Reinforcement Learning for Games

By Adam Swayne
16151852

Final Year Project
completed under the supervision of Mr. J.J. Collins
In fulfilment of the requirements for a bachelors degree of
Computer Games Development

May 4, 2020

Contents

List of Figures

Listings

Acknowledgements

This project could not be completed without the leadership and guidance from Mr. J.J. Collins.

I'd like to also thank my family and friends, for their constant support and source of inspiration.

Abstract

Reinforcement learning, a machine learning paradigm, has proved to be quite effective in the area of game-playing. In this project, I am going to briefly introduce the origin of Monte Carlo Tree Search, and talk about my implementation based on work done by previous authors. The algorithm in question, incorporates temporal difference learning and was applied to the games of Tic-Tac-Toe, and Connect4, but I am confident that it can be extended to other game-playing areas due to the generality in which I incorporated. I tested the algorithm vs Minimax, as well as the standard Monte Carlo Tree Search algorithm and found promising results. I also assessed the impact that hyper-paramaters have on the behaviour of the algorithm, with regards to both the learning rate, as well as the exploration rate of the algorithm. Monte Carlo Tree Search, enhanced through the use of temporal difference learning can be shown to exhibit a better performance than the standard Monte Carlo Tree Search algorithm, achieving a win rate of around 70%, and around 80% against Minimax. I also found that the tuning of hyper-paramaters is absolutely paramount in the success of the algorithm, as there can be a wide discrepancy between win rates if the wrong paramaters are chosen. I think further work could be done in order to increase the performance even more as an extension of this project.

Chapter 1

Introduction

1.1 Overview

For as long as computer games have been around, mathematicians and computer scientists alike have been trying to develop ways for an Artificial Intelligence (AI) that is capable of not only playing games adequately, but is also able to reach human-like intelligence and be capable of beating the best human players in the world.

Primordial AI algorithms usually focused on problems that are intellectually difficult for humans but relatively straightforward for computers to solve(Goodfellow et al. 2016), but numerous strides in the field of AI have lead to the development of ***machine learning*** algorithms that don't just brute-force search all possible states of a problem domain, but also gives the AI the capacity to use past experiences to influence future decisions (learning). This brute-force searching of the state space becomes too computationally expensive to do as games get more and more complicated with higher branching factors, as such, algorithms such as Minimax grind to a halt as the state space grows exponentially.

For the purpose of this project I will be mainly focused on reinforcement learning, a machine learning paradigm, with particular focus on using ***Monte Carlo Tree Search*** (Coulom 2006) for the games Connect4 and Tic-Tac-Toe.

1.2 Research Objective

The intent of my research with regards to reinforcement learning and Monte Carlo Tree Search for the game Connect4, and Tic-Tac-Toe is:

Primary Objectives

- Quantify the efficacy of the algorithm, through detailed analysis of the performance of the algorithm.
- Assess some hyper-paramaters of the algorithm, in order to assess the changes they have on the behaviour of the algorithm.
- Assess how the algorithm fares vs benchmark algorithms in the literature, such as Minimax, by monitoring its win rate over time as it trains in games of self-play.

Secondary Objectives

- Gain an understanding of Reinforcement Learning, as well as the use of neural networks.
- Gain an understanding into previous work done on the subject, with regards to AlphaGO, as well as TDGammon.

1.3 Methodology

The following is my approach to this project:

- Define the research topic.
- Undertake a literature review in order to gain a deep understanding of the domain.
- Define the objective of my research.
- Create a prototype which is the culmination of my research.
- Carry out empirical studies on the prototype, in line with my research objectives.
- Do a detailed evaluation of the empirical studies.

1.4 Overview of the Report

This report is split into 5 different chapters. This chapter briefly gives an overview of my project.

In the second chapter, I list all current research done to date, in detail.

In chapter three I then implement what I have learned, and go through the code that I have developed.

I will then be running a few experiments on the algorithm developed as outlined in chhapter four, and will discuss the impact that certain *hyperparamaters* have on the behaviour of the algorithm in question.

The final chapter will then be a discussion on my conclusion from the results observed.

1.5 Motivation

I am hoping to have multiple versions of this project, the first version should incorporate vanilla Monte Carlo Tree Search. The second version is a new algorithm I have developed, called TDMCTS (Temporal Difference Monte Carlo Tree Search) which stems based on previous work done in the literature. The changes made to the regular algorithm is outlined in chapter three.

I think the most challenging aspect of this project will be trying to prove that learning is taking place, which I am hoping to assess by either playing it against previous iterations of itself over time, or by playing it against a different algorithm as a benchmark; ideally Minimax and then graphing its wins over time.

As well as that, another key challenge is finding the right balance of hyper-paramaters. If any such hyper-paramater is wrong, it can throw off the whole learning and tournament process, and so deep experimentation needs to be done in order to find a right balance between them.

Chapter 2

Background

2.1 AI and Machine Learning

Machine Learning, first coined by Samuel Samuel (1959) in 1959, is a form of applied statistics (Goodfellow et al. 2016), and involves statistically estimating complicated functions, as well as learning from previously given data. It's the computer's job to "learn" the appropriate function, so that it can accurately predict the output of the function, given previously unseen inputs. By learning we mean, *"A computer program is said to* ***learn*** *from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"* (Mitchell 1997, p.2). In simpler terms, machine learning is the ability for an agent to learn from data.

P, is usually determined based on the accuracy of the model (i.e. How many images were correctly labeled in a dataset) or in this case, the amount of times the agent wins against a fixed-difficulty AI like Minimax. **E**, in my case the agent playing games of Connect4 and achieving a reward for the outcome of the game would be its experience. **T**, the set of tasks would be trying to win a game of Connect4.

Machine Learning algorithms can be split into three main paradigms: ***reinforcement Learning*** (Trial and error learning from an interactive environment), ***supervised learning*** (Making predictions about the future based on previous data), and ***unsupervised learning*** (Finding and learning hidden patterns in sets of data). There are also other variations, which are a combination of one or more of the previously mentioned paradigms.

The goal of machine learning is to learn an approximate function so that it can classify and map inputs to certain outputs, but also be able to predict accurately for previously unseen inputs (as is the case using a neural network when the state space is simply too large), doing this well is known as **generalization**. (Goodfellow et al. 2016)

2.1.1 Supervised Learning

In supervised learning, the agent is given a list of labeled data, and based on that labeled data, is able to approximate the function that maps the labeled inputs to the labeled outputs, and thus is able to make inferences about future unseen data based on how the previous data was labeled, also known as classification. This method of learning has proven to be particularly useful in the field of computer vision, and when given a large enough dataset to work off can reach accuracies of up to 99%, far outperforming even a human (Mihajlovic 2019). It is called supervised learning because a human (i.e. the "teacher") has the job of telling the agent whether or not it is right or wrong with each output that it produces. This method of learning can be inapplicable when it is difficult to correctly label the data.

In Supervised Learning, the datasets must all be setup and laid out in a certain way, as a result there can be a high cost involved in the capturing of datasets to use for classification. If you were to apply supervised learning to Connect4, you would need to label every single state in Connect4. This would lead to a dataset involving around 1.6×10^{13} (16 trillion) possible states, which is not feasible at all. One popular application of the supervised learning method is learning handwriting, like the MNIST dataset; a dataset that contains 60,000 patterns and 10,000 test set examples for use with recognising handwritten digits. (LeCun et al. n.d.)

2.1.2 Unsupervised Learning

As opposed to supervised learning, the agent is tasked with finding the *"best representation of the data ... that preserves as much information about x [an input] as possible"* (Goodfellow et al. 2016, p.3). Unsupervised learning tends to lead to classification of clusters of data, with future inputs being labeled as members of one cluster or another. Unsupervised learning provides a nice advantage over supervised learning where the outputs are hard

to be correctly identified by a human. It's particularly good at finding previously unknown patterns in datasets, but can be less accurate than supervised learning as it doesn't involve any human intervention (although this doesnt' always necessarily make it less accurate as it can find previously hidden patterns unknown to humans, making it more accurate than supervised learning in most instances) and has seen application in the use of anomaly detection (Eskin et al. 2002).

2.2 Reinforcement Learning

Reinforcement Learning is the process by which an agent; given an observation of the current state of the game (i.e. the state, a list of actions and a reward for picking each action), tries to maximise its reward over a period of time (in our case throughout the full game of Tic-Tac-Toe or Connect4).

The idea behind Reinforcement Learning is based on an idea from pyschology called classical conditioning (K 2019), most commonly known as Pavlovian conditioning named after the phamous pyschologist Ivan Pavlov. The idea behind his experiment was simple. Everytime Pavlov would ring a bell, he would serve food to his dog. Overtime, just the act of ringing the bell would cause the dog to salivate, as it learned to associate the bell with the notion that it was getting fed. In my case with Connect4, the agent learns which states are good and which states are bad based on the reward (the meat in the Pavlov experiment). As a result we would hope that the agent would strive towards maximising its reward (i.e. getting the food, or in my case, winning the game).

Reinforcement learning can be split into multiple separate paradigms, three of the most common methods used are called: ***dynamic programming, Monte Carlo methods*** and ***temporal difference learning***. In Reinforcement Learning, the agent is not told anything about the rules of the game, nor is it told which actions it should take, it is supposed to experiment with actions, observe the reward, and then remember which actions led to states with higher rewards. (i.e. Reinforcement learning is about evaluating actions as opposed to instructing the agent the correct action to take (Sutton & Barto 2018))

2.2.1 Elements of Reinforcement Learning

Reinforcement Learning is said to be composed of four main components: ***Policy, Reward, Value Function*** and ***Model*** (Sutton & Barto 2018).

·**Policy -** The policy is a state-action mapping which determines the behaviour of the AI given the current state of the game, and list of actions that can be applied. The state, in relation to Connect4, could be a two dimensional array representing which slots were taken up by which player, while the actions would be a list of columns on the Connect4 board that are not yet full. The policy *should* tell us the best possible move to make at a certain stage in the game.

·**Reward -** At all stages during the game, the AI is returned a single number, the reward. The main aims of our AI is to maximise this reward (i.e. win the game). The reward for a state could be +1 for a win, -1 for a loss

·**Value Function -** The value function **V(S)** can be seen as a function that represents the expected average reward for a given state, it intends to maximise the reward over the long run. We already know the values of terminal states (i.e. when there is a 4 in a row for us, the reward is 1, when there is a 4 in a row for the other player, the score is -1.) The idea of the value function is to try and figure out the value of the states leading up to this terminal state (i.e. How much is a 3 in a row worth if it is our turn, as opposed to the same state but its the other player's turn?), estimating this value function can be seen as the core idea of Reinforcement Learning.

·**Model -** When the agent is given a state and an action, having an accurate model of the world might lead the agent to be able to make inferences about the world it's interacting with. The model might predict the probability of ending up in the next state given the current state and action. The model isn't necessary, as is seen in Monte Carlo methods and in Temporal Difference learning (See section 2.2.5).

2.2.2 Exploration & Exploitation Tradeoff

In Reinforcement Learning, there is a tradeoff between ***exploration & exploitation***. At every time step **t**, the agent has a choice of which actions

to take. It would be unwise to *always* choose the action that the policy has regarded as the best, because the policy is based on previous experiences, this can lead to the agent always choosing actions with which it has seen success in before. Although this sounds OK, it is a sub-optimal approach as past experiences are only going to be a certain sub-set of the total search space of the game. This use of not always exploiting should only be used if the entire state space hasn't been sufficiently explored. As a result, it is more advantagous to occasionaly branch out from what the agent already knows (i.e explore), rather than stick to what it knows was succesful in the past (i.e exploit). One thing to note about exploration & exploitation is that as the agent gets closer to the end of its training, it should tend to stick to what it already knows, and thus it should explore more early on in the learning process when it doesn't know much about the game, and as it improves it should stick to what it already knows and refine its strategy. This balancing of exploration and exploitation is handled through the use of the UCT algorithm in the selection phase of MCTS (See section 2.5.2)

2.2.3 Markov Decision Process

In order for our agent to be able to process the game, we have to model the game environment in a certain, consistent way so that the agent receives a representation **(observation)** of the state, which is dependent on the action just chosen by the agent, as well as the reward for that state at specific time steps $t = t_0, t_1, t_2, t_3 \ldots t_n$ (Note: time step here doesn't necessarily have to mean a fixed point in time, it can also mean how many actions it took to get to the current state). We can model our environment based on the Markov Decision Process because in our game, any given state and action can predict the next state, without having prior knowledge of past states that we have iterated through (i.e. *"Future is independent of past, given present")* (StanfordOnline 2019).

