance of Reinforcement Learning agents. The following section, details the overarching aims of this report and what should be achieved by the end of it.

1.1 Aims and Objectives

By the end of this report I aim to have:

- Thoroughly researched the concept of Reinforcement Learning.
- Researched the concept of value functions and different methodologies for implementing value functions including using No Approximation, Heuristics and Neural Networks.
- Researched different methodologies for training data production in the context of RL including Random data, Epsilon Greedy and Monte Carlo Tree Search.
- Designed experiments for comparing the performance of the different methods of calculating a value function as well as different training data production methodologies.

- Implemented the researched methods for calculating the value function in the scope of 4x4 Tic Tac Toe including using No Approximation, Heuristics and Neural Networks.
- Implemented the different methods for producing different types of training data including Random data, Epsilon Greedy and Monte Carlo Tree Search.
- Run the experiments necessary to find the performance impact each of these implementations has on the RL agents.
- Performed critical analysis to find which implementation of RL provides the best performance.

1.2 State of The Art

A vast number of research papers have had similar focus as this report - attempting to find optimal methodologies for producing Reinforcement Learning agents. For example there have been papers regarding the need for approximation in value-based learning algorithms **[12]** and the methods that can be used for approximating these functions. As well as this, the method of training methodologies and policy improvement has been the subject of attention. This includes the use of epsilon greedy **[14]** Q-learning **[15]** as well as Monte Carlo Tree Search(MCTS) **[13]**

In the context of the value function, neural networks have been used in state of the art Reinforcement Learning implementations and have been shown to perform in a superior way to methods like heuristics. Also, in terms of training data and policy improvement, Monte Carlo Tree Search has been used in state of the art implementations of perfect information game learning agents. Thus, in my experiments, these observations are expected to hold i.e.that neural networks provide the best value function implementation and that MCTS provides the best training methodology.

Part II

Background

Chapter Two

Background

This background details any concepts which are required to understand the experiments undertaken in the report. The section has been split into three separate sections, with each section detailing concepts that are requisite to understanding different parts of the project. First, general RL concepts are explained, then, concepts relating to value function approximation and, finally, concepts relating to training data methodologies in RL. By the end of this chapter, a comprehensive understanding of the concepts discussed in the report should be seen.

2.1 General RL Concepts Background

2.1.1 Reinforcement Learning

This concept is integral to understanding the bulk of this report and is a concept that, at its core, is fairly simple. We are attempting to produce agents which gain an increasingly sophisticated understanding of an environment based purely on its interactions with said environment. This learning is done by:

- Taking actions in the environment and collecting the 'rewards' that result from these actions.

- Using this information to inform future actions.
- Optimising our actions and thus, maximising reward. **[1]**

This algorithm does provide a good basis for understanding RL. However, it is more than slightly reductive as it does not demonstrate the vast technical challenges associated with producing a working RL implementation. For example, how each environment is modelled, how actions are chosen, or even how 'reward' is quantified.

2.1.2 Markov Decision Processes

. In order to study the performance of an RL agent, often, RL has been applied to a specific scenario, and then, performance measured in this scenario **[13]** . However, in order to construct this RL agent, a way to model the situation in terms of the possible actions and states that exist in the environment is needed. This is where a Markov Decision Process (MDP) is useful. In an MDP, we model the decisions that can be made using a mathematical framework. This framework lends itself ideally to the states and actions that exist in situations where we wish to apply RL as we can easily define:

- States in the environment as states in an MDP
- Decisions to be made from states as actions in an MDP
- The probability of making a decision from a state as the transition probability in an MDP.

Crucially, however, in an MDP, each state must be independent of previous states. This is known as the Markov property. Therefore, if we wish to model the situation we are applying RL to, we must ensure it observes this property:

$$P[S_{t+1}|a, S_0, \ldots.., S_t] = P[S_{t+1}|a, S_t]$$

- $P[S_{t+1}|a, S_0, ..., S_t]$ - the probability of state S_{t+1} given action a and state $S_0, ..., S_(t-1), S_t$
- $P[S_{t+1}|a, S_t]$ - the probability of state S_{t+1} given action a and state S_t

2.1.3 Return, Reward Functions and Policies

In our RL agents, at every turn, we will be attempting to achieve the objective of our scenario. In RL, we do this by maximising our reward. The concept of reward(r) dictates the 'goodness' of an action or a state. That is, if an action or state has high reward, it is more 'optimal' than an action or state with low reward. Reinforcement agents thus seek to maximise the sum of future reward. This 'future reward' is often referred to as the return(R). **[2]**

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + ... = \sum_{k=0}^{\infty} r_{t+k+1}$$

- R_t - the total Return from time t .
- r_{t+1} - the reward at time t+1.
- $\sum_{k=0}^{\infty} r_{t+k+1}$ - the summation from r_{t+1} to $r_{t+\infty}$

By focusing on maximising return, we are able to improve our policy which is the way an agent acts within an environment. The concept of a policy and of improving our policy is a key idea in reinforcement learning.

2.1.4 Value Functions

Another prerequisite of the project is a good understanding of what is meant by a theoretical 'value function'. A value function simply returns the 'value' of a state or action, given a specific policy(π). In the context of MDPs, we usually consider there to be two types of value function - the state value function(V) as well as the action value function(Q):

The state value function is simply a calculation of an expected return when we start from initial state s and act within our policy rules:

$$V^{\pi}(s) = \mathbf{E}_{\pi}[R_t|s_t = s]$$

- $V^{\pi}(s)$ - returns the value of a state s given a policy π .
- $\mathbf{E}_{\pi}$ - is expected value given a policy.
- $[R_t|s_t = s]$ - is the Return at t given the state s_t is equal to state s

The action value function, on the other hand, is the calculation of our expected return when we start with an initial state s, take an action a, and then act as our policy has defined:

$$Q^{\pi}(s, a) = \mathbf{E}_{\pi}[R_t|s_t = s, a_t = a]$$

- $Q^{\pi}(s, a)$ - returns the value of a state s and action a given a policy π .
- $\mathbf{E}_{\pi}$ - is expected value given a policy.
- $[R_t|s_t = s, a_t = a]$ - is the Return at t given the state s_t is equal to state s and action a_t is equal to a

For the purpose of this project, when an action's 'value' is referred to, this relates to the output of the action value function when given the current state and the action. Conversely, when a state's value is mentioned, this is referring to the output of the state value function when this state is passed in.

2.2 Background on Value Function Approximation

This section discusses the finer details of the different implementations of value functions that have been implemented in my project.

As previously mentioned in 2.1.4, the value function is a theoretical function that would, given a policy return the expected return of a state. However, when we move to scenarios with large scale, exact calculations of what this so-called 'value function' would return for each state are completely infeasible. It has been necessary therefore to produce methods to approximate this function for these kinds of scenarios. It has been observed in previous studies **[12]** that these approximation techniques have been key to finding policies using Reinforcement Learning that perform well in large scale problems. The extent to which improvements are seen with increasing amounts of approximation is what has been studied in this report. These levels of approximation are represented in increasing amounts by the following:

- Method with no Approximation
- Heuristic approaches
- Neural Network approaches

2.2.1 No Approximation

Simply if value function implementation exhibits no approximation, there will be no generalising of the knowledge gained from the training data. That is, what we learn by seeing a state in the training data will be only applicable in the testing data if we see that exact same state.

2.2.2 Heuristic

A heuristic in the context of learning is where we utilise 'rules of thumb' which, although not optimal, provide a good basis for understanding the problem that is faced. In our example of 4x4 Tic Tac Toe, we wish to use heuristics to find 'indicators' for the value of states. These

heuristic features should be predictors for a win or a loss and should importantly be non state-specific. Thus, when we change our heuristic values, we should be attempting to improve our understanding of every state rather than just one. Due to this property, heuristics are incredibly useful for implementing 'value function approximation' as we can draw inferences from other states to approximate the value of an unrelated one. However, different heuristics will have very different abilities to inform the value of states. It is therefore important that any heuristics is chosen is done so with common sense and forethought. In my experiments, multiple types of heuristics have been implemented to see how each compares.

2.2.3 Neural Network

Neural networks attempt to mimic the function of the human brain and thus, the way that the human brain is able to 'learn'. In neural networks, we model the neurons of the human brain using 'artificial neurons' which each take inputs, pass these through a function and produce an output. When we train a neural network, we pass data which is associated with a 'ground truth' through the layers of the network. We can then adjust the weights of our neurons and therefore ensure that in the future we make better predictions.

In the context of Reinforcement Learning, we are trying to minimise what is known as the loss function. This usually represents the error in our results and thus, minimising it is key to improving our performance. In my agents, the loss function may, for example, be the error in our prediction for the value of a state. Thus, we can cast 'learning' simply as an optimisation task where we minimise error. This learning or optimisation often takes place in the form of a backpropagation gradient descent task.

Backpropagation gradient descent in neural networks is a fairly simple process. It involves the neural network making a prediction of the value and then comparing this value with the expected value. By calculating the gradients of the error, we can adjust our weights and biases in response to this. We aim to make updates to our weights and biases by an

appropriate learning rate every time we see training data to eventually reach a minima where the error is minimised.

What Makes Neural Networks Useful For Value Function Approximation?

The reason why neural networks are so ubiquitous in the field of value function approximation ties directly to the Universal Approximation Theorem. This was proposed by Kurt Hornik in 1991 and states, "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units". This means that, given a suitable learning algorithm and training data, for any value function, even a single hidden layer neural network is able to approximate it.

This property makes neural networks ideal for implementing value functions. If the proper parameters are used and ideal training data is fed in, we should be able to produce the best approximation that is possible for our value function. This is unlike our other approximations which do not exhibit this property.

2.3 Training Data in RL Background Information

This section starts with a basic background to the importance of training data in reinforcement learning. Then it moves onto the methodologies implemented in the project and what is meant by each concept.

2.3.1 Training Data in RL

The concept of Reinforcement Learning is so interesting is the fact that it does not require optimal data (and in some cases 'optimal data' is a hindrance). Also, the sub-optimal data is never explicitly corrected and instead, we use sub-optimal data as a necessity in order to 'explore' the environment.