Named after the Russian mathematician, Andrei Markov, the Markov Decision Process (MDP) is a way in which we can model decision making by a rational agent where outcomes are based largely on stochastic processes. The idea of a MDP is to find an optimal policy function $\pi_{\mathbf{t}}(a_{\mathbf{t}} \mid s_{\mathbf{t}})$ that the agent will use in order to select the most optimal action given the current state of the game; it determines how the agent will behave at a given time (Sutton & Barto 2018, ch.3).

An MDP consists of:

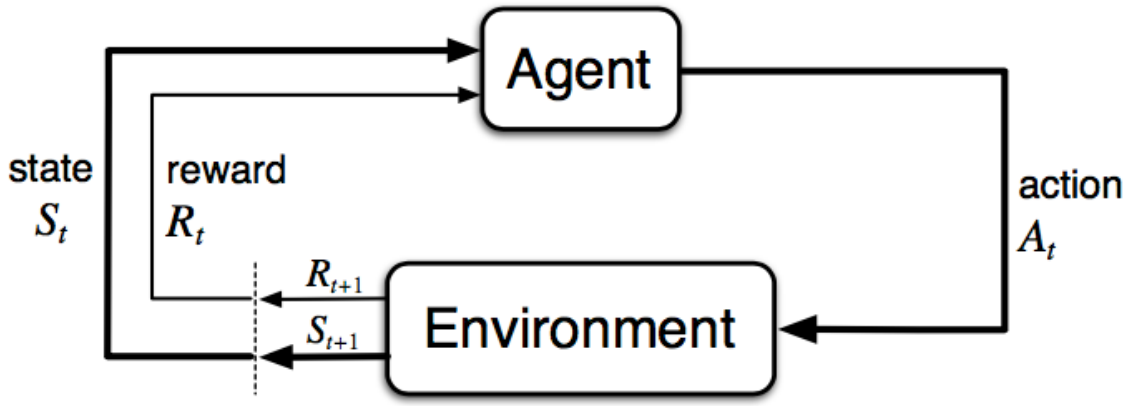

Figure 2.1: A typical model of an MDP, (Sutton & Barto 2018)

· The state space of the game **(S)**, a set which consists of board configurations of the game and which includes the current player turn for each state.

· A set of actions $(\mathbf{A_t})$, which can be applied to the state $\mathbf{s_t}$ (Note: $s_\mathbf{t} \in S$).

· A probability function $\mathbf{P}(s' \mid s_\mathbf{t}, a_\mathbf{t})$, where $a_\mathbf{t} \in A_\mathbf{t}$, that determines the probability that the state $\mathbf{s_t}$ will transition into the state **s'** given the selected action $\mathbf{a_t}$. (Note: $\mathbf{s' = s_{t+1}}$)

· A reward function (also known as a value function) $\mathbf{R_t}(s' \mid s_\mathbf{t}, a_\mathbf{t})$ which is the reward that we would expect to be received by the player for the state *s'* using the selected action a_t on the state s_t at time-step t using the policy $\pi_\mathbf{t}$ (Note: s_t is the state at time-step t, while s' is the state at time-step t + 1).

· A *discount factor*, γ which adds a decreasing weight value to subsequent rewards, forcing the agent to pursue actions early, which should make the agent tend towards finding the shortest path to the goal (terminal) state, furthermore it also stops the agent from getting stuck in an infinite loop as is the case of MDP's with infinite size, or an MDP that has a loop. By setting a discount factor that reduces itself to 0 gradually, we ensure that the reward will converge to a value, and not get "stuck". Although in our case, this isn't a problem as the state-

space of Connect4 is a tree-like structure, and therefore there are no loops when going through gamestates.

(Vodopivec et al. 2017)

Solving the problem of finding an optimal policy function requires finding a policy function that *"maximises the cumulative discounted reward, which is also known as the return"* (Vodopivec et al. 2017). This policy function could simply be a lookup table listing the mapping between states and actions, as is seen in **Q-learning**, or in our case, the use of a search process i.e. **Monte Carlo Tree Search** (See section 2.5).

MDP problems are usually trained in *batches*, in which there are sub-batches called *episodes*. You can think of an episode being a full game of Tic-Tac-Toe or connect4. How the results from each batch and episode are interpreted are based on which reinforcement learning paradigm you use to analyse the data, as well as what collection of algorithms you intend to use.

2.2.4 Dynamic Programming

Dynamic Programming (DP) is a *"collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP" (Sutton & Barto 2018, ch.4)*. A problem can be solved by DP if it satisfies the following two properties: (Cormen et al. 2016)

1. **Optimal Substructure** - the problem can be composed (and therefore solved through the combined use) of multiple usually simpler subproblems. The problem itself can usually be defined recursively.

2. **Overlapping subproblems** - The aforementioned subproblems arise in multiple instances. Like in the fibonacci sequence, f(4) can be broken down into f(3) * f(2) * f(1) .., and so that value of f(4) can be reused when trying to calculate f(5) (Note: f(5) = f(4) * f(3) ...) instead of having to recalculate all the values before it again. A problem which has an optimal substructure but no overlapping subproblems are said to be called *Divide and Conquer* algorithms (Cormen et al. 2016).

Relating the two aforementioned points to connect four is quite simple, imagine as the game is being played there comes a point where there are two different moves which have the same value, and going down either of these

paths will eventually lead to a win. This shows that the game can be broken down into multiple sub-paths or sub-problems.

The Bellman equation, named after its creator, Richard Bellman who introduced DP in the 1950s, is an equation used for finding the optimal policy for a certain problem through the use of the value function. It is a recursive function which operates based on a known model of the world, and can be expressed as:

V(s) = $\Sigma_a[\pi(a \mid s)](R(s') + \gamma V(s'))$ In layman's terms, V(s) is the sum of the values of the next states s'. π is the policy function, which outputs the best (max) action to take in the state s, outputting the next state, s'. R(s') is the reward received for the next state. G(s') is the expected gain we would receive from this next state and γ is the hyper-paramater used to discount the value of future rewards.

Dynamic Programming algorithms don't wait until the end of an episode to learn. Results are backpropagated at each time step so that an estimate of each state can be made before the episode has finished, therefore estimates are calculated based on previous estimates, also known as bootstrapping.

With Dynamic Programming, one requires a model of the environment. What this means essentially is that the exact probabilities of being in a certain state is known beforehand.

2.2.5 Monte Carlo Methods

Unlike Dynamic Programming algorithms, Monte Carlo methods do not require a model of the world and they make estimates based on repeated random sampling of the world, and then summing all these sample returns. The idea is that if enough (large amount of) random samples are done, then the estimated averages of these samples should converge to their actual values due to the *Law of Large Numbers* (otherwise known as the *"law of averages"*) (Smith & Kane 1994), therefore it's safe to say that the more simulations or samples that we do the more accurate the result is.

Unlike DP, Monte Carlo Methods only update the value of each state at the end of each episode, and thus; the updated value for the value function can be represented through linear interpolation between the reward gain received at the end of the episode, compared with the current value for the state: V(s) = V(s) + α (R(s') - V(s)), where s' is the terminal state, and R(s') is the reward returned from that state..

2.2.6 Temporal Difference Learning

Often seen as a mix between the two aforementioned methods, Temporal Difference learning combines sampling random returns (Monte Carlo) with the bellman optimality equations (Dynamic Programming). At every time step, the estimate of the current state is updated based on the estimate of the next state. This updating of state estimates based on **temp**orary (previous) estimates is where the name is derived from (essentially they are learning a guess from guessing, with each guess hopefully becoming more accurate than the last. This is also known as *bootstrapping*), and saw huge success in Tesauro's TD-Gammon program (Tesauro 1995). According to (Sutton & Barto 2018, ch.6), TD updates are known as *sample updates* because they use the value of successive states to back up their values based on the reward observed.

One thing to note is that, although the current state is updated based on the next successive state; the next state isn't available until it has actually been explored and sampled. Violante (2018)

There are many different Temporal Difference learning methods, in the following sections I will briefly touch on a few of them, but essentially the algorithms just involve some linear interpolation of two values, the successor/next (s') state with the current state (s) and can be summarised by: $Estimate = old_estimate + step_size(Target - old_estimate)$. The target involves some information about the next state, while the step size is just the learning rate (i.e. How much we should step in the direction of this new value, as opposed to our old value).

·**TD(0) -** Also known as *one-step TD* (Sutton & Barto 2018, ch.6), as the name suggests involved the immediate back up of the next state to the state before it. (i.e. : V(s) = V(s) + α(R(s') + γV(s') - V(s))

·**TD(1) -** Similar to Monte Carlo in that it only does its backup at the end of the episode. (V(s) = V(s) + α(G(t)- v(s)), the only difference between this and TD(0) is that the immediate reward is swapped with the gain from the future states. A problem with TD(1) over TD(0) is that if the game or environment does not have any terminal states then TD(1) is not applicable.

·**TD(λ) -** This algorithm was invented by (Sutton & Barto 2018, ch.6), based off of Arthur Samuel's earlier work on a checkers program (See

section 2.4.2). The idea is there is a λ paramater, which again is a hyperparamater between 0 and 1. The number increases based on how **recent** and how **often** a state is visited. Therefore more popular states will have a value closer to 1, while less-often visited states will have a number closer to 0. This λ paramater is multiplied by the TD target to scale the change in value of the state.

There are two views when it comes to Temporal Difference. The *forward* view and *backward* view. The forward view essentially looks forward n amount of steps and uses the reward to update all future estimates, multiplied by the λ paramater to decay the future estimates, while the backward view updates all prior states to the current state at every time step t.

TD(λ) again has a slightly different formula to TD(0) and TD(1), and is expressed by: V(s) = V(s) + α (R(s') + γV(s') - V(s))(E(s)), where E(s) is the eligibility trace of the state in question.

·**Q-Learning -** This algorithm uses Q-Values(also called state-action values) to iteratively improve the performance of the agent. Action-values are whereby the mapping is state-¿action instead of state-¿state, and is usually paired with an ϵ-greedy policy, i.e. a value between 0 and 1 is chosen for ϵ, then at every time step a random number is generated, if it is higher than ϵ a random exploratory move is done.

·**SARSA -** (S)tate, (A)ction, (R)eward, (S)tate, (A)ction involves the mapping of states to successive actions (i.e. State-Action pairs) and is a slight variation of Q-Learning. The difference between the two is that SARSA is an **Off Policy** learning technique, while Q-Learning is an **On Policy** learning technique. According to (Gupta n.d.), On Policy refers to an agent that *"...learns the value function according to the current action derived from the policy currently being used"*, while Off Policy refers to the agent learning *"...the value function according to the action derived from another policy"*. In layman's terms, SARSA learns values based on the policy that it is under, while Q-Learning learns relative to its greedy policy.

2.3 Deep Learning

In recent times, deep (reinforcement) learning has seen a wide array of success in the field of computer game AI (Mnih et al. 2013) (Silver et al. 2016), (Silver et al. 2018), (Silver et al. 2019) with regards to traditional board games such as Chess, GO and Shogi, as well as more visually demanding games such as Atari.

According to (Nicholson 2019), *"Deep reinforcement learning combines artificial neural networks with a reinforcement learning architecture that enables software-define agents to learn the best actions possible in [a] virtual environment in order to attain their goals. That is, it unites function approximation and target optimization, mapping state-action pairs to expected rewards"*. In a sense, it's finding a way to generalise from unseen states in a game, based on learned previous knowledge. This ability to infer from unseen states is what grants it superhuman-level in the aforementioned games. This is done through the concept of **function approximation**. As has been briefly stated before, it's unfeasible to directly track and update a lookup table which contains all the data regarding the different states of the game, with their corresponding learned values. As such, the use of a function approximator is to approximate these unseen states, based on data regarding ones it already knows. (Let's say the AI sees that there are 1,000 known states which offer it a reward of 1, that is, leads to a win i.e. a four-in-a-row in Connect4 or a three-in-a-row in Tic-Tac-Toe. Would you say it is correct to then infer from that those seen states that all states with four-in-a-row/three-in-a-row for their respective games will net a reward of 1?).

Neural networks, based on inspiration from the human brain has multiple layers, each of which hold structures of differing length known as neurons. The first layer would have enough neurons to count for the input to the lookup table (For example, it could take in the current state of the game as a one-hot vector, as well as the current player's turn), while the output would be the total length of the actions in the game (so in a six*seven game of Connect4 it would be seven, while in a three*three Tic-Tac-Toe it would be nine). Each neuron from each layer is connected to the neuron in the layer preceding it, as well as the neurons in the layer succeeding it. This connection between the neurons is known as a weighting factor. The idea is that once the sum of the values of the neurons multiplied by their weight in the proceeding layer pass a certain threshold, the neuron in the current layer would be told to fire, this leads on to the next layer etc. until it reaches the

end of the neural network. At the end, the network will say the probabilities that it's sure of with regards to which action to take next. Of course, it would be extremely inaccurate at first, and so an error is backpropagated through the neural network, which reassesses each individual weight to each neuron, in this way, the neural network is able to fine-tune itself based on the input to the neural network.

A deep neural network, is as the name suggests, similar to an artificial neural network, but with many many layers between the input and the output layers. The idea of adding multiple layers is, in theory, to add more layers of abstraction. If you were to create a deep neural network which could recognise hand-written digits, you'd think you might have a layer which would recognise, for example, the different segments of the number, a layer which would recognise whether the number has a loop like in an 8 or a 9, or a straight line like in a 1. You'd also think to maybe add a layer to recognise where these are in the picture (like a loop being above a slightly straight line is likely to be a 9, while two circles would be an 8, and one circle would be a 1). This conceptually, makes more sense than having one shallow layer which does all the work. This way of learning, where the AI figures out complex problems, building its way up from layers upon layers of quite primitive concepts below it is known as ***Deep Learning*** which has applications in many fields such as natural language processing, image processing (computer vision) and also use in social network filtering.

2.4 Overview of AI in games

2.4.1 Introduction

There have been numerous advances made with regards to AI in games, it can be said that the gaming industry drives the IT industry, and with that it also drives the field of Artificial Intelligence. Today, we are quite far away from having iRobot-like machines due to a number of technical difficulties. For starters, we don't quite have the hardware perfected, yet. As a result, the software side (i.e the AI) is usually abstracted from the real world; free from the constraints of modern-day robotics systems and instead implemented in an artificial game world.

In these kinds of worlds we don't need to worry about the sensors or other complicated parts of the machine, instead we can separate the hardware from

the software, and deal with both of them in parallel. In this way, human-like general intelligence research doesn't have to be delayed until we've figured out how to get a robot to walk and be able to hold things effectively.

2.4.2 Brief History

·**1955 -** Arthur Samuel's checkers program (which incorporates machine learning) becomes the first program to play a game better than its creator.

·**1979 -** Hans Berliner's backgammon program BKG becomes first computer program to defeat a world champion in any game.

·**1992 -** Gerry Tesauro's TD-Gammon, based on Reinforcement Learning techniques (namely Temporal Difference Learning), reaches championship-level ability. This was an improvement on his previous program, Neurogammon, which used supervised learning techniques to achieve a high level of play. (See section 2.6.1)

·**1994 -** The checker's program CHINOOK becomes the first program to beat a world champion in an official game of skill.

·**1997 -** DeepBlue beats the world chess champion at the time, Gary Kasparov.

·**2016 -** AlphaGO, developed by DeepMind, becomes the first computer program to beat a world champion in the game of GO (4-1 Lee Sedol) (See section 2.6.2)

(Bostrom 2014, ch.1)

The interesting thing to note is that as the years go on, more and more complex games are being solved computationaly. This is due to not only the increase in computing power, but because of the development of more complex (smarter) algorithms that do more than just brute-force search a state space examing all possible states.

2.5 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that was introduced in (Coulom 2006). It is an informed search algorithm which focuses on the analysis of the most promising moves and involves the use of

random sampling. It was used by DeepMind when they created AlphaGO, an Artificial Intelligence that achieved superhuman playing abilities for the ancient chinese board game, Go. Go is quite a complex game, much more complex than Connect4 by a large margin, and was believed to not be solvable by computers due to the large search space, before AlphaGO defeated the 18-time world champion Lee Sedol in a best-of-five and won (4-1) (Silver et al. 2016) (see section 2.6. There is around 10×10^{170} possible states for the game, with a branching factor of about 250 (Levinovitz 2014)

2.5.1 Elements of MCTS

MCTS is an any-time algorithm (Sista 2016), which means at any given moment you can stop the search, and it will return the best move to make, which is great for board games where there is a time limit for the agent's turn. It consists of four steps: ***Selection, Expansion, Simulation*** and ***Backpropagation***

· **Selection -** Starting at a **root node (R)**, and then selecting a child from R, based on a **selection policy**. This selection policy helps to manage the problem of Exploration & Exploitation (See 2.2.2). If the child node **(C)** has been selected before, then run a *simulation*, likewise if the node has not been visited before, run an *expansion*.

· **Expansion -** Expand all child nodes from **(C)** , then run a *simulation* on each. Each child node would be the next state of the game if a chosen action was taken from the list of actions from the current state.

· **Simulation -** From **C**, play a full playout (rollout) until a **terminal state** or a **Simulation Depth (S)** is reached. Rollouts can consist of either heavy rollouts, or light rollouts. Heavy rollouts would require us using some sort of heuristic evaluation function, while a light rollout would result in the game being played out with random moves from each player.

· **Backpropagation -** Using the **Reward** function, calculate the reward for the full rollout, and then add this reward to every node from **C** back up to the node **R**, while also updating the value of every node along the way. The returned result is simply the reward associated with the given state (i.e. +1 for a win, -1 for a loss)

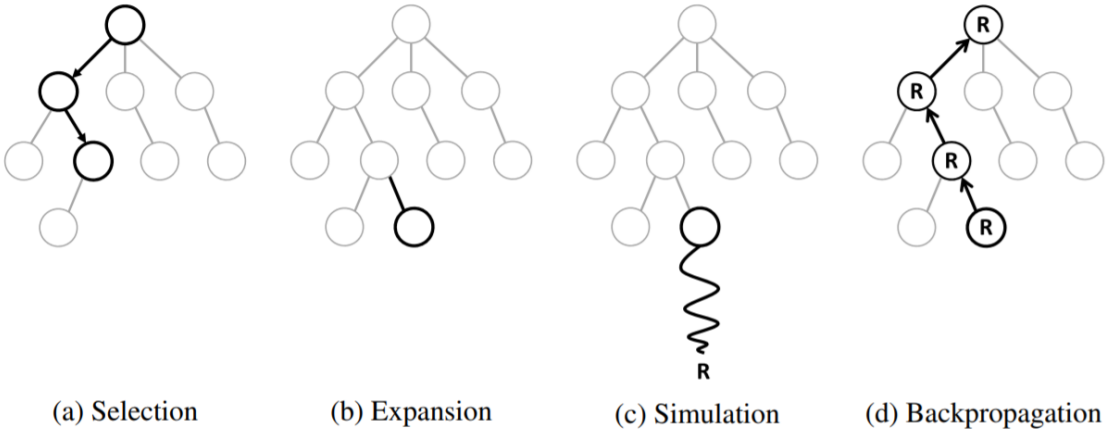

Figure 2.2: Simplified MCTS tree, (James et al. 2017)

2.5.2 Selection Policy

In the selection phase; we have to find a way to determine the successive children for a given state, this is also known as the Multi-Armed Bandit problem and can be thought of as trying to figure out which lever to pull (action) in a slot machine to maximimse your overall chance of winning. There are a few different types of algorithms that can handle this problem, namely; Greedy, ϵ-greedy and Upper Confidence Bound (UCB) which are mentioned by (Sutton & Barto 2018). The algorithm for use in my MCTS prototype as described by (Kocsis & Szepesvári 2006) is UCB Applied to Trees (A.K.A UCT). It is defined as follows:

$\frac{w_i}{n_i} + \epsilon\sqrt{\frac{lnN_i}{n_i}}$, where:
w_i is the number of wins of the node.
n_i is the number of times the node was visited.
ϵ is the exploration factor.
N_i is the number of times the *parent* node was visited.
This Selection Policy can also be known as a Tree Policy.

2.5.3 Child Policy

Although very similar to the selection policy, the difference between the child policy and the selection policy is that the child policy is selected at the end

of all simulated rollouts. That means when the agent has finished searching the space and wants to return a value it is critical that exploration is NOT done, otherwise this could lead to an agent potentially not choosing a 4 in a row (with regards to connect4) because it chose to "explore" other options instead.According to (Browne et al. 2012) in traditional Monte Carlo Tree Search, there are four ways to select the best child from the root:

· **Max Child -** Selecting the child that has the highest score.

· **Robust Child-** Selecting the child that has the highest visit count.

· **Max-Robust Child -** Selecting the child that has both the highest score, and the highest visit count.

· **Secure Child -** Selecting the child which maximises a lower confidence bound.

2.5.4 Rollout Policy

When playing out a simulation, there are different kinds of rollout methods that one can use, namely: light rollouts and heavy rollouts.

· **Light Rollouts -** The state is played out with completely random moves from both opponents until a terminal state is reached.

· **Heavy Rollouts -** The right action to choose is based off of a certain heuristic evaluation function (i.e. If the agent has 3 in a row and it is its turn, this is a very favourable state as it can lead to an instant win).

As seen in AlphaGO Zero (Silver et al. 2018), the use of light and heavy rollouts were completely replaced with a high-quality neural network to predict the evaluation of positions. In my case, I will just be using light rollouts.

2.6 Reinforcement Learning in Games

In this section I am going to explore some Reinforcement Learning examples from the literature. Namely from Gerald Tesauro, and DeepMind's AlphaGO.

2.6.1 TDGammon

TDGammon, developed by (Tesauro 1995) is a temporal-difference Backgammon (See section 2.7 for more information on the game) agent, which was originally developed in order to provide a detailed comparison of temporal-difference reinforcement learning with Tesauro's previous Backgammon agent called Neurogammon. Neurogammon (Tesauro 1989), is a supervised learning Backgammon agent which won the 1st Computer Olympiad in 1989, and was the first backgammon agent to use a neural network, and was estimated to be comparable to a strong intermediate level human player.

TDGammon, as said by Tesauro (1995), was *"designed as a way to explore the capability of multilayer neural networks trained by TD(λ) to learn complex nonlinear functions."* and used temporal difference learning (TD(λ)) through games of self-play to learn.

The first version of TDGammon had no prior knowledge of Backgammon, and was paired with a multi-layer perceptron (neural network) with the input to the neural network being the current board configuration, as well as a vector which flipped depending on whose turn it was to play. This network had 197 neurons in its input layer, and 40 in its hidden layer. Due to the complex nature and high branching factor of Backgammon (see section 2.7), a simple mapping of states to actions as well as crafting good heuristics for board evaluation has proved difficult. The network learned through gradient-descent and weighting errors were backpropagated throughout the network, the weights were initialised randomly at the start, and through the process of self-play was able to achieve the same level of performance as the best backgammon AI's at the time (which was coincidently his previous NeuroGammon) after around 300,000 games of playing against itself.