Thus, in general, we require the data provided to our RL agents to toe the line between exploiting our understanding of the problem (providing more optimal data), while ensuring that exploration (providing less optimal data - ensuring more variety in visited states) occurs. It has been shown in many studies that simply selecting random actions for the training data does generally diminish performance **[10]** - this is following complete exploration. However, if we ignore the concept of exploration, and make our best moves the entire time, we can unwittingly run into the issue of overfitting.

Overfitting often occurs when the data we consider does not capture the entire scope of an environment. By omitting important data, our RL agent can start to draw conclusions which do not hold for all states - drastically curtailing performance. Therefore, it is imperative that we do not ignore exploration completely.

2.3.2 Random Policy

Using a random policy for training data generation is a fairly self-explanatory process. For each piece of training data that is produced, all actions are made completely randomly. This is an implementation that represents a maximum level of exploration while neglecting exploitation of the environment. It is likely therefore that using this policy removes the risk of overfitting.

2.3.3 Epsilon Greedy Policy

Training using an Epsilon Greedy Policy is method that has been proposed to balance the need for exploration with that of exploitation. Unlike in the random policy where all of the training data can be produced before the training takes place. In strategies like epsilon-greedy, the process learning and training are intertwined. That is, we feed the training games into our agent as we are producing the training games. Thus we are making our exploitative moves using our best possible knowledge.

In this method, we define a value of epsilon((ϵ)) which is the probability that given the choice between exploration and exploitation, we choose exploration. When exploitation is chosen, simply the action the agent thinks will give the highest return is chosen and when exploration is chosen, we simply pick an action completely at random. There are many implementations of ϵ-greedy. For example, we could have a fixed or dynamic value of ϵ.

2.3.4 Monte Carlo Tree Search

In the process of MCTS, we carry out the balance between exploration and exploitation slightly differently than in the previous example. We attempt to produce a 'tree' which represents all the possible games and states that we've visited previously. As we train further, we feed our observations into the tree and update the state's values that are in the tree. Compared to the previous agent, here we do not ever take suggestions from the neural network directly. Instead, the tree that we produce is used to produce our training data, we feed our values into the neural network from this tree.

This constant improvement takes place via a continuous process of:

- Selection: Playing out a game until a state with no children is found. We ensure some randomness so no overfitting.
- Expansion: Expanding the tree to include a child of this state
- Simulation: Playing out a game from this child and recording the result.
- Backpropagation: Updating values of states further up the tree using this result

It can be said that as the neural network is learning, the Monte Carlo tree is also gaining an understanding of the problem and helping to maximise our optimal moves in future training games.

Part III

Design

Chapter Three

Choosing 4x4 Tic Tac Toe

In order to conduct fair experiments for our implementations, a scenario needed to be chosen for which training RL agents would allow for useful results to be produced. It was also important to test each agent on the same environment. This common environment could be any number of different scenarios. It was decided that since using a perfect information game (with no chance) allows for easy comparison between agents, this was what was would be used for my project. However, in making the decision of which perfect information game the experiments should be conducted on, a number of requisites need to be discussed to assess their suitability:

3.1 Requirements of the Game

Each of the following requirements represent something that needs to be met before ensuring that a scenario is suitable for our experiments:

- Whether the environment's scale is large enough.
- Whether the scenario allows for the application of various different implementations of value functions.
- Whether it allows for different value functions to be implemented.

- Whether it allows for different training data methodologies to be used
- Whether it allows for performance data to be collected and comparisons to be made

After carefully considering each of these points in turn, that the game of Tic Tac Toe has been chosen as the perfect scenario for this project. Each of my four requirements are discussed here to show why this is:

3.1.1 The Scalability of Tic Tac Toe

A main factor guiding this decision has been looking for a scenario which had the ability to scale up to a suitably large level. This is because scenarios and games that have a very small scale can be sometimes very easily solved by RL regardless of implementation. It is therefore imperative that a complexity is chosen that allows for differences in the performance of agents.

In Tic Tac Toe, by adjusting the size of the grid(e.g. 3x3,4x4 etc.), as well as tweaking the rules, we can easily adjust scale and thus, satisfy this condition. For example, a game of Tic Tac Toe with grid size 3x3 has 19,000 states, while a game at grid size 4x4 has 43 million states. Thus, by increasing the grid size to 4x4, the necessary scale of the environment is satisfied.

3.1.2 Allowing for Different Value Functions to be Implemented

This requirement would likely have been satisfied by any number of different scenarios and 4x4 Tic Tac Toe is no exception. We can simply program separate agents, all with different implementations for their value functions.

3.1.3 Allowing for Different Training Data Methodologies to be Used

Again, Tic Tac Toe is perfectly suitable for testing different training data production methods. We can simply program separate methods for producing the training data and feed this data into the reinforcement learning agent.

3.1.4 Allowing for Performance Data to Be Collected and Comparisons to be Made

As discussed previously in 3.0, as this is a perfect information 2 player game, this lends itself to direct comparisons to be made between RL agents. The performance data collection can be as simple as counting the number of wins/losses in each playout and thus, making conclusions from these results.

3.2 The Game's Ruleset

After deciding on 4x4 Tic Tac Toe, the first thing to define will be the actual rules of the game our agent will be playing. These will be defined as:

- A two-player game with one playing as crosses, the other noughts.
- Each player in turn plays a piece on the 4x4 grid with crosses going first.
- If a player gets 4 of their pieces in a row, then they have won
- If there are no spaces on the grid left, the game is drawn

3.3 Applying Markov Decision Processes to Tic Tac Toe

A first step of planning the implementation of an RL agent to play Tic Tac Toe can be defining the states and actions of Tic Tac Toe in terms of an MDP. This, for example, can show how a 'states' and 'actions' are related in the game considered. For simplicity's sake, this is shown in this model in terms of a Tic Tac Toe game with a 3x3 grid, although the same ideas apply to 4x4 Tic Tac Toe:

3.3.1 States

A state in my game of Tic Tac Toe is defined as simply a position on the game board. For example, every game will start with the same empty board state:

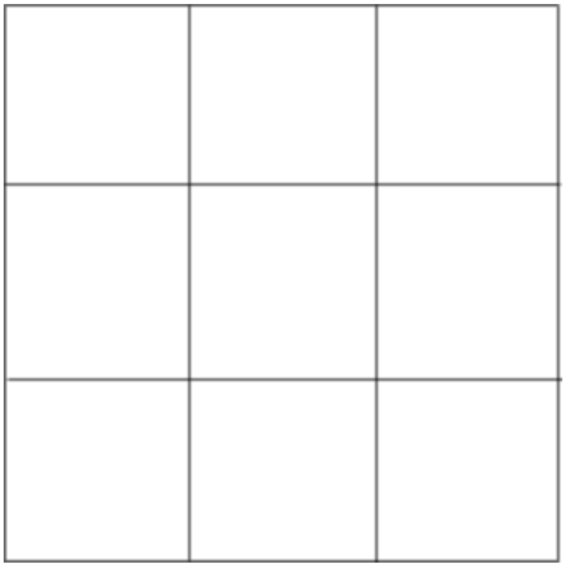

Figure 3.1 The empty state

When any move is made, we will have moved into a new state which is only dependent on the previous state and the action made as show in Figure 3.2. This means that the states of Tic Tac Toe exhibit the markov property.

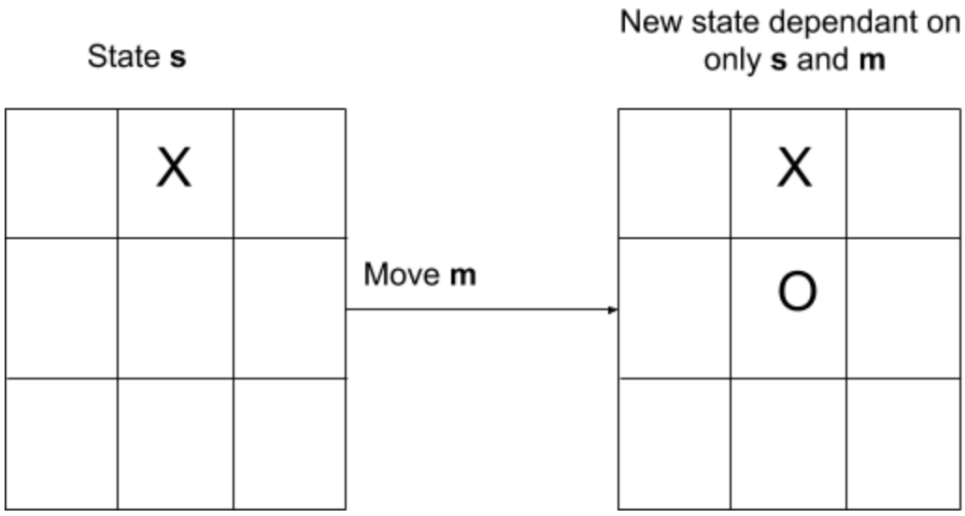

Figure 3.2 How the markov property applies to Tic Tac Toe

3.3.2 Actions

The actions available from each state of the game will be defined as simply the possible moves available from that state. This is shown in Figure 3.3

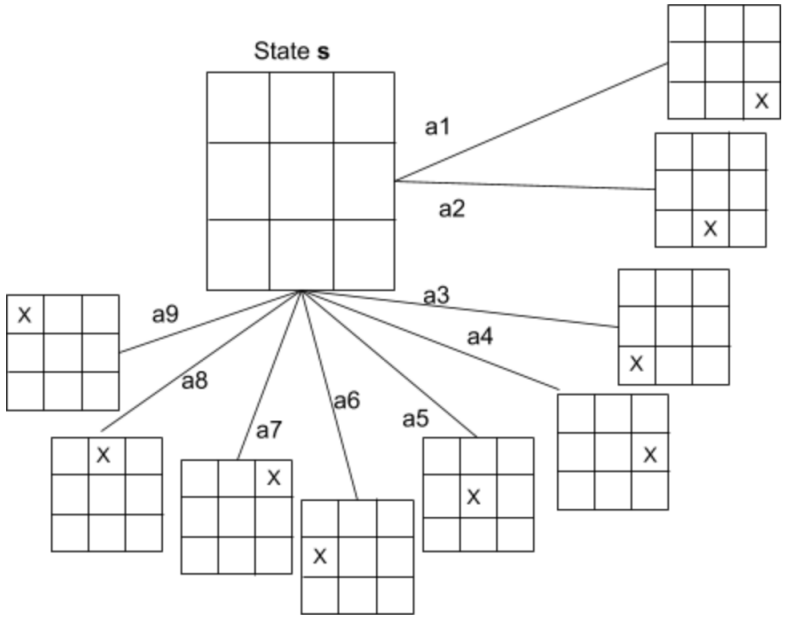

Figure 3.3 Defining Actions in Tic Tac Toe

3.3.3 Training to Play Tic Tac Toe

As has been discussed in 2.1.1, the training of an RL agent in any scenario requires states and actions data along with some kind of 'reward' data. In the scenario of Tic Tac Toe, the training data will be made up of the states seen in complete games of tic tac toe along with the results of those games. By passing this data into an RL agent, the ability of the agent to play the game should continuously be improved.