After incorporating hand-crafted heuristics to guide the network initially, Tesauro's next iteration of TDGammon contained 80 units, as opposed to the previous 40, and after 300,000 games was able to compete with some Backgammon experts.

Later versions of TDGammon incorporated the use of predicting the opponent's move straight after their own move, to anticipate what move the opponent would make(assuming the opponent would choose the best move for themselves). This augmentation enhanced it so that it was able to play against Backgammon grandmasters.

Again, TDGammon was updated, into version 3.0,(containing 160 hidden neurons) which after 1.5million episodes was near the playing strength of the

best human players in the world, and was regarded at the time as the world champion. An interesting take from Tesauro's TDGammon is that, it learned moves which were deemed unconventional at the time. It developed its own opening strategy, which was unheard of at the time, and ever since then, pro players have adopted its openings into their own strategy.

2.6.2 AlphaGo

Monte Carlo Tree Search has seen great success in a variety of games, namely in GO, Chess and Shogi (Silver et al. 2016) (Silver et al. 2018). Chess and GO are estimated to have state space complexities of around 10×10^{50} and 10×10^{170}, respectively). Unlike Chess, the interesting thing to note about GO is that there doesn't seem to be any good evaluation function to determine how good a state is to be in. As such, MCTS can be applied as its use of random rollouts mean that you don't need an evaluation function as the value of a state is determined based on how many wins that has been gotten from that state. This idea of a tree search, paired with a (deep) neural network for generalisation is the reason for MCTS great success in games.

AlphaGO, developed by DeepMind in 2016 beat the reigning GO champion Lee sedol (4-1), something thought to be not computationally feasible for at least the next 10 years. Its success was made by the MCTS algorithm, as well as a policy network (which was first trained on handcrafted "good" moves, and then later trained by evaluating moves) and a value network (which was in charge of evaluating board positions, i.e. it approximated the value function (see section 2.2) for the state

the MCTS algorithm is the main algorithm behind the widely renowned AlphaGO, which trained on a custom system using TensorFlow; with sixty-four GPUs, nineteen CPUs and four TPUs. A single system has been quoted to be around $25 million (Silver et al. 2016) (Calhoun 2017).

There were a few iterations of AlphaGO (AlphaGO Fan, AlphaGO Lee, AlphaGO Master, AlphaGO Zero, Alpha Zero and now MuZero). AlphaGO Zero was able to beat the original AlphaGO (100-0) in the game of GO after just three days of training. The algorithm then beat AlphaGO master after twenty-one days. The striking thing about AlphaGO Zero is that unlike AlphaGO, it didn't rely on handcrafted human moves during the training of the policy network. It learned entirely through self-play, this meant that it could be extended to other games such as Chess and Shogi (and maybe even other domains if the computation power is there).

Deepmind generalized the AlphaGo Zero algorithm into the next iteration, Alpha Zero, achieving superhuman level of play after twenty-four hours, and defeating programs such as Stockfish and Elmo, and even beat the AlphaGo Zero algorithm which had done three-times the amount of training that it had done.

The most recent version of the AlphaGo algorithm is MuZero (Silver et al. 2019), which has extended the playing field from traditional board games of Chess, Shogi and Go, to Atari games. The algorithm achieved record scores in the 57 Atari games that it was tested on, beating all previously made algorithms, and showed that Monte Carlo Tree Search is not only good at solving logical board games, but also games which require a visual aspect to them.

2.7 Backgammon

Backgammon, one of the oldest games in the world, is a two player turn-based game where each player has fifteen pieces called checkers. Each of these checkers can be placed between twenty-four triangles depending on the outcome of the roll of two dice. To win, you must get all your checkers off the board. After every roll of the dice, the player is given the option to move their checkers. The winner is often picked after reaching a pre-defined number of points. The twenty-four triangles are split between two sides of the board, and are numbered from 1 to 24. To determine who plays first, a die is rolled, and the player with the highest number goes first. When the game commences, each die roll corresponds to the number of moves that must be made on a single checker (i.e, if you roll a 2 and a 1, then one checker is moved 2 spaces, while a second is moved only 1 space). A checker cannot land on a triangle occupied by two or more of the opponent's pieces. An opponent's checker can be sent all the way to the far end of the board if a player land's on the checker. This checker must then re-enter the board before the opponent's other pieces can be moved. Strategical blocking can be implemented in order to block the opponent from moving certain checkers forward.

Once all the checkers have been placed on the board, further dice rolls can be used to take checkers off of specific points (i.e, if I roll a 6, then I can take a piece off of point 6). If a player wins, whereby the other player has not yet taken off any of his/her own pieces, then the win is called a *gammon*.

If the same happens but the other player also has checkers on the winner's side of the board (called the home board), the winner scores a *backgammon*.

It has been estimated that there are about 1×10^{19} (10 quintillion) legal board positions for the game. (Wong 1998)

I personally did not implement Backgammon, I focused moreso on Connect4, with preliminary testing done in Tic-Tac-Toe, in the following sections I will discuss these games in more detail.

2.8 Connect4

Connect4 is a two-player turn based game where each player, after being assigned a colour, must take turns dropping their coloured disc into the slots of the game. There is usually seven slots, each being six high (i.e. the game is played on a 6x7 grid). When a slot is picked, that disc then falls to the furthest available slot, which is either at the very bottom of the slot; or just above another disc previously placed in the slot. The objective of the game is to get four coloured discs in a row, while not allowing the other player to get four in a row. Four in a row can be achieved either horizontally, vertically or diagonally. Possible variations to the game could be a variable amount of rows or columns, as well as a different amount of discs needed to win the game.

In Connect4, the search-space size for a regular six row by seven column grid where each grid square can be either "GREEN", "RED" or "BLANK" (for two players) results in 3^{6*7} or 1.094×10^{20} (109 quintillion) possible states, although realisticaly, many of these states are illegal. If there is an empty grid square, then all squares above it must also be empty. Removing all these illegal states, as well as removing states based on what player goes first (i.e. if player one goes first, then all states where player two starts first can be removed) further reduces the search space to be around 1.6×10^{13} (16 trillion) possible states (Lijing & Gymrek 2010)

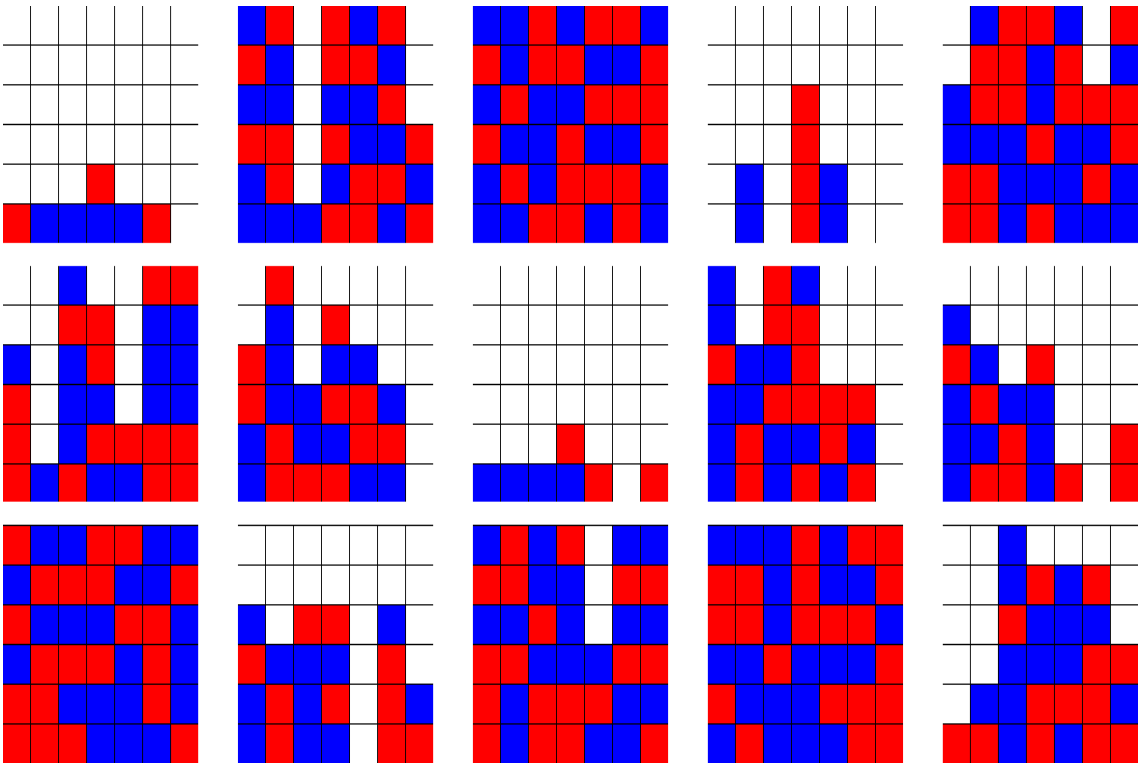

Figure 2.3: Some example games of Connect4

2.9 Tic-Tac-Toe

Tic-Tac-Toe, also known as Xs and Os, or Noughts and Crosses is a two-player turn based game where each player takes turns drawing an X or an O on a 3x3 grid. The objective of the game is to get a 3 in a row of your respective symbol. Unlike Connect4, Tic-Tac-Toe does not require the column to be full up to the point where you want to put an X or an O, every square on the board is available from the start. This means that each board position has three possible states (X, O or EMPTY) for each of nine possible cells leads to a search-space size of 3^9 or 1.9683×10^4 (19 thousand) possible board configurations for the game. This search-space can be further reduced by removing mirror states and invalid states, as such, brute-force algorithms can solve a game of Tic-Tac-Toe quite easily and in a feasible amount of time.

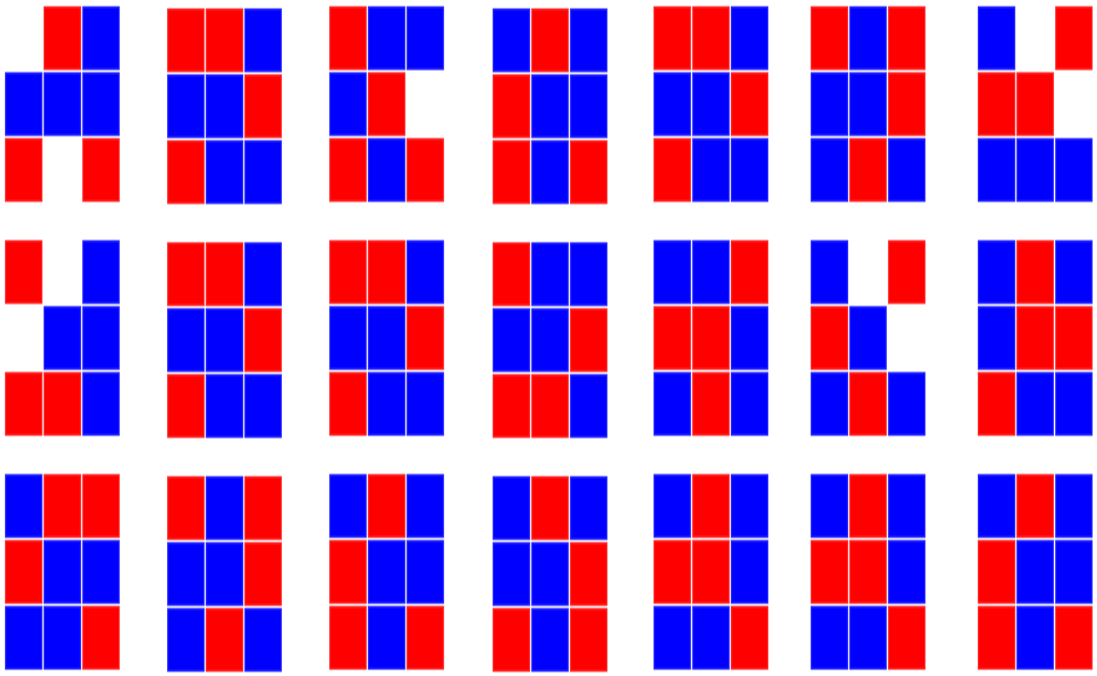

Figure 2.4: Some example games of Tic-Tac-Toe, (Note that X's and O's can be represented as colours instead)

Chapter 3

Prototype

3.1 Overview

In the following chapter I will outline my MCTS implementation, as well as a slightly modified version I call TDMCTS (Temporal-Difference Monte-Carlo Tree Search). I will then outline the reason for using TDMCTS over MCTS, as well as any modifications made in my implementation which also includes some optimizations made to the algorithm as well as state representations. I will then end the chapter with a discussion on improvements that could be made to improve the algorithm. All the code written for this implementation is in python, but of course can be implemented in other languages too. All code can be found on my github: https://github.com/MrSwayne/Reinforcement-Learning-for-Connect4

Figure 3.1: Work done to date

3.2 The Board

3.2.1 Representation and Bits

Originally, my Connect4 board was represented using a Numpy 3D array of ints, but it has since evolved and is now represented as a 64 bit integer, my implementation is based from (Wojciechowski 2017) but implemented slightly differently. The decision to migrate from a 3D array to a 64bit integer representation was due to performance. Integer can make use of simple highly efficient logical bitshift operations to compare boards and to check the win condition that only take one CPU clock cycle, The use of which resulted in a 25x increase in performance while simulating rollouts in my MCTS algorithm.

Figures 3.2 and 3.3 show how the board is represented, both internally, and externally, respectively. Internally, there is a separate integer to represent each player, which are initialised to 0 at the start of the game. In figure 3.3 you can see there are darker shaded bits for the top row, these are essentially empty bits that will not ever be set, the reason for this is when we do bitshift operations later to check the win of the board, bits can wrap around and mess up the result. As a result, there is an empty row at the top, and empty columns after the last visible column, to fill up the remaining bits from 49 to 64, as our number is a 64bit integer. Similarly the BitBoard can be used for the game Tic-Tac-Toe as the two games themselves are very similar, which actually only requires a 16 bit number to represent each player's individual state.

Figure 3.2: How each board is represented in memory, the red numbers represent the bit that is turned on. Note the correlation between the bits in Fig 3.3

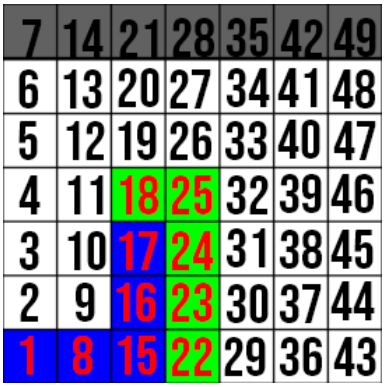

Figure 3.3: A visual Representation of a Bitboard, note the "extra" bits at the top

3.2.2 Actions to Bits

As we have established earlier, the agent is only given a list of actions (i.e. [1,2,3,4,5,6], representing each column that has a free slot), how do we translate these actions to bits in the string? I wanted to come up with a general solution for solving this which can be used for any number of rows and columns. Assuming [Row = 0, Column = 0] is the top-left most cell, and [Row = Rows - 1, Column = Columns - 1] being the bottom right (I chose this format because the board is printed out to the console from left-to-right, top-to-bottom). Getting the bit index can be found through the following formula: Bit = (**R** - **r**) + (**c** (**C** - (**C** - **R** - **n**))) - 1 With: **R** = Number of Rows, **C** = Number of Columns, **r** = row index, **c** = column index, **n** = Number of empty rows

In our case, n is set to one, as we only need one hidden row. xde3 3 ff v8hub x

The following function, written in python, shows how to set and reset a selected bit.

```
1    def set_bit(self, player, bit, turn_on=True):
2        if turn_on:
3            self.boards[player] |= (1 << bit)
4        else:
5            self.boards[player] &= ~(1 << (bit))
```

This means that we can calculate the bit that needs to be flipped just

from the row and column, this makes it very easy to translate between agent's actions and the actual representation of the board.

3.2.3 Checking Win Condition

The checking of the win condition is where the true power of BitBoards come into play, instead of having to search for a certain amount of neighbours in each direction and tot them up to check if a player has 4 in a row, BitBoards can be heavily manipulated to check all positions in the board in parallel. My function is based on (Wojciechowski 2017), but adapted as a general solution for any number of discs in a row. Note the directions array, these can be easily calculated by examing figure 3.3, and noting the algebraic difference between each direction you want to check. For example, when checking vertically; the bits are only one apart, while checking horizontally, the bits are (ROWS + 1) apart due to the extra row on the top. Similar calculations can be done to get the two diagonals.

```
1
2       #Bitboard black magic
3       def check_win(self):
4           ##Vertical |, horizontal -, diagonal \\, diagonal /
5           directions = [1, self.rows + 1, self.rows, self.rows + 2]
6
7           #Only need to worry about the player who placed last
8           player_to_check = self.get_player_turn(prev=True)
9
10          board = self.boards[player_to_check]
11
12          for direction in directions:
13              m = board
14
15              #Loop for however many discs we need in a row
16              for i in range(self.win_span):
17
18                  #Shift the board i amount of columns/rows
19                  m = m & (board >> (direction * i))
20
21              #If after the board is shifted and logical AND'ed
                    together >= 1, that means there is 4 in a row
```

```
            if m:
                self.game_over = True
                return player_to_check

        #If there are no actions, then it's a draw
        if(len(self.get_actions()) == 0):
            self.game_over = True
            return 0
        #Game is ongoing
        return -1
```

Looking at the code it is quite hard to understand intuitively what is going on. If you have a look at fig 3.4 and fig 3.5, note the link between the numbers on the board and the numbers under the individual states on the right. Notice the number of bits that are shifted are based on the distance between the squares you want to check. For example, in figure 3.4 there is only a distance of one between each of red's squares, therefore the board is shifted to the right by one, then anded together. If the result is greater than one, then the game is over. Similarly in figure 3.5, there is a distance of eight between each square diagonally, therefore the board is shifted to the right eight bits, then anded together. If you wanted to make the win condition five in a row then you would shift it to the right one additional time, conversly you can shift it one time less if you would like it to be three in a row for games such as Tic-Tac-Toe.

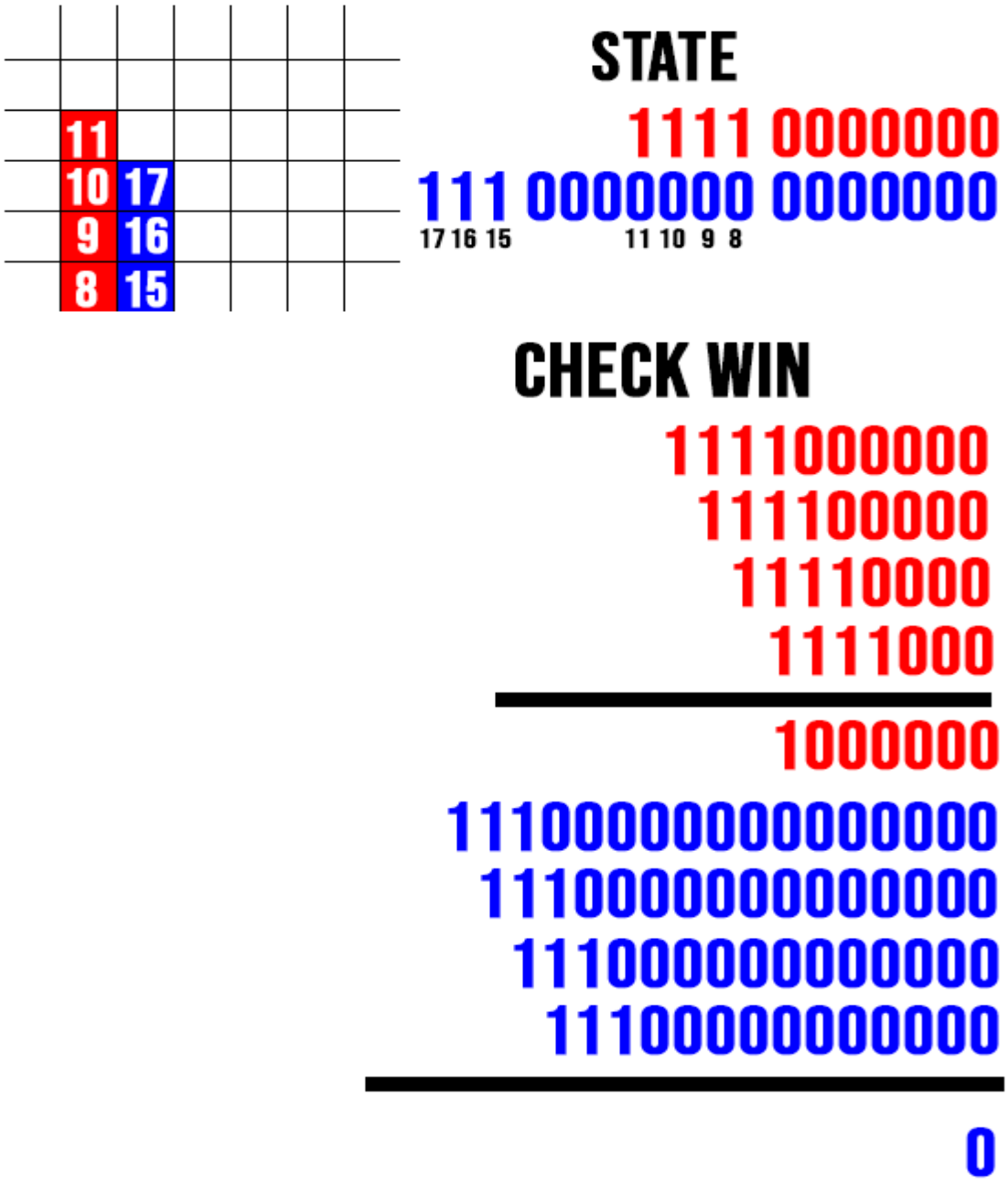

Figure 3.4: Bit-wise checking of state win for a vertical four-in-a-row, note in this case, red has won because the result is greater than 0