Now that this scenario has been chosen and planned for, how it can be used to produce useful data is now prudent. This is seen in my experimental design:

Chapter Four

Experimental Design

This experimental design section of the project is concerned with creating experiments that are able to be used to capture the data I'll require to make cogent conclusions regarding my aims. The general experimental design makes up the bulk of this section. This design is what that is needed to ensure valid experiments occur and that useful results are gained.

4.1 How Can Performance Be Measured?

Performance will be quantified in this project by how well each agent performs in relation to other RL implementations. Each time the implementation of a certain element of my agent is changed, this trained agent will be saved and therefore be able to use this to compare against others I've created. Since the perfect information game, 4x4 Tic Tac Toe is used, these games have no chance element and head-to-head results can be easily compared.

More specifically, at the end of the implementation stage, a round robin of all the agents will be performed to compare their performance. A vast number of games will be played between each of the agents (around 10,000). We can then record the number of wins, losses and draws between the trained agents and thus, generate comparable results.

4.2 How the Games will Be Played Out

As has been have mentioned in 4.1, the data used to inform my conclusion will be based upon direct play outs between agents. However, in these play outs a number of assurances should be in place in order to ensure that fair results are seen.

For example in the case of a Tic Tac Toe game, each agent should play an equal number of games on each side of the board. Also, there should be randomized starting states to ensure a variety of games are played. The number of games to be played should also be very substantial - I have decided 10,000.

4.3 Control Variables

It is important to remember that we are attempting to conduct experiments that remove all biases possible. To ensure this, a number of variables need to be control variables (they should not be changed regardless of experiment). For example, all the agents will be trained using the exact same number of training games - 60,000. This number was chosen due to technical constraints as any substantially larger number of games would cause great slow downs in many of the experiments.

Also, for each concept we are attempting to study, this concept will be the only aspect of each agent that is changed. For example, if a new agent is created to represent changing the value function, the training data that is passed into the new agent, as well as all of its other aspects will be identical to the previous.

4.4 Which Experiments Will Be Conducted

The experiments that will be taken out will be split into 2 separate categories. One set of experiments will concern the impact of changing the implementation of value functions. The other category of experiments will concern the impact of differing the methodology for

producing training data on performance.

4.5 Value Function Experiments

For these experiments, all the implementations concerning value functions discussed in the previous 'Background' section will be covered. These implementations being:

- Value Function with No Approximation
- Value Function Using Heuristics
- Value Function Using Neural Networks

For each of these concepts, at least one agent will be created to represent it. These against are to be tested against each other in order to conclude which provides the best performance.

4.6 Training Data Methodology Experiments

Again, these experiments will be used to see how performance can be changed by altering this variable. Multiple methodologies will be considered along with a control 'supervised learning' methodology to act as a benchmark for the other implementations:

- Randomly Generated Data
- Epsilon Greedy Implementation
- Monte Carlo Tree Search Implementation
- Supervised Learning Implementation

Chapter Five

Value Function Experiments System Design

In the previous section concerning experimental design, all the experiments that will be undertaken to collect the data required have been detailed. However, the listing of these experiments does not capture the vast number of technical decisions that are required to actually produce agents that represent each of these concepts listed. This has been split into two chapters: The system design for the value function experiments, and the next, the system design for the training data experiments.

Here, for each value function experiment, the implementation design decisions that needed to be taken before the implementations could take place are shown. This starts with the design decisions required for the first implementation - for the value function with no approximation and move through the other, more complex ones.

However, the first decision to be made affects all of the agents implemented in this section. This is, which training data methodology should be used. It was decided that simply, a random policy would be used to train each of these agents due to the simplicity in implementing such a policy.

5.1 Value Function With No Approximation Design

As has been discussed previously in 2.2.1, when we say an implementation exhibits no approximation, we mean that any information gained from a state seen in our training data will not be generalised for any other state. This concept could, for example, be implemented by storing every single state possible in a data structure along with their respective value. When we see a state in our training data, this will be associated with a ground truth indicating that this state occured in a win, loss or draw. We would then update this state's value in our data structure.

For the data structure, it seems sensible that some kind of 'Map' would be used with the state being the key and the value being the state value. This ensures very fast O(1) access time to the value, given the state.

We do expect, however, that since only 60,000 games are being considered and 4x4 Tic Tac Toe has 43 million states, a great number of states will not be included in the Map and others will have limited information.

5.2 Heuristic Design

Pre-implementation design decisions were also required for the heuristics that are to represent this concept. That is, even basic outlines of the heuristics to be implemented haven't been outlined as of yet. Here how these heuristics are chosen and the design decisions behind them are shown.

The first step in designing heuristics to approximate our value function is to simply look for features of a state that may inform a win or a loss. There is no exact science to finding useful features. I decided to look to heuristics for other similar games to inform my decisions.

In computer chess for example, heuristics can be used as a way of 'pruning' the searches that take place and thus, improving performance. One example of such heuristics, the piece's

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

Figure 5.1 Piece-Square table for a Bishop in chess

positioning on the board can be used to quickly tell how useful that piece is - these are known as Piece-Square tables **[4]** It seems sensible that this kind of heuristic would also be useful in the context of Tic Tac Toe. To investigate this, a similar method where the placement of noughts and crosses on the board correspond with the value will be implemented - these have been labeled my 'simple heuristics'.

Also, as has been mentioned in 2.2.2, heuristics can take a number of forms and complexities. Thus, in order to ensure that my heuristic approaches represent a fair reflection of the concept of heuristics, these heuristics should be compared with another, more complex heuristic implementation. This time, the patterns on the board should be taken into account rather than just piece placement on its own.

In order to do this, each 3x3 section of the grid should be considered and evaluated separately. Then, for the 4x4 grid, these individual evaluations can be combined to produce an evaluation for the entire grid. This acts as a more sophisticated version of the previous heuristic. These will be my 'complex heuristics'.

This decision is built upon the fact that in the game of 3x3 Tic Tac Toe, a much better understanding of the states can be built up using only 60,000 games when compared to our 4x4 Tic Tac Toe. Therefore, hopefully, if we break our 4x4 grid into combinations of 3x3 grids, and then evaluate each of these grids individually, we could theoretically produce a

better understanding than if we took the grid as a whole. This is shown in the following graphic:

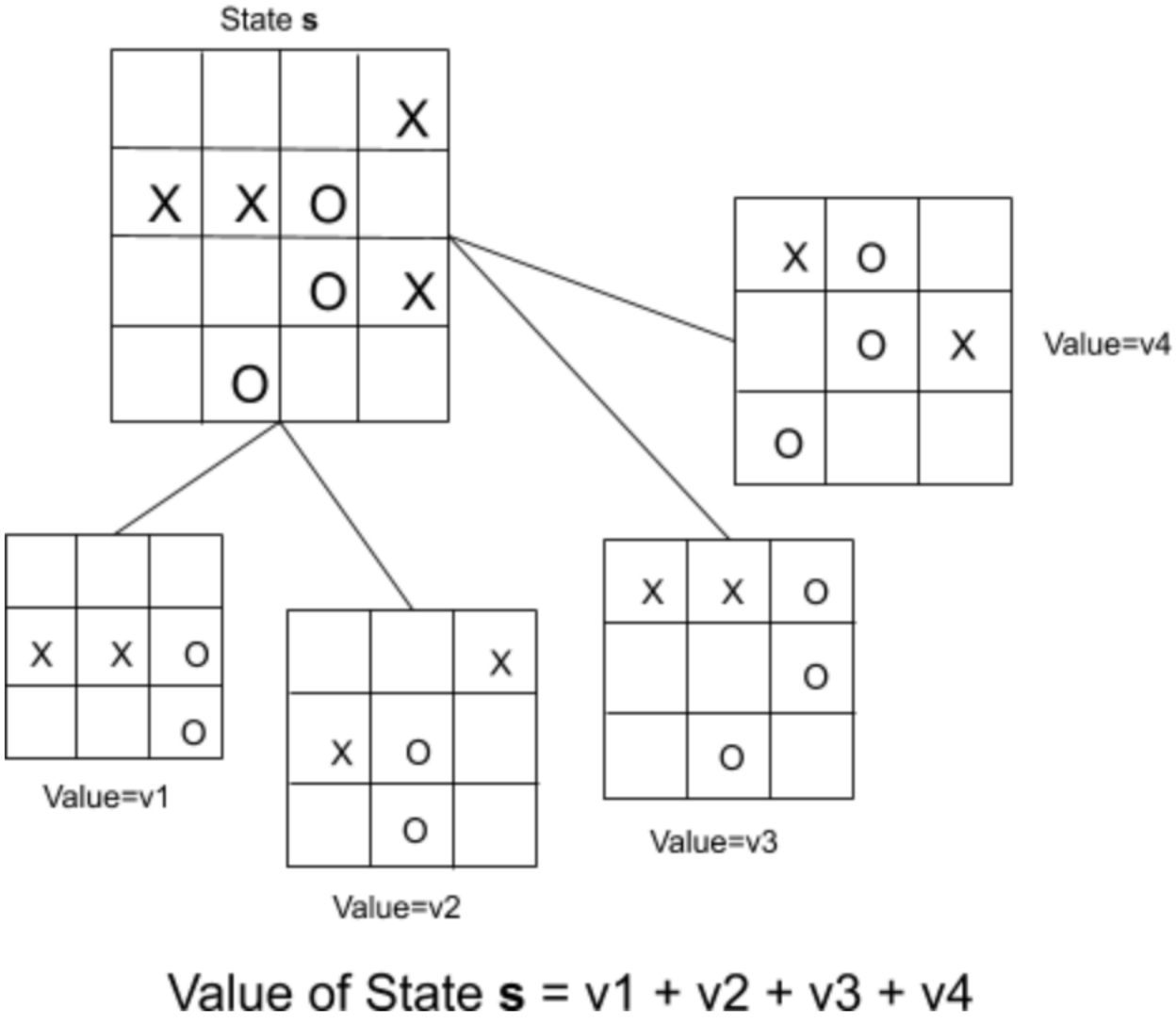

Figure 5.2 Complex Heuristics Value Calculations

Each of these heuristic methods for value function approximation will be implemented. Then their performance will be compared against each other as well as against the other agents implemented.