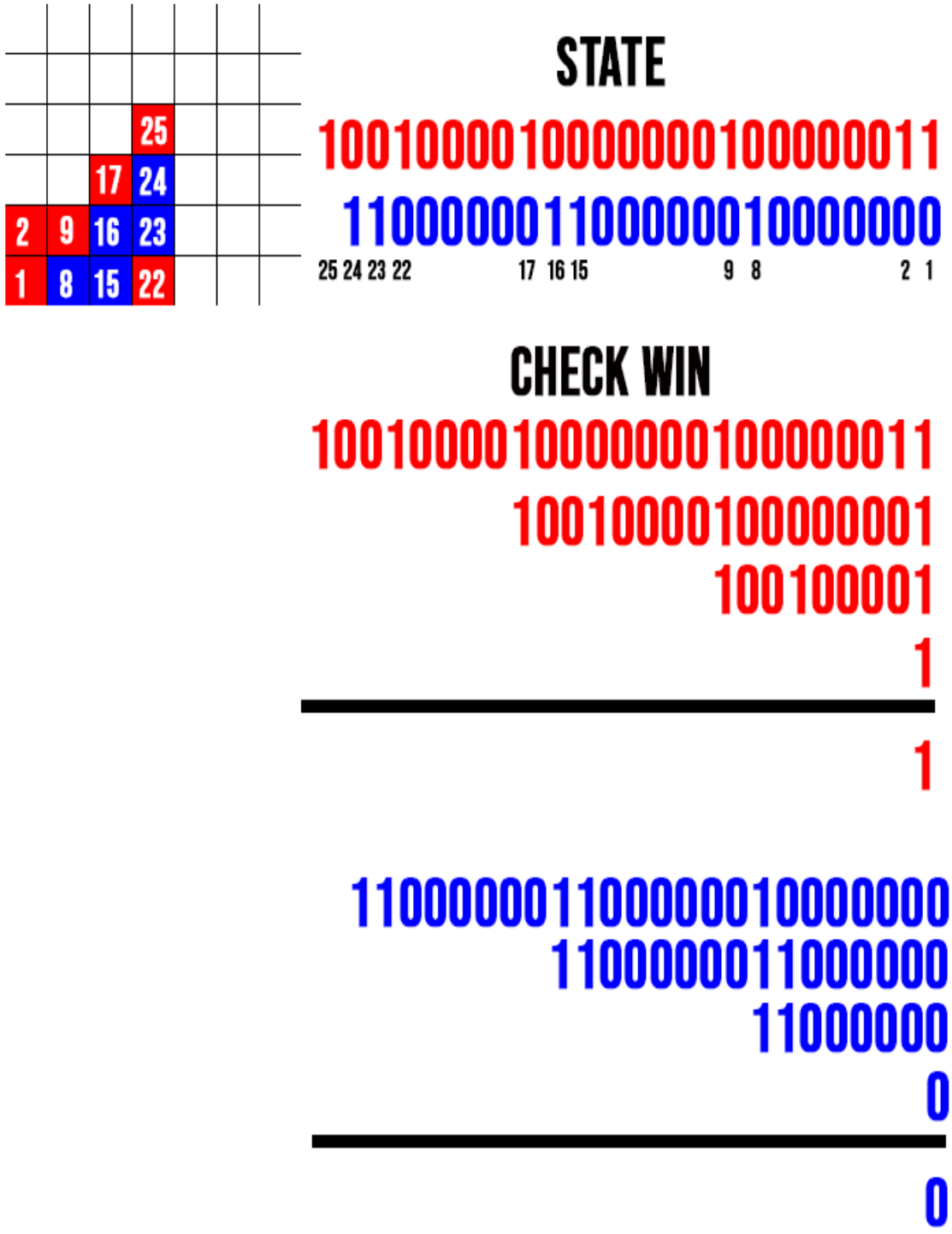

Figure 3.5: Bit-wise checking of state win for a diagonal four-in-a-row, note in this case, red has won because the result is greater than 0

3.3 GUI

Figures 3.6 and 3.7 show the rudimentary front-end I have developed as part of my project. As you can see, because of the nature of bitboards, the GUI for connect4 and Tic-Tac-Toe are essentially the same.

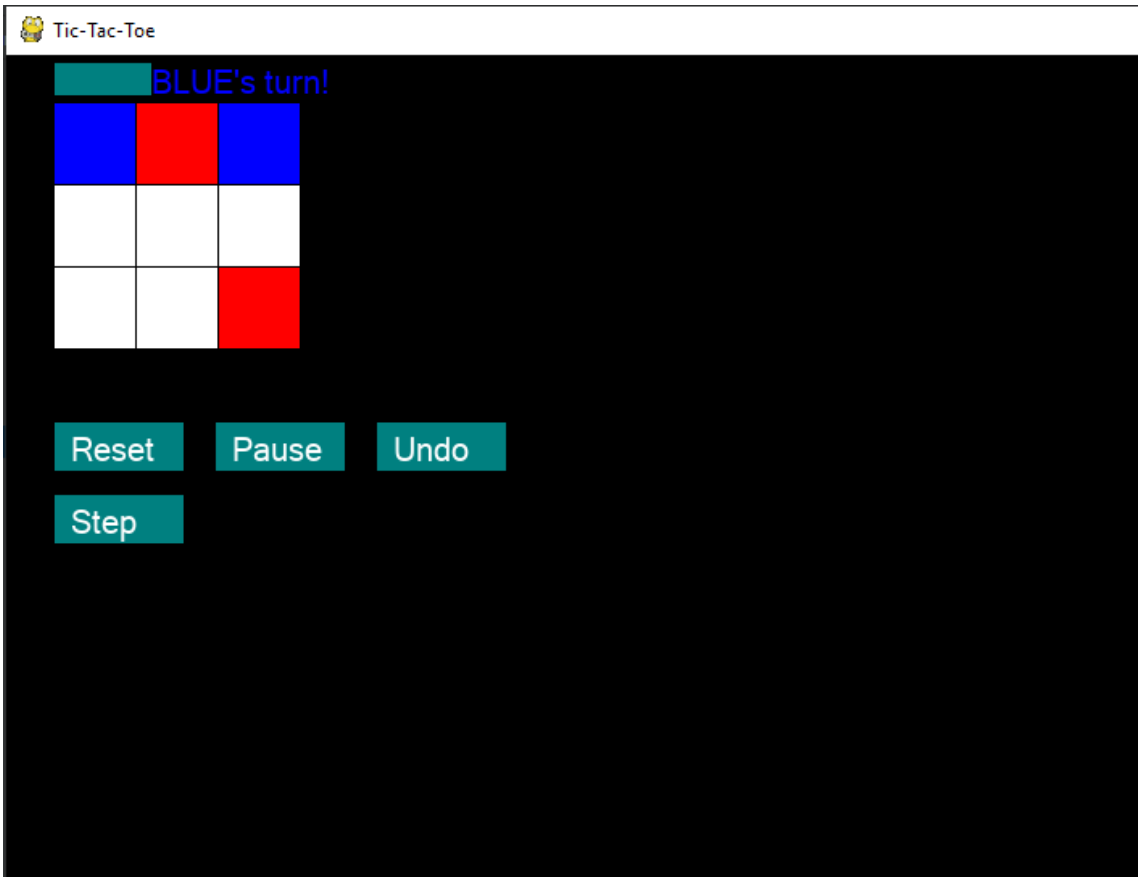

Figure 3.6: Front-end GUI for Tic-Tac-Toe

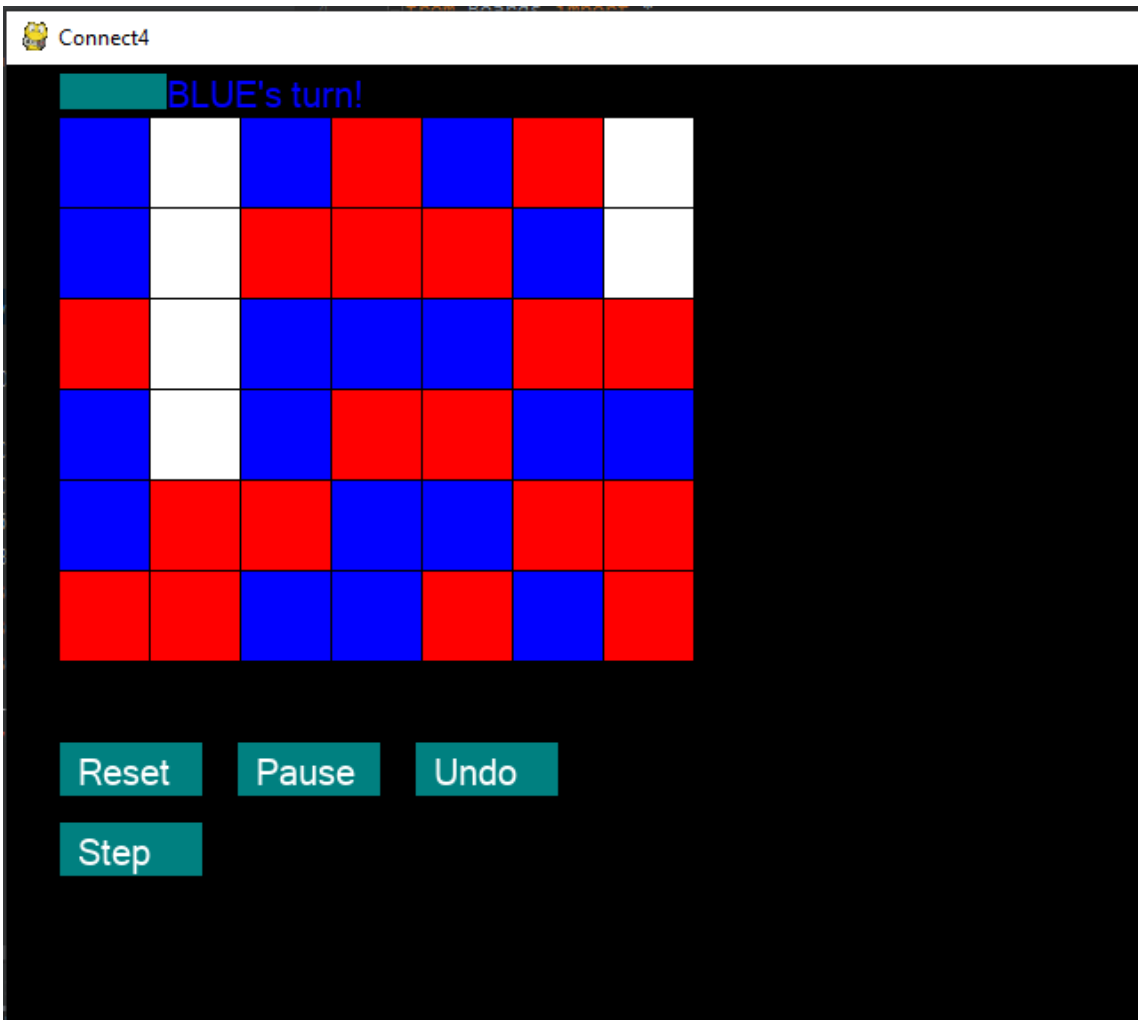

Figure 3.7: Front-end GUI for Connect4

3.4 Monte Carlo Tree Search Implementation

In this section I will explain in detail my implementation of MCTS, bear in mind that all the code presented here is my own, unless otherwise stated, and reasons for implementing it in such a way will be discussed briefly along with papers that I got inspiration from. I will first introduce regular MCTS, and then show you the tweaks that I have made to the functions to incorporate temporal difference learning into them.

3.4.1 The Algorithm

So in essence, the main MCTS algorithm (see Listing 3.1) is quite simple and easy to understand. First of all, the current board configuration (state) is passed to the algorithm (this initial state is then set as the root node of the tree), then the should_continue() function is called, this returns true or false based on initial computational constraints specified when initialising the algorithm. For example, the user has an option to supply either a time duration, or the amount of simulations, n. The idea is that the algorithm terminates once the constraints have been exceeded.

If the algorithm is within its computational constraints, then a node (a recursive wrapper for each state, includes variables such as visit count, score, value, the list of children states, the parent state that came before it, the previous action that led to the current state and the current board configuration) is selected based on the *selection policy*. Then, if the state is not a terminal state, an expansion is done. A simulation is then done from that selected node to a terminal state based on the *rollout policy*. The simulation function then returns the reward, as well as the number of steps taken to get to the aforementioned terminal state. After all that, the reward is backpropagated up through the tree.

After the computational constraints have been exceeded, the best action is selected from the current state, and returned based on the *child policy*.

Listing 3.1: My MCTS algorithm

```
1
2  def get_move(self, state):
3
4     while(self.should_continue()):
5        self.current_n += 1
6
7        node = select_node()
8
9        if not node.state.game_over:
10          self.expand(node)
11
12       reward, num_steps = self.simulation(node)
13
14       self.backpropagate(node, reward, num_steps)
15
```

```
16    best_child = self.child_policy(self.root)
17
18    return best_child.prev_action
```

3.4.2 The Selection and Tree Policy

The selection policy (Listing 3.2) starts at the root node, and cycles through the tree until it reaches a node that has not been visited. The selection of the best children is called the *tree policy*, and is implemented using UCT (Upper Confidence Bound Applied to Trees), which is used to weigh exploration vs exploitation.