5.3 Neural Network Design

A neural network is an extremely complex concept. Thus, it is no real surprise that a vast number of design considerations were required before the implementation of the neural network. Each of these considerations are individually extremely important and are needed to ensure that the network's performance is as optimal as possible. The decisions required are listed here and have been discussed on by one:

- Number of layers and hidden layers
- Types of layers
- Activation Function
- Type of optimizer function
- Loss Function

5.3.1 Numbers of Layers

The Universal Approximation Theorem **[16]** questions the number of layers we should use in our neural networks. Since this theorem states that deep networks with multiple hidden layers are no more expressive than shallow ones, does it make sense to only have single layer networks?

No, the answer to this statement is that although they exhibit the same expressiveness, if an appropriate number of layers are used, deep networks can often represent functions using far fewer nodes than shallow networks. This can greatly reduce memory usage and can improve computation efficiency. **[5]** In most areas, deep networks are able to outperform shallow ones when representing the same function.

Thus, I have decided that my network will be a deep one with 1 input layer, 1 output layer as well as 2 hidden layers. This will likely strike a balance between the efficiency of the model

without having too many layers and nodes - something which could hinder performance.

5.3.2 Types of Layers

After deciding how many different layers and hidden layers we should have, we have only really defined the basic structure of our nn model. In order to actually define how these layers will interact and 'work', we need to make decisions regarding the 'type' of layers in the network. These possible 'types' include but are not limited to:

- Dense Layers: A standard layer in a neural network model. Each neuron in the layer receives input from every node in the previous layer.
- Pooling Layers: These layers are used to reduce the number of parameters in the network by downsizing the size of outputs of previous layers. Often used in CNNs(Convolutional Neural Networks) to reduce the size of representations. These help to improve performance.
- Flatten Layers: This works in conjunction with the pooling layer to produce a form that can be passed to a dense layer.
- Convolutional Layers: These layers are often used for improving neural network performance in visual tasks. Performs 'convolutions' on the input data to help to extract features and then pass on to the other layers in the network. By doing these convolutions, we can help to reduce the number of parameters and thus, improve performance.

In my model, trial and error has been used to finalize the exact structure of the model as there is no exact science to which work in which scenario. However, in the end I decided to use: 1 input layer, 1 output layer,as well as 2 dense layers. These layers were also combined with 2 convolutional layers which theoretically can be used to extract features and patterns that may not otherwise have been captured. A basic structure is shown here:

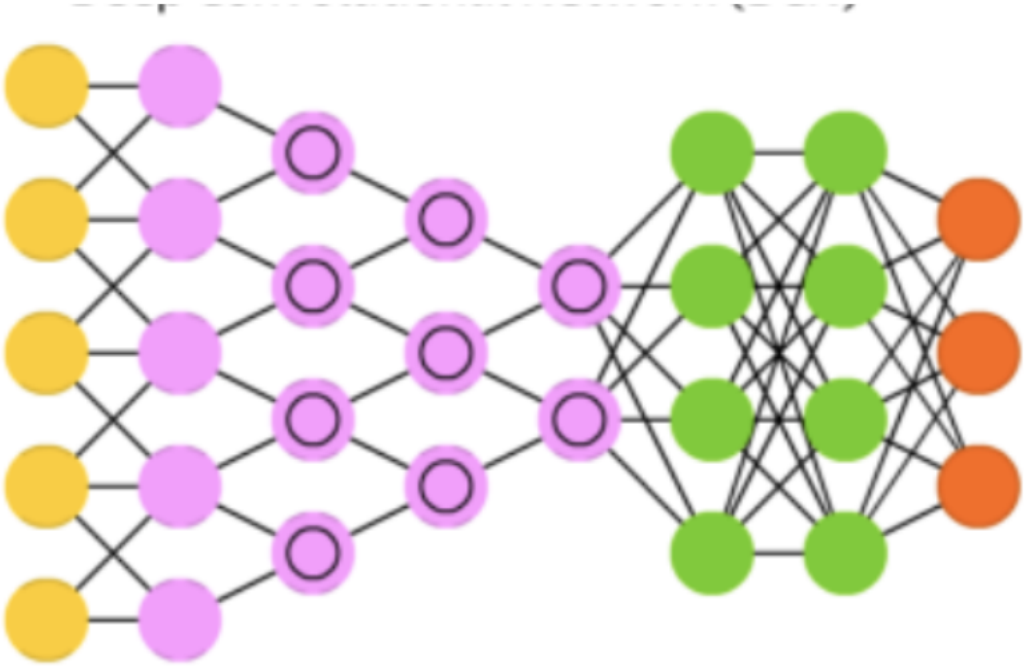

Figure 5.3 Neural Network Structure

5.3.3 Activation Function

The activation function is another concept that is important in neural network learning. This is a function that the output for each neuron is passed through and determines whether or not the neuron should be 'fired'. As well as this, they also act to normalize the output. There are numerous different kinds and they all have benefits and disadvantages and choosing the right one can be a difficult decision.

A ReLu or Rectified Linear Unit activation function has been used in my Neural Network. There were a number of different activation functions to choose from including the sigmoid and tanh. However, there is strong evidence for ReLu outperforming these functions in the context of gradient descent by a factor of 6 and thus, the Relu function was chosen over the others. **[6]**

It has been suggested that the reason why this increase in performance is seen is due to the actual function used. That is, it performs $R(z) = max(0, z)$, therefore, when a negative

value is passed in, it simply sets the output to 0. This aspect of the function causes a fewer number of neurons to be fired. Therefore, improving efficiency.

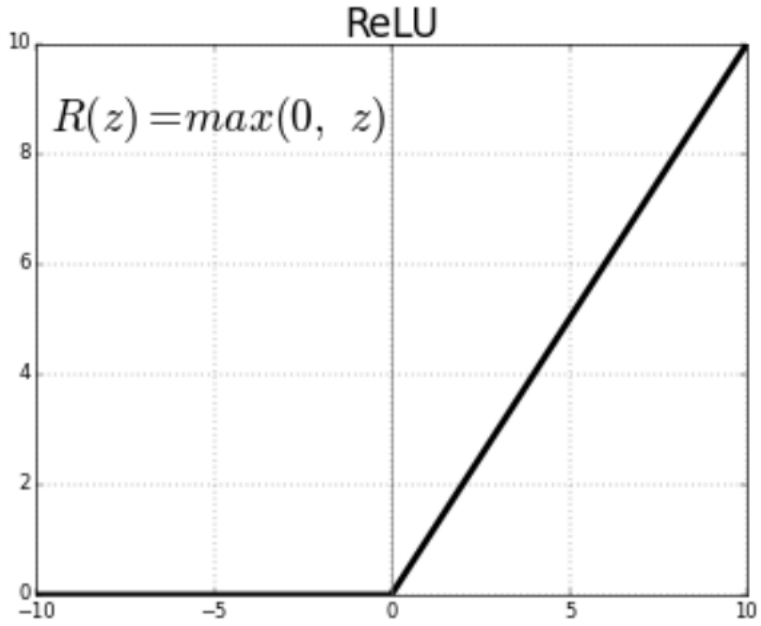

Figure 5.4 ReLu Activation Function

5.3.4 Optimizer Function

The basis of all optimizer functions is what is known as stochastic gradient descent **[7]** This is used to update the weights and biases in the neural network in order to effectively lower the error in our loss function. In the context of neural network training, stochastic gradient descent is performed using backpropagation. An equation for this gradient descent optimizer is shown here:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x, y)$$

- θ - represents the weights and biases to be updated in the gradient descent operation.
- η - is the learning rate

- J - is the loss function
- $\nabla_\theta(\theta; x, y)$ - is the gradient of the loss function with the given when the weight θ is passed along with the training example x and the expected result y.

This equation shows that we update our weights in the neural network by subtracting the gradient (error) times the learning rate. This means that if we have a large error, we will make larger changes than if we had a small error. However, it is not an optimal method in terms of the time to converge to the optimal value. How can this be improved?

Momentum

Momentum is a feature of gradient descent algorithms which is added to reduce the time to convergence. It does this by taking into account the values of the changes made in previous updates and using these to indicate whether a larger or smaller update should be made. **[8]** This means that if many updates all change in the same way, we assume the minimum is a long way away in this direction. Thus, we should theoretically get to the minimum in fewer updates. The equation is shown below and has the extra terms: γ(momentum) as well as υ_t (the last update value):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) + \gamma \upsilon_t$$

This momentum does offer improvements over normal stochastic gradient descent. However, it is possible that by using momentum we can 'overshoot' out minima. We can remedy this issue and even further improve our performance by using this momentum feature along with an adaptive learning rate.

This adaptive learning rate - as the name suggests - is where the learning rate adjusts to be able to avoid overstepping the minimum. This could be done for example, by starting out with large steps and then reducing the size of the steps as we get closer to the minimum. Thus, we remove the main issue with the momentum method.

One example of an optimizer function that contains these features is the Adam optimizer **[9]** This is the one that will be used for my neural network.