This function is implemented slightly differently in regular MCTS, vs TDMCTS. In regular MCTS, the exploitation part of the equation incorporates total wins to date for the node, divided by the total visits to the state. In TDMCTS, the current value (-1 ¡= V ¡= 1) of the state is used as the exploitation part. (See section 2.5.2 for details on the original formula). The inspiration for incorporating the Temporal-Difference value instead can be attributed to (Vodopivec & Šter 2014)

Listing 3.2: The Selection Policy

```
1  def select_node(self):
2         node = self.root
3
4       while (len(node.children) != 0):
5          node = self.tree_policy(node)
6
7       return node
8
9  def tree_policy(self, node):
10    max_score = float("-inf)
```

3.4.3 The Child Policy

The child policy, as mentioned previously, is how the best child is determined from a node. As stated previously, There is a distinct difference between the child policy and the tree policy, and that is, with the child policy; exploration is set to 0. The reason for this is because when the best action is being

returned from the algorithm, we don't want it to explore other actions if it can already win immediately from the current state.

There are other ways to implement this child policy in the base MCTS algorithm as mentioned in section 2.5.3.

3.4.4 Expansion

The expansion function is quite easy to get your head around (Listing 3.3), it loops through all available actions from the current state, creates a child node for each successive state, and then appends them to the parent state's children list. You can see there are extra variables sent to the child, like the depth, the current player that the node represents, as well as data and a learn variable. The data variable is a reference to the tree data, updating the node in turn updates the tree data. If the learn variable is set to false, the tree data is not updated, which is useful when the algorithm needs to differentiate between training mode and tournament mode.

Listing 3.3: The Expansion function

```
1  def expansion(self, node):
2     node.create_children()
3
4  #Located inside the Node class
5  def Node.create_children(self):
6     self.children = []
7
8     for action in self.state.get_actions():
9        _state = deepcopy(self.state)
10       _state.place(action)
11
12       child = Node(
13               parent = self,
14               state = _state,
15               player = state.get_player_turn(),
16               prev_action = action,
17               depth = self.depth + 1,
18               data = self.data,
19               learn = self.learn
20            )
21
```

```
22        self.children.append(child)
```

3.4.5 Simulations and The Rollout Policy

The Simulation or rollout phase (Listing 3.4) can be broken down into two separate functions. A terminal state is reached based on the rollout policy, in this implementation I incorporated the use of light rollouts. The reward is then calculated based on the outcome of the terminal state (a win=1, draw=0 or a loss=-1). This is then returned to the main function for the backpropagation phase.

Listing 3.4: The Simulation and Rollout Policy implementation

```
1
2  def simulation(self, node):
3     terminal_state, num_steps = self.rollout_policy(node.state)
4     reward = self.reward(node, terminal_state)
5
6     return reward, num_steps
7
8  def rollout_policy(self, state):
9     temp_state = deepcopy(state)
10
11    steps = 0
12    while not temp_state.game_over:
13       actions = temp_state.get_actions()
14
15       if len(temp_state.get_actions()) > 0:
16          temp_state.place(random.choice(actions))
17          steps += 1
18       else:
19          break
20    return temp_state, steps
```

3.4.6 Backpropagation

The backpropagation phase is probably the most important part of the entire algorithm. My TDMCTS algorithm went through various iterations as I dug

deeper into the literature. Originally, I didn't like the fact that the tree data is usually thrown out after the tree

process, with the temporal-difference learning part only learning from states that actually occur in games. (See section 4.3 for more information). I first decided to implement a Temporal-Difference single backup from (Vodopivec & Šter 2014), but overtime I modified it very slightly based on my own understanding from research done.

(Listing 3.5) is my current implementation for MCTS. The reward (-1 ¡= R ¡= 1) is negated at every step to account for the opponent's move. Therefore, a win for a player would be backpropagated through that player's nodes, and would cause a loss to be backpropagated through the other player's nodes. Of course, the score added is +1 for a win, +0.5 for a draw, and +0 for a loss.

(Listing 3.6) is my current implementation for TDMCTS. First of all, the reward is discounted by the discount factor (γ) based on how many steps it took to get to the terminal state (the idea being, states closer to the terminal state reached would have a higher value associated to it, while states further away would be more discounted).

The learning rate (α) was originally scaled based on how many visits the node/state had. The idea being a tweak to a state that was visited 100,000 times would be less than a state that was only visited 10 times. The hope is that the decaying learning rate would cause the values of each node to converge toward the optimum value. What I found while experimenting was that if the learning rate became too small; many training episodes were wasted doing the exact series of moves. The reasoning for this repetition I suspect was because as the learning rate decayed, the change in value to the state eventually became too miniscule to make a noticeable difference. If player one found a new way to win, then player two would do the same series of moves many times until the aforementioned miniscule change propagated through many games, eventually causing it to learn that those series of moves are bad moves. Conversely, if this learning rate was capped at a value that is not low enough, then it would lead to unstable learning, where its winrate would change drastically between episodes. I found that setting the alpha equal to the maximum value based on the number of visits, and some specified value (self.a) was the best way to get the best of both worlds, with this specified value passed during the initialisation of the algorithm. I experiment with this changing value in 4

Listing 3.5: My MCTS backpropagation phase

```
1
2  def backpropagate(self, node, reward, num_steps):
3
4     while node is not None:
5        if reward >= 1:
6           node.score += 1
7        elif reward == 0:
8           node.score += 0.5
9     reward *= -1
10    node.visit_count += 1
11
12    node = node.parent
```

Listing 3.6: My TDMCTS backpropagation phase

```
1
2  def backpropagate(self, node, reward, num_steps):
3     reward *= (self.gamma ** num_steps)
4
5     alpha = max(1 / (node.visit_count + 1), self.a)
6     node.V = node.V + alpha * (reward - node.V)
7
8     while node is not None:
9
10       node.visit_count += 1
11
12       if node.parent is not None:
13          target = -(self.reward(node.parent, node.parent.state) +
                self.gamma * node.V)
14
15          alpha = max(1 / (node.parent.visit_count), self.a)
16          node.parent.V = node.parent.V + alpha * (target -
                node.parent.V)
17
18       node = node.parent
```

3.5 The Adversary

In order to verify that this TDMCTS algorithm is actually learning, it needs to be played against a fixed opponent. That is, the TDMCTS agent has to train for a certain amount of episodes, then after a certain batch of training episodes it should be put against this fixed opponent. I decided that a good opponent to put it against would be Minimax, and regular vanilla MCTS. The reason why I chose Minimax as the adversary is because it has been shown that the MCTS tree, when given enough time, eventually converges to a Minimax tree (Bouzy 2007). Another reason for its use is because of how easy it is to implement. My implementation of Minimax implements AlphaBeta and is based on code from (Vinayb21 et al. n.d.), but the heuristic evaluation function is different and entirely mine. The reason I chose MCTS as a secondary opponent was to not only get more objective results, but to also show the power of this TDMCTS algorithm over regular MCTS, as well as that, it was far less computationally expensive to use MCTS as an adversary for longer experiments, as Minimax's search branches almost exponentially the farther you look ahead.

I admit that the evaluation function could be better crafted, as there were issues with it as seen in section 4.2.2. What I found was that because the implementation was again based off of bitboards, I failed to produce a proper evaluation function that made use of the efficient bit-wise operations. This resulted in a O(R * C * W) (R = rows, C = columns, W = amount needed in a row) evaluation function, which was handcrafted to weigh states based on how many disks were in a row versus how many the other player had.

3.6 Keeping Track of Experiments

In order to keep track of all the experiments that I intend to do, I developed a system whereby the program would read in a bunch of config files, configured with certain paramaters for the training and tournament games, as well as paramaters on where to save these files. This sped up the process immensely and meant that I could not only run multiple config files at once, but I could train and play tournament games in parallel. This system also allowed me to run different experiments in parallel as well. I also had an option whereby the user could specify if the board generated should be a Connect4 board, or a Tic-Tac-Toe board.

Listing 3.7: An example Config.ini file. Note the hyperparamaters, as well as the types of algorithms, leading to high flexibility with these experiments

```
1
2
3  [GENERAL]
4
5  #select the Mode that you'd like to run options: SIMULATION,
       TRAIN, PLAY
6  mode = SIMULATION
7  seed = 42
8  board = TicTacToe
9
10 #Run a simulation in a tournament setting.
11 [SIMULATION]
12 players = PLAYER_1,Minimax
13 episodes = 100
14
15 #If this option is set, it can be used to iteratively play
       tournament games with every file in a directory, useful for
       parallel tournament and training experiments
16 iter = PLAYER_1
17
18 #Whether or not the agent should learn from this experience
19 learn = false
20
21 [TRAIN]
22
23 #Amount of episodes to train for
24 episodes = 5000
25
26 #After how many episodes will there be tournament games
27 batch = 200
28
29 #How many tournament games between players and enemy
30 tournament_games = 100
31 players = PLAYER_1,PLAYER_2
32 enemy = Minimax
33
34 #Config for TDMCTS
35 [PLAYER_1]
```

```
36
37 #Type of algorithm, choices: Random, TDMCTS, Minimax, MCTS,
       AlphaBeta
38 type = TDMCTS
39 colour = BLUE
40
41 #Hyperparamaters
42 exploration = 0.5
43 learning_rate = 0.005
44 n = 100
45 memory = ttt/e05_a005
46
47 #Config for TDMCTS
48 [PLAYER_2]
49 type = TDMCTS
50 colour = RED
51 exploration = 0.5
52 learning_rate = 0.05
53 n = 100
54 memory = ttt/e05_a005
55
56 #Config for Minimax
57 [Minimax]
58 type = ALPHABETA
59 colour = GREEN
60 depth = 6
61 use_heuristic = true
62
63 #Log file location
64 [LOGGING]
65 file_name = ttt/tdmcts_vs_minimax
66
67 #Location of log and memory files
68 [IO]
69 data_path = data/
70 log_path = logs/
```

3.7 Validity of Prototype

The question can be raised, am I sure I created a valid prototype? Although I didn't do any proper scientific validity studies, I merely checked it by playing against it personally. I spent months on this prototype, constantly running it against Minimax, and standard Monte Carlo Tree Search until it was strong enough to beat the algorithms.

When I first developed the GUI for the project, I gave it to six close friends of mine, who played it in a best of five. Now, they're obviously not Tic-Tac-Toe or Connect4 masters by any means, but the algorithm beating them in every single game bar one, was enough evidence to me that it at least had a strong enough beginner intelligence from the get-go. That one game that it lost, I analysed the log files and tweaked the exploration rate slightly, and ran the games again against the close friends, and the algorithm was succesfull in beating them.

After that phase ended, I decided to play it online, I used the website "https://poki.com/en/g/connect-4", which provided an interface to play Connect4 against other players, as well as play it against the AI they have on the website (which I am fairly certain is Minimax up to a certain depth depending on the difficulty). The AI, was able to beat the AI easily, but had some trouble against players. There were some players it would beat easily, and some not so easily, I played around 10 games overall, and it won 6, which, to me, was enough for the purpose of validating the prototype.

3.8 Timeline

I finished the implementation of my prototype by about week 8 in the semester (which is around the middle of March), and then spent the end of March to the start of May designing, running, and analysing the experiments ran. There were few complications with regards to data corruption which set me back a small bit but overall I feel like with the time that I had, I did as many experiments as I possibly could, running some even in parallel to try and speed up the process.

3.9 Extensions and Future Work

Although my learning agent can achieve a very high winrate against Minimax at depth 6 (See chapter 4 for results), there are still a few shortcomings of the algorithm that I will discuss.

3.9.1 Upgrades to BitBoard implementation

I think a good way to improve the implementation of BitBoards is to add the ability for transposition. That is, mirrored states will share the same value. This not only reduces the memory footprint of the algorithm; but also reduces the amount of states that need to be visited as in the early stages in the game many of the first initial actions can be mirrored.

Furthermore, I ran a python profiler on the code as it ran. What I noticed was that the most called function was the deepcopy operation of the board configuration. A deep copy is when the board is essentially cloned, so that any change made to the clone does not affect the original board, this is vital for simulations to effectively sample the space ahead. This deep copy operation in itself is expensive and I can think of a couple of ways to improve the running time of the algorithm. I think that a good way of cutting down considerably on the amount of deep copies is to make a slight modification to the expansion function (See Listing 3.3). As it currently stands, every time a node is visited, all its children are created immediately. This effectively leads to a waste of deep copies as some nodes will be created, but never visited as the original value from the first simulation of the parent could be so bad that the selection policy never selects that node again.