5.3.5 Loss Function

After previously describing the function of the loss function, it was fairly simple to apply this to my Tic Tac Toe environment. That is, the mean squared error will be used as the loss function with a win as value +1, draw as a 0 and a loss as a -1. The model would predict the value and try to minimize its error.

After deciding my loss function, the complete structure of my neural network model has been defined, this Neural Network can then be implemented using tensorflow and keras:

Chapter Six

Training Data Experiments System Design

Now, the design discussions are focused on the methods of training data production and policy improvement. For each of these experiments, the only change will be the training policy. The first decision that needed to be made was common to all the implementations in this section and concerned which value function implementation was to be used. The decision was made to use a neural network for each of these due to simplicity and flexibility of this approximator.

6.1 Randomly Generated Data

For the randomly generated data, very little design needs to be taken into account. Simply, 60,000 games of Tic Tac Toe can be produced, the results stored and the RL agent could then be trained using these games.

6.2 Epsilon Greedy Policy

As has been discussedin 2.3.3, an epsilon greedy strategy is when the best action is chosen with some probability, while a completely random action is chosen with another. The

strategy I've chosen is a slight variation of this algorithm.

When exploration is chosen, actions are still chosen at random. However, when exploitation is chosen, a probability that the second or third best move is chosen is introduced; depending on how close in value these actions are to the best action. For example, if the second move is much worse than the first, we will hardly ever choose this move, however, if they are very close, there is some probability that this is chosen. I reasoned that this would allow for more exploitation without the risk of overfitting.

Many programming languages allow this kind of implementation to be done fairly simply i.e. where you can feed in probabilities and with these probabilities, pick each move in the ratio of these probabilities.

6.3 Monte Carlo Tree Search Policy

A basic overview of MCTS is seen in the background section. Thus, the design decisions to be made in this section of the report relate to the finer details of how we intend to actually implement MCTS in our current context. The continuous process of selection, expansion, simulation and backpropagation are still relevant to this context and more detail about how they are to be implemented is shown here:

6.3.1 Selection:

In selection we start at the root node, in the case of my Tic Tac Toe agent, this root node would be the empty game state. Then, from this state, we transition down the tree by making moves and attempting to move toward the 'most promising' state. This finding of the 'most promising' states in the tree is the main goal of MCTS and a 'promising' state is decided based on a number of factors. For example, how many wins have occurred in previous games from this state, how many times this state has been visited and so on. Using these variables, an educated decision is made about the value of the node in comparison to

other possible nodes.

We therefore 'select' these moves until we get to a point where we reach a leaf node. This is a node such that it has not been reached by any previous selection and simulation iteration. Once we have reached this node, we then perform what is known as an expansion.

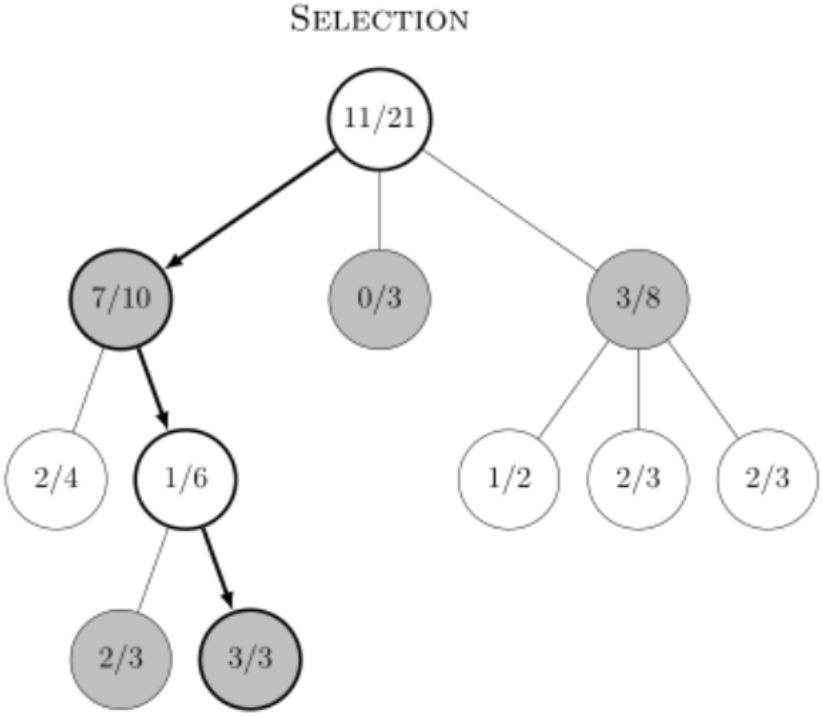

Figure 6.1 Selection Step of MCTS

6.3.2 Expansion

An expansion is the simplest of all 4 steps in our MCTS process. It simply dictates that when we reach a leaf node that hasn't been visited before, we expand the tree to include a child of this leaf node. A child node can be any node that results from a valid action from this current leaf node. This allows for future selection to this node and thus, expansion. We

then select this child and move onto simulation to give an idea for how good this state is.

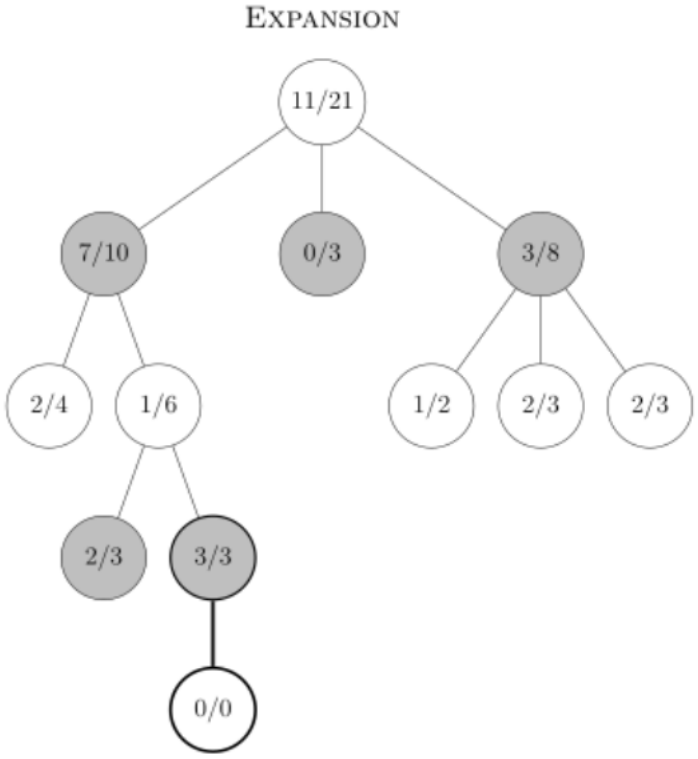

Figure 6.2 Expansion Step of MCTS

6.3.3 Simulation

The simulation stage is how we gather the information about the value of the state within the tree. We can choose to simulate 1 or more 'playouts' from this state which can be generated simply by playing random moves from this point until a terminal state is reached. We then record the results of these playouts in the node as the value ie. how many wins were seen compared to how many losses. We can also increment the visited number on this node.

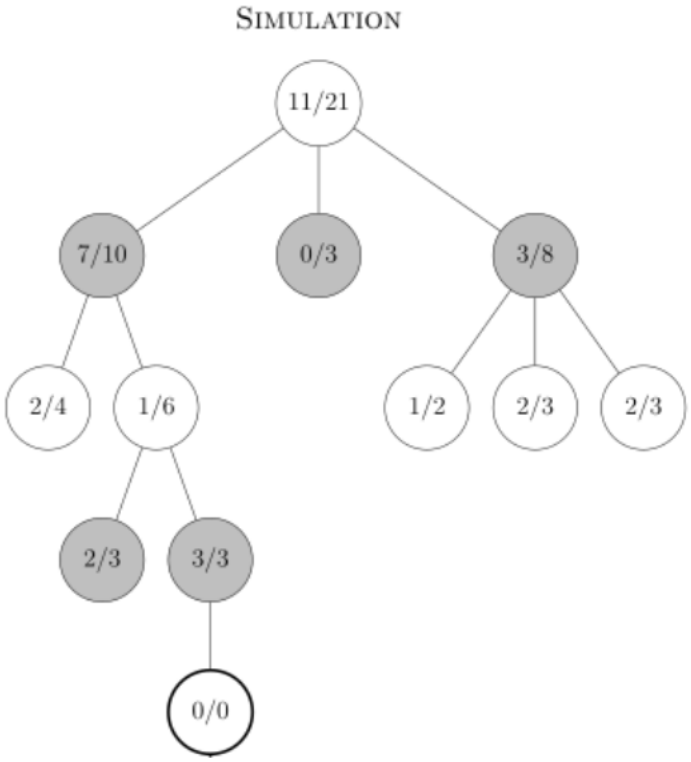

Figure 6.3 Simulation Step of MCTS

6.3.4 Backpropagation

After completing the simulation stage, the next step is backpropagation. This allows the simulations to be able to impact decisions further up the tree. Simply, once the results from the play outs from the expanded node have been calculated, we update the values of the nodes further up the tree as shown. This will thus, influence future selection stages:

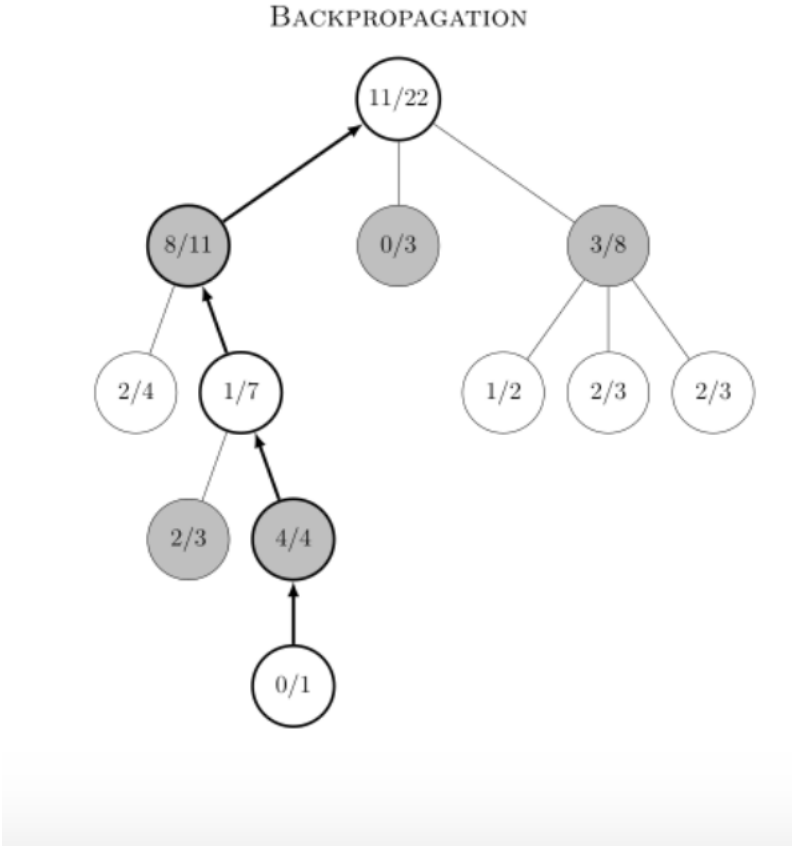

Figure 6.4 Backpropagation Step of MCTS

Each of these Selection, Expansion, Simulation and Backpropagation steps produces a game that can be added to our training data. Thus, as we complete more and more iterations,newer training games will be more optimal than older ones.

6.4 Supervised Learning Data

This supervised methodology will be used as a benchmark for the performance of the RL agents implemented in this section. The supervised method showcases training the RL agent with the 'best possible' data. In this case, the RL agent will be trained with the training samples being generated using a minimax algorithm at 2 move depth.

A number of alterations, however, do need to be made in order to ensure that, as we discussed, the amount of 'exploitation' does not exceed a threshold that would cause overfitting. This is a major issue with these kinds of supervised learning implementations and does require clever methods in order to avoid while keeping the benefits of the good quality training data.

In order to avoid this overfitting, I've decided to take on a 2 pronged approach. That is:

- Ensuring that the starting positions are completely randomised. Thus, providing a starting point for variety in games.
- Also, when moves are fairly similar in value according to the min-max algorithm, simply pick any of these moves at random.

These measures would reduce the chances of overfitting and allow the RL agent to perform to its maximum potential. After figuring out how to reduce overfitting, focus was now moved to actually designing this algorithm:

6.4.1 Generating the Supervised Moves

As has been stated previously, this training data will be generated by a 'minimax' algorithm at depth 2. But what does this minimax algorithm actually achieve and how?

To start with, we think of value as a spectrum going from - to +, with one of the players hoping to get the greatest positive value and the other attempting to get the greatest negative value. The player hoping to get a positive value is known as the maximising player and the other is known as the minimising player.

We go to a certain depth, and we 'pretend' that the minimising player will minimise as much as possible, and the maximising player will maximise as much as possible. By doing this, we find the value of a state after playing out for a certain depth, given each player plays completely optimally.

Unless we have gone deep enough to reach a terminal state, we don't have definitive values for that state. Therefore, we sometimes use heuristics in this algorithm to approximate values. However, in my implementation, I have decided against this. This is due to the fact that it will ensure that the optimal actions it decides to pick are actually optimal. Also, by not using heuristics, there will likely be more states in the algorithm with similar or the same values. Thus, we can use this to improve our exploration of the environment in our training data generation algorithm.

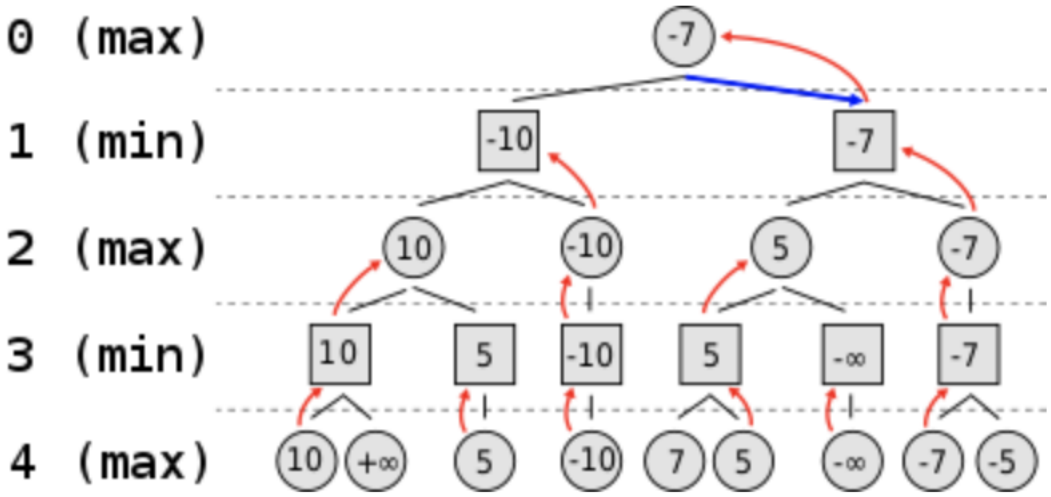

Figure 6.5 Minimax Algorithm

Part IV

Implementation

Chapter Seven

Tools and Technologies

Before implementation, I would like to outline which languages and tools have been used to undertake my implementations. That is, python has been used as the primary programming language. This decision was made primarily due to the availability of certain libraries and modules that are not available in other languages. For example, later in my project, TensorFlow and Keras have been used for neural network work which are both fully supported in python but have more limited support in other languages.

Chapter Eight

Implementing Value Function Experiments

This section details the process of implementing the planned agents for my value function experiments including any algorithms that were pertinent to their implementation. For each implementation, an algorithm is shown for how training data is used to train the agent, as well as how the value of a state can be returned.

The first algorithm shown here is the one that shows how training data is generated. This is an algorithm common to all the implementations shown here.

8.1 Training Data Methodology

As has been mentioned previously in 5.0, games generated using random moves will be used for all the experiments pertaining to value functions. For each randomly generated game, the result of the game will be calculated and then associated as the 'reward' for all the previous

states in the game. This process is shown in the following algorithm:

Algorithm 1: Producing Training Data Using Random Policy

```
Result: 60,000 randomly generated Tic Tac Toe Games
gameNum=0;
games=[];
while gameNum<60,000 do
    state= the empty state ;
    gameStates=[];
    player=cross;
    while the game has not been won and a square is available do
        Play a random move to state as player;
        gameStates.append(state);
        player=!player;
    end
    result=result of game;
    games.append(gameStates,result);
    gameNum++;
end
```

8.2 No Approximation Implementation

This implementation will exhibit no value approximation. Thus, for each state in our training data, we will update only the single value corresponding to this state in the 'Map' data structure. Here, how the training data is used by the agent to train the value function is seen, as well as how testing occurs with a trained agent:

8.2.1 Training the Value Function

When we train in Reinforcement Learning, we aim to ensure that the states that are good for crosses have high positive values. On the other hand, states that are good for noughts, have large negative values. In this case, if a state is associated with a crosses win - we increase its value by our 'learning rate'(0.1). If it results in a crosses loss, we decrease the state's value by the learning rate. We would associate the state with its value using a Map as discussed previously:

Algorithm 2: Training No Approx. provided array of states with a result

```
gameStates is an array of x states with a single result;
stateMap is a Map of all states with an associated value;
for each state, result in gameStates do
    if result is a cross win then
        stateMap(state)+=0.1
    else
        stateMap(state)-=0.1
    end
end
```

8.2.2 Testing the Value Function

Since we have stored each state's value in a Map along with the state, finding the value of a state using our value function is very simple. For example, if we implemented our table as a map, we could simply input the state as the key and we would obtain the value. As explained, in the following algorithm, we maximise value for crosses and we minimise value

for noughts:

Algorithm 3: Finding best state using my No. Approx implementation

```
states is an array of possible states;
stateMap is a Map of all states with its associated value;
bestState=states[0];
player= type of player to be considered i.e. cross or nought ;
for states[i]; i= 1 to n do
    if player==crosses then
        if stateMap(states[i]>stateMap[maxState]) then
            bestState=states[i]
        else
    else
        if stateMap(states[i]<stateMap[maxState]) then
            bestState=states[i]
        else
    end
end
```

As has been described in my experimental design , to test our value function, we test the trained agent's performance against other agents. This previous algorithm is to be used whenever this agent needs to make a move. The move state returned will be the move to be made.

8.3 Simple Heuristics Implementation

As has been said previously, these simple features will be associated with the Piece-Square heuristics that are often used in computer chess. Instead of finding the 'value' of each state, the value of each square in the grid is to be found. When the agent plays a game, it will then prioritise squares with a high value.

First, before these heuristics are actually implemented, the training data that is to be used for the training must be considered. This requires some changes compared to other agents. For example, since the pieces on the grid never move, if we consider only the final state of each game, all other states in a game are somewhat irrelevant. Thus, as is seen in the following algorithm, when the gameStates are passed in, only the final states are used to produce my final weightings.

8.3.1 Training using Simple Heuristics

Algorithm 4: Training using Simple Heuristics

```
gameStates is an array of x states each with a result;
positionTable is a 4x4 array, each element representing a grid position;
finalState is the end state of gameStates;
if result is a cross win then
    for each position in finalState do
        if position contains a cross then
            positionTable[position]+=0.1
        end
        if position contains a noughts then
            positionTable[position]-=0.1
        end
    end
end
if result is a noughts win then
    for each position in finalState do
        if position contains a noughts then
            positionTable[position]+=0.1
        end
        if position contains a cross then
            positionTable[position]-=0.1
        end
    end
end
```

8.3.2 Finding Value of a State using Simple Heuristics

Algorithm 5: Finding Value of a State using Simple Heuristics

```
state is the state we want to find value of;
value=0 for each position in state grid do
    if value=value+positionTable[position] then position's contents==player
    end position's contents== !player
    value=value-positionTable[position]
end
```

In this algorithm, only the method for finding the value of a specific state is shown to reduce redundancy. In this methodology, for both crosses and noughts, value is maximised.

8.4 Complex Heuristics Implementation

These complex heuristics build upon the previous ones. However, instead of simply considering the placement of the pieces on the grid, it utilises the concept of combinations of pieces on the grid i.e. pattern arrangement.

As in the no approximation implementation, the Map data structure will be used to store the data that informs our evaluation. However, unlike in the no Approx algorithm, this Map will store 3x3 patterns rather than complete 4x4 states.

Also, simply randomly generated games should be used as the training data. However, unlike in the simple heuristic methodology where only the final state is necessary to consider, in this method, each state is necessary to inform our value function. The following algorithm shows how this training occurs:

8.4.1 Training using Complex Heuristics

Algorithm 6: Algorithm For Training Using Complex Heuristics

```
gameStates is an array of x states each with a result;
patternMap is a map of all possible 3x3 patterns along withe their value;
for each state,result in gameStates do
    for each pattern in state do
        if result=cross win then
            patternMap(pattern)+=0.1;
        end
        if result=noughts win then
            patternMap(pattern)-=0.1;
        end
    end
end
```

8.5 Complex Heuristics Implementation

8.5.1 Finding Value of a State using Complex Heuristics

This algorithm will be used in any testing that occurs when we wish to find the evaluate of a state. For the first time, in this algorithm, we designate a maximising player (crosses) and a minimising player (noughts). This takes advantage of the property of a zero-sum game

where the value for one player is inverse the value for the other:

Algorithm 7: Finding Value of a State using Complex Heuristics

```
state is the state we wish to find the value of;
value=0;
patternMap is a map of all possible 3x3 patterns along withe their value;
for each pattern in state do
    value=value+patternMap(pattern);
end
```

8.6 Neural Network Implementation

The main complexity in this section came with creating the neural network keras model and defining all the properties that had been outlined in my system design. Once this has taken place, the neural network can be used like other implementations to train an RL agent. This process is shown in the following algorithms:

8.6.1 Algorithm For Training Using Neural Network

The training data for the neural network will be pretty much exactly the same as for the previous implementations i.e. 60,000 randomly generated games. However, due to the flexibility of the neural network, we are able to pass in other features. I decided that I would take advantage of this and, with each state, associate the player to move with it (represented in blue in the following diagram):

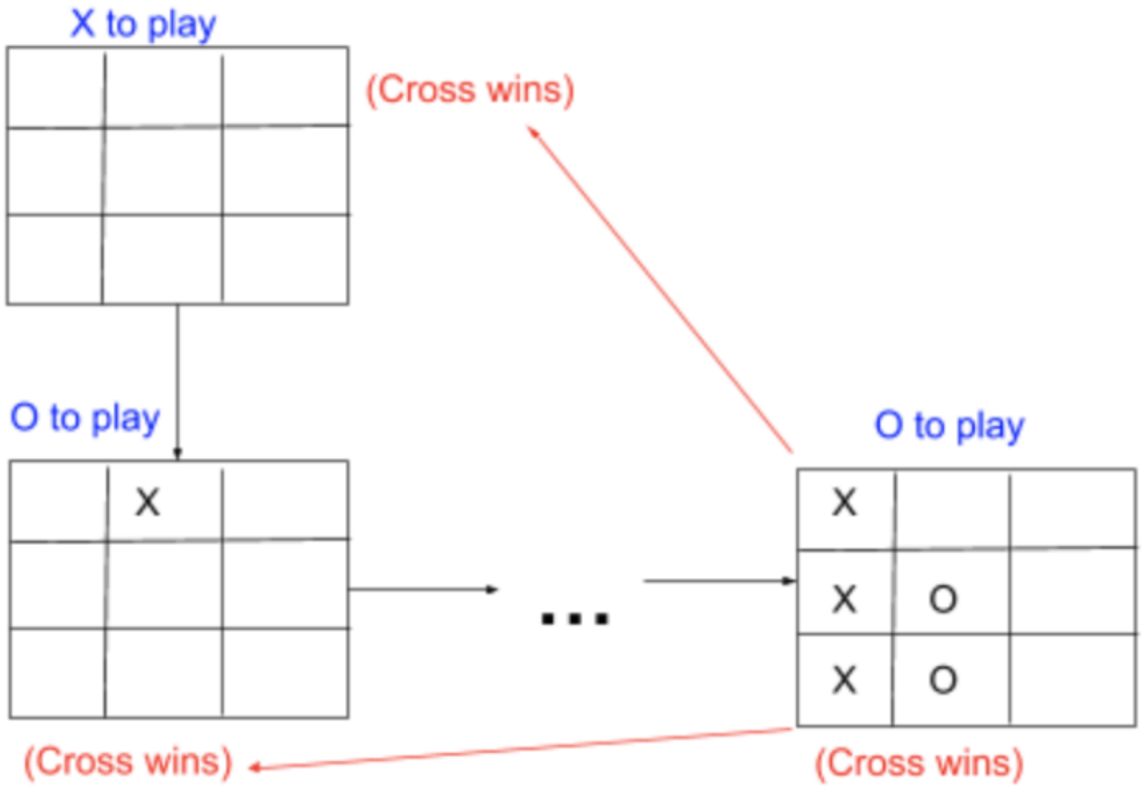

Figure 8.1 Training Data Passed into Neural Network

Algorithm 8: Algorithm For Training Using Neural Network

gameStates is an array of x states each with a result and player to move;

pass all states, results, player into neural network;

Neural Network makes predictions;

Neural Network adjusts weights and biases using the adam optimzer to reduce error;

8.6.2 Algorithm for Finding Value of a State Using the Neural Network

Finding the value of a state is fairly similar to the previous implementations. However, instead of querying a map to find the value, we use the prediction function of the neural

network to return our value: Here, a universal algorithm for when we play as crosses as well as when we play as noughts is introduced. For crosses, maximise value. For noughts, minimise value.

Algorithm 9: Finding value of state using Neural Network

```
state = state we wish to find the value of;
value=NN.predict(state);
```

Chapter Nine

Implementing Training Data Experiments

Here, the algorithms that are needed to implement the required agents are shown. For each implementation, the focus will be on simply how each training methodology is carried out and any decisions that occurred during the actual implementation. As has been discussed in 6.0, the value function implementation will be constant between all these implementations (a neural network):

9.1 Implementing Random Policy Training

Incidentally, due to the structure of this project, this implementation is identical to the neural network implementation in the value function section of the project. Thus, an algorithm has already been given for how a random policy training methodology takes place. This is shown as 'Algorithm 1' in this report.

9.2 Implementing Epsilon Greedy Training

Although this may have seemed simple to do in my System Design, it actually proved to be a major challenge, mainly due to issues with the TensorFlow predict and fit methods. For

example, my initial implementation of the algorithm which went like this:

Algorithm 10: Training using Epsilon Greedy Policy

```
for i=0 to 60,000 do
    gameStates = [];
    state = empty state';
    Game not over possStates = array of all the possible states from state;
    values=[];
    for each state in possStates do
        values.append(NN.predict(state));
    end
    if rand(0,1)>epsilon then
        state= new state probabilistically picked from possStates based on values
         array;
    else
        state= randomly picked from possStates;
    end
    gameStates.append(state)  result= result of the game ;
    pass gameStates into neural network as training data;
end
```

This may seem innocuous, but when you consider that each time we run the prediction on the model for the x possible moves, the single prediction takes 0.17s. Therefore, considering each game requires 9 moves to complete and we are producing 60,000 games, we are running this command 540,000 times. This leads to 26 hours of time required just to produce these 60,000 games. Also, since after each game, we pass the game into the neural network to allow it to learn and influence our future games, the model.fit tensorflow operation must be run - taking around 10 seconds. Thus, in total, to run this for 60,000 games multiple days or weeks would be required.

After studying the concept of batch predictions in TensorFlow, I realised that increasing the number of batches in each prediction only very marginally increases the time for prediction. Therefore, theoretically, if we could play 1000 games in parallel and feed these 1,000 states into the prediction algorithm each time, we could theoretically greatly speed up performance. This could then be repeated around 60 times to produce the benefits of epsilon greedy:

Algorithm 11: Training using Epsilon Greedy

```
for i=0 to 60 do
    gameStates = []*1000;
    while not all games have ended do
        state = [empty state] * 1000';
        possStates = [array of all the possible states from state[i]]*1000;
        totalValues=(NN.predict(possStates));
        for t=0 to 1000 do
            if rand(0,1)>epsilon then
                state[i]=possStates[i][probabilistically picked based on values in
                  totalValues[i]]
            else
                state= randomly picked from possStates[i];
            end
        end
        gameStates.append(state);
    end
    pass gameStates into neural network as training data;
end
```

Now, considering that each game takes 10 moves to complete and the predictions take roughly the same amount of time as previous, we should be able to make all these predictions

within a much shorter time span. This is a monumental improvement in the performance of the algorithm. This simple, although extremely logistically complex change now made a once unviable methodology viable.

After finally finding an algorithm for implementing the neural network led training data generation, the finalised data can finally be input into the actual neural network and be used to train it. Unlike in the previous random methodology, this training data is produced in batches and used to iteratively train the neural network. This allows each iteration to produce 'better' and more optimal training data than the last.

9.3 Implementing MCTS Neural Network Training

This algorithm is by far the most complex one that is implemented over the course for my project. The implementation started simply by implementing the concept of MCTS training data generation in the way that is shown here i.e. that is following the selection, expansion, simulation and backpropagation steps using the number of wins and losses as the indicator to a good value state. All the training data produced by completing the games in this way would then be passed into the neural network. This basic algorithm is shown here:

9.3.1 Algorithm for Generating Data using MCTS

After explaining each individual step that is involved in a Monte Carlo Tree Search in the System Design, only a high level algorithm for my implementation is provided. Each step in

a MCTS is referenced in the algorithm without explaining too much in detail:

Algorithm 12: Generating next state using MCTS

```
Create tree with root node R;
// selection steps
while R has children do
    R = a child of R -picked based on number of wins and number of visits;
end
// expansion steps
Create child(s) from R;
// simulation steps
Simulate a game using random moves from L;
// backpropagation steps
Record results and backpropagate;
Store all the states that occurred in this play-out;
Repeat for 60,000 games;
```

This algorithm offers advantages over the ϵ-greedy method due to the fact that the exploitation is taken from the tree rather than the neural network. Thus, proving to be a lot faster than the other algorithm. These 60,000 games are then passed as the training data into the NN of the agent.

9.4 Implementing Supervised Methodology

This training data methodology uses a minimax algorithm to implement exploitation. This algorithm has been explained in the system design section. The way this has been imple-

mented is as follows:

Algorithm 13: Training using Supervised Learning

Take all possible states that are one move away from the current state;

if *first or second move* **then**
| Pick a possible state at random and make this the current state

else
| Find the minimax value of each possible state

end

if *the value of the highest value state is far greater than any other* **then**
| Pick this state and make this the current state

end

if *the value of the highest state is similar to other states* **then**
| Pick any of these states at random and transition to this state

end

This is repeated for every move in 60,000 games;

Part V

Evaluation

Chapter Ten

Results

As has been mentioned in 4.1, a round robin has been conducted of all the trained agents that were produced in this project. These have been split into two separate tables for clarity. The first representing the round robin of the value function experiments and the second representing the round robin of the training data experiments.

Each value in the following tables represents the win rate (discarding draws) when the row agent plays 10,000 games against the column agent:

10.1 Value Function Implementation Results

Read the Row, then Column

	Random	No Approximation	Simple	Complex	Rand NN
Random		45.3%	33.8%	34.2%	16.4%
No Approximation	54.7%		34.6%	30%	25.2%
Simple	66.2%	63.4%		44.9%	26.25%
Complex	65.8%	70%	55.1%		27.5%
Rand NN	83.6%	74.8%	73.75%	72.5%	

Figure 10.1 Value Function Results

10.2 Training Data Implementation Results

Read the row, then Column

	Epsilon Greedy	MCTS NN	Rand NN	Supervised NN
Epsilon Greedy		48.5%	48.7%	41.8%
MCTS NN	51.5%		48.8%	42.2%
Rand NN	51.3%	51.2%		42.8%
Supervised NN	58.2%	57.8%	57.2%	

Figure 10.2 Training Data Results

Chapter Eleven

Discussion of Results

11.1 Discussion

From these experiments, a great deal of conclusions can be drawn from the data gathered. For example, starting with the value function experiments, we can see that the simple trend of increasing approximation yielding increasing performance is seen. In the first table, the agents are ordered by increasing approximation, and a clear upwards trend can be seen with the win rate of these agents.

For the value function with no approximation, the results show fairly poor results with only a 54.7% win rate against even the random agent. The results against the other agents were also fairly poor. This, however, was an expected result and is likely due to the gaps that would arise in the agent's knowledge when only 60,000 games are considered and no approximation is used.

We then moved onto the simple heuristics. A marked improvement in performance from the value function with no approximation was seen. From 50% total win rate to 67% against the random agent. To me, it's quite surprising that such a simple heuristic would perform so much better than the fairly sophisticated lookup vf. This reveals the benefit of even basic approximation in these contexts.

The complex heuristics, however, did not offer much change over the simple heuristics and

only a slight increase in performance was seen between the simple and complex heuristics. I had, perhaps, expected a slightly greater increase in performance. However, this may highlight diminishing returns in using heuristics for such approximation.

The neural network implementation highlights a major improvement over all the previous implementations of the value function. Despite using the same training data and testing process as previous implementations, the proportion of wins were astoundingly higher than the previous versions of the agent. Over a 70% win rate against every other agent was seen. These results undoubtedly show this implementation to be the best performing value function.

Moving on to the training data experiments, we can see that our expectations did not seem to be played out in our results. The first implementation, epsilon greedy, did especially underform - even compared to the randomly generated training data nn. This is really surprising to me as from research papers, I would have expected this one, due to its increase in exploitation, to outperform the method with the randomly generated moves. However, perhaps due to the fact that we aren't using enough data, I can see that this increase in exploitation seems to have only harmed this implementation. I attempted this methodology with various epsilon values (in the context of epsilon greedy), yet could still not see an improvement in performance.

The MCTS trained agents results show that while an improvement was seen between this agent and the previous NN-Led agent, the new MCTS agent actually seemed to perform to a lower standard than both the Supervised NN as well as the Randomly trained NN. The loss to the Supervised NN was actually somewhat expected. However, the fact that MCTS agent lost to the Randomly trained agent (albeit to a small margin) solidified the fact that the implementations which exhibited some form of exploitation had not performed to their expected standard.

As expected, however, the supervised agent vastly outperforms each of the other implementations. This highlights the fact that improving our training methodology can, in

fact, have a substantial implication in the final performance of the agent. I also tested this algorithm by playing it myself and could instantly notice a large increase in the difficulty of achieving a win against the agent.

11.2 Critical Analysis

From the value function experiments, we can quite confidently conclude that simply, as we produce value functions that are more generalizable, they are able to perform better at our 4x4 Tic Tac Toe game. For example, as we moved from our first implementation which exhibited no generalizability into our heuristic implementations, we saw major improvements. Then, as we moved into using neural networks, results were further improved.

However, I can say that in general, my comparisons between the different training data methodologies that I produced did not support the results that other published research papers had offered. I had previously expected the RL agents which exhibited some kind of exploitation to outperform the other agents. This was not seen in my results.

There may have been many reasons why my implementation of these agents did not outperform the randomly trained agent. For example, it is possible that the limited number of games I used for each of these experiments very much limited the performance of many of these algorithms. This is especially true for the Monte Carlo tree search implementation, which in Google's alpha go, used millions of games to train their NN **[11]** this is compared to only 60,000 for my implementation.

The reasons I set my number of games to this fairly low number was for 2 main reasons. Firstly, due to time and technical constraints. If, using my current computation power, I had aimed to reach the number of training games that alpha go used, it would take many months of training. The other reason was to keep the experiments fair with the same number of training games. Perhaps, if I had done this project again and had more resources, it may have been useful to have trained each for 200-300,000 games and recorded the difference

in performance. In general, my results do not, in fact, show conclusive evidence for improvements based on differing training samples. This may be offer an opportunity for future work.

Chapter Twelve

Conclusion

In general, I was pleased with the way my project turned out as I have been able to gain an understanding of a vast number of different concepts that relate to Reinforcement Learning. Also, I believe that actually implementing these concepts have given a new dimension to many of these concepts. Although some of my results were not as expected, they do give me some purpose to investigate these findings further with future work.

12.1 Achievements

By the end of my project, I had met all the aims and objectives that had been set out at the start of the project. I have been able to gain an understanding of the concepts involved in Reinforcement Learning far beyond the scope of what's taught in course units. The implementations of Reinforcement Learning have also been a core achievement in the project. Each of these implementations have been produced using only knowledge of the concepts involved with no modules or prior code being utilised.

The experiments I ran throughout the course of the project have been also cogently and fairly conducted and have allowed for critical analysis and conclusions to be drawn. Many of these results corroborated results that had been commonly accepted. Also, the final agents produced good performance even with a limited number of considered games.

12.2 Personal Development

Prior to starting this project, I had only a fairly rudimentary understanding of Reinforcement Learning and the concepts involved. Thus, a great deal of outside research was required in order to implement the different concepts involved in my project. Reinforcement Learning is not a focus of the courses I am undertaking and thus, in order to understand concepts like value function approximation and policy improvement, all this research had to be undertaken independently.

This project has also reinforced the importance of good experimental design and planning prior to implementation of an experiment. This is another key personal development I experience over the course of this project.

12.3 Future Development

This future work could, for example, include testing the implementations with more training data e.g. 2 or 300,000 examples for each implementation. This would help to test my previous hypothesis regarding the second section of the project. Aside from this, I would also like to test other Reinforcement Learning concepts such as Q-Learning and compare these to the results shown here.

Bibliography

[1] Kaelbling, L.P., Littman, M.L. and Moore, A.W., 1996 Reinforcement learning: A survey. Journal of artificial intelligence research, 4.

[2] Medium. 2020. How Does The Bellman Equation Work In Deep RL?. [online] Available at: https://towardsdatascience.com/how-the-bellman-equation-works-in-deep-reinforcement-learning-5301fe41b25a

[3] SpringerLink. 2020. Value Function Approximation. [online] Available at: https://link.springer.com/referenceworkentry/10.1007%2F978-1-4899-7687-1_876

[4] Chessprogramming.org. 2020. Piece-Square Tables - Chessprogramming Wiki. [online] Available at: https://www.chessprogramming.org/Piece-Square_Tables

[5] Web.stanford.edu. 2020. [online] Available at: https://web.stanford.edu/class/cs234/CS234Win2018/slides/cs234_2018_l5_annotated.pdf

[6] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems.

[7] Robbins, H. and Monro, S., 1951. A stochastic approximation method. The annals of mathematical statistics.

[8] Qian, N., 1999. On the momentum term in gradient descent learning algorithms. Neural networks, 12(1).

[9] Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[10] Tokic, M. and Palm, G., 2011, October. Value-difference based exploration: adaptive control between epsilon-greedy and softmax. In Annual Conference on Artificial Intelligence . Springer, Berlin, Heidelberg

[11] Npr.org. 2020. NPR Choice Page. [online] Available at: https://www.npr.org/sections/thetwo-way/2017/10/18/558519095/computer-learns-to-play-go-at-superhuman-levels-without-human-knowledge

[12] LeemonBaird,1995,Residual Algorithms: Reinforcement Learning with Function Approximation https://www.sciencedirect.com/science/article/pii/B978155860377650013X

[13] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. and Hassabis, D., 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419)

[14] Besbes, O., Gur, Y. and Zeevi, A., 2014. Optimal Exploration-Exploitation in a Multi-Armed-Bandit Problem with Non-Stationary Rewards. SSRN Electronic Journal,.

[15] Christopher J. C. H. Watkins Peter Dayan, 1992. Q-learning

[16] Hornik, K., 1991. Approximation capabilities of multilayer feedforward networks. Neural Networks, 4(2)