Another way to cut down considerably on the time is to calculate the board configuration based on the given states of each player, removing the need for any deep copies at all.

3.9.2 Memory Replay

Memory replay has seen success in the literature (Ruishan & James 2018) (Andrychowicz et al. 2017). The idea is that because of the inherent nature of the backpropagation phase in the Monte Carlo Tree Search algorithm, states are "temporally dependent" on each other, that is, at each backpropagation step the reward is backpropagated throughout all states that led up to the current point.

This sounds OK in theory, but that is not how a human learns. When a human does a task and let's say for example it fails the task, does the human disregard all the steps leading up to the end of the task as a failure? Or does it learn what was good and what was bad? The latter of course is the intuitive way in which we learn, and can be achieved by randomly choosing games that the agent has played, and starting at a random board position for that game, and continuing on from here. In this way it can eventually learn what subset of moves were good moves and what subset of moves were bad moves.

3.9.3 Rollout Policy

I will briefly explain the use of neural networks in the next section with regards to the selection and rollout policies. Even without generalisation, there can be a few improvements made in order to improve the rollout policy. A heuristic evaluation function could be crafted in order to chose moves that are more likely to occur in a game. This has the downside of reducing the generality of the algorithm as an evaluation function must be crafted for each game.

Perhaps the selection policy could be applied to the rollout policy down to a certain depth. The idea behind it being that the algorithm could evaluate the next immediate moves that the opponent can make. In this way, it could reduce the "horizon effect" which is a known problem in the algorithm. The horizon effect is where there could be a move not evaluated properly that the algorithm has missed, which could lead to an instant loss, this is due to the inherent randomness of the algorithm, whereby if there is a single move that leads to a loss, and six moves that don't, the odds of that move being randomly picked is inherently low.

3.9.4 Generalisation

The generalisation of states has seen vast improvement in gameplay performance as seen in (Silver et al. 2016) and (Tesauro 1995). I found that my algorithm seemed to excell in states in which it has seen before, but lacks the ability to generalise between unseen or similar states, so when put into a state it has not seen before, all its previous knowledge is not utilised and it is basically acting like a brand new agent with 0 episodes of training done. I propose that a neural network could be trained on the tree data after a period

of training, and then this neural network can either be used as a policy network, whereby the network will replace the selection policy, which outputs what action to take at each time step in an actual tournament game.The network could also be trained as a value network, this means that it could replace the use of light rollouts in the Monte Carlo Tree Search simulation phase by selecting more probable states than just complete randomness.

Chapter 4

Empirical Studies

In this chapter I am going to discuss results that I have gotten through episodes of training with the algorithm. It's clear from figures 4.1 and 4.2 that the algorithm has seen significant improvement in its performance over-time, evident in both Tic-Tac-Toe and Connect4.

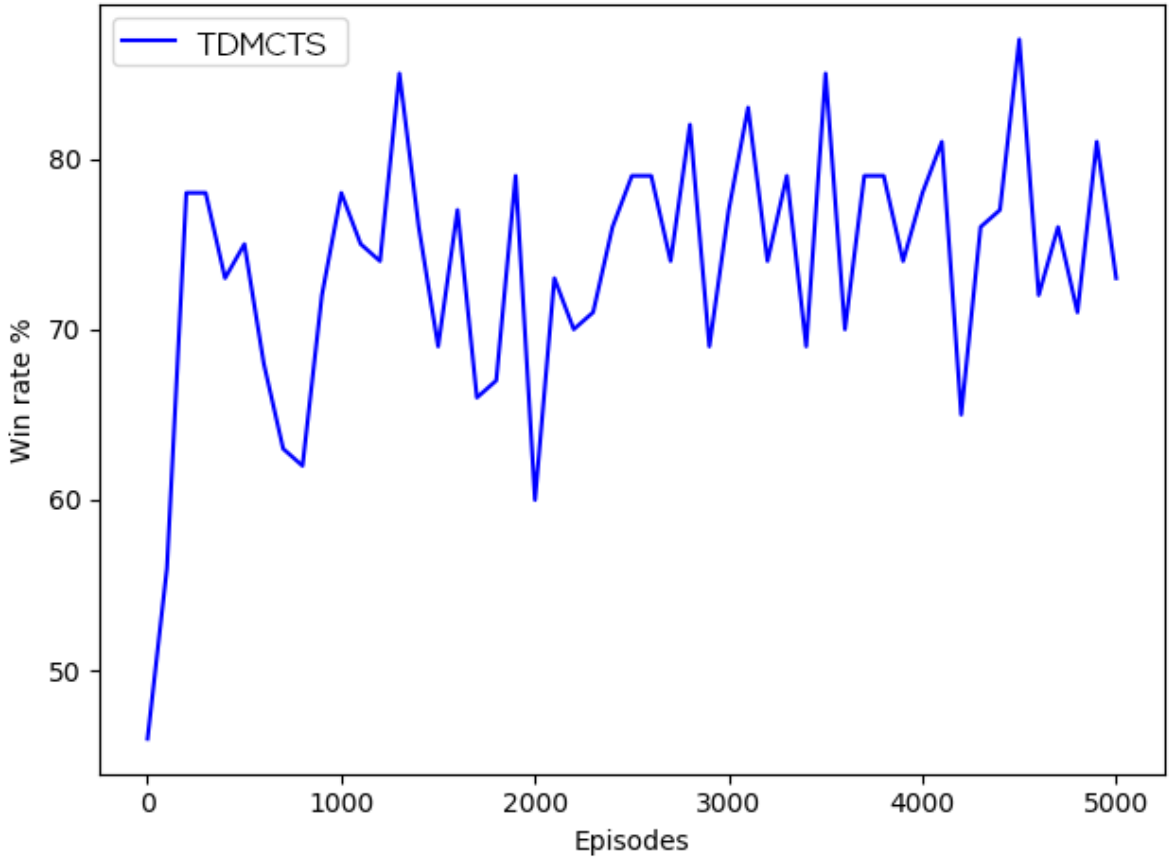

Figure 4.1: TDMCTS, n=200, e = 0.5 against Minimax depth 6, for the game Tic-Tac-Toe

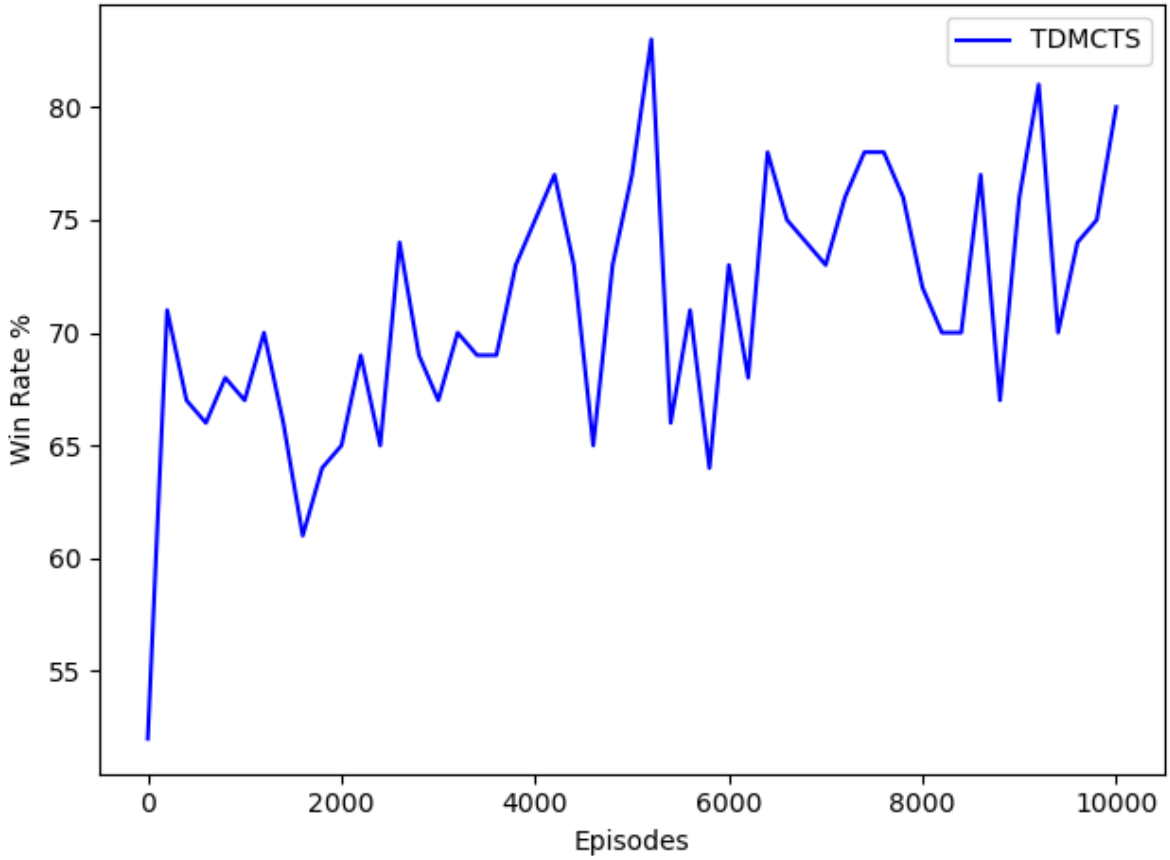

Figure 4.2: TDMCTS, n=200, e = 0.5 against Minimax depth 6, for the game Connect4

4.1 Infrastructure

I incorporated the use of Google Cloud Compute Engine in order to do all my training as the hardware that I had originally at home was simply not powerful enough to do training sessions in parallel, I also found it hard to keep my hardware on for days at a time without serious overheating issues. The data was then sent from the cloud to my computer in order to do further analysis on the data gained from training and simulating.

Eventually I ended up getting new hardware, which allowed me to train

on my own computer at home, as well as in the cloud, with a share drive allowing my information to be synced across machines.

There were many different iterations to my TDMCTS algorithm. The original was based off of (Vodopivec & Šter 2014) TD-UCB, in which a single target was backed up to all other nodes in the backpropagation phase, the only difference being without the use of eligibility traces. This algorithm has developed over time, as seen in section 3.4.

The training paramaters, are provided to the config file discussed in section 3.6, in order to ensure the consistency of results.

All in all, each experiment took about one week to run, and if I had access to better hardware I would have loved to have trained the algorithm a lot longer.

4.2 Games vs Adversaries

In the following section, I am going to discuss the various results I got regarding the algorithm's performance vs different AI's. I found quite a big similarity in win rates between Tic-Tac-Toe and Connect4, and so when I am discussing the win rate, I have taken into account the both games, unless otherwise stated.

4.2.1 TDMCTS vs Random Player

Originally I decided to play TDMCTS vs a completely random opponent. What I found out was that with n=100, e=0.5, TDMCTS was able to achieve a 100% winrate with **no** learning whatsoever. This is a good indication that TDMCTS itself is able to learn quickly enough in the playout and backpropagation phase to lead it towards a win with such little knowledge of the state space, performing much better than a completely random player.

4.2.2 TDMCTS vs Minimax

Originally, I trained TDMCTS vs a depth 4 variant of Minimax, then had tournament games against Minimax at depth 6. Fig 4.3 depicts TDMCTS being pit against Minimax after each round of training. I observed that the Minimax heuristic weighted states the same way every time (the heuristic used involved counting the amount of tokens in a row minus the opponents

amount in a row), therefore TDMCTS was able to learn a set of moves in which it could win every single time. As a result, TDMCTS reached a 100% winrate after 300 episodes. This, to me, seems like TDMCTS had exploited a weakness in my Minimax heuristic evaluation function, and therefore didn't have to do much more learning as it would play the same set of moves in the training phase, and in the tournament phase, and because Minimax doesn't inherently learn from experience, it was powerless against the algorithm.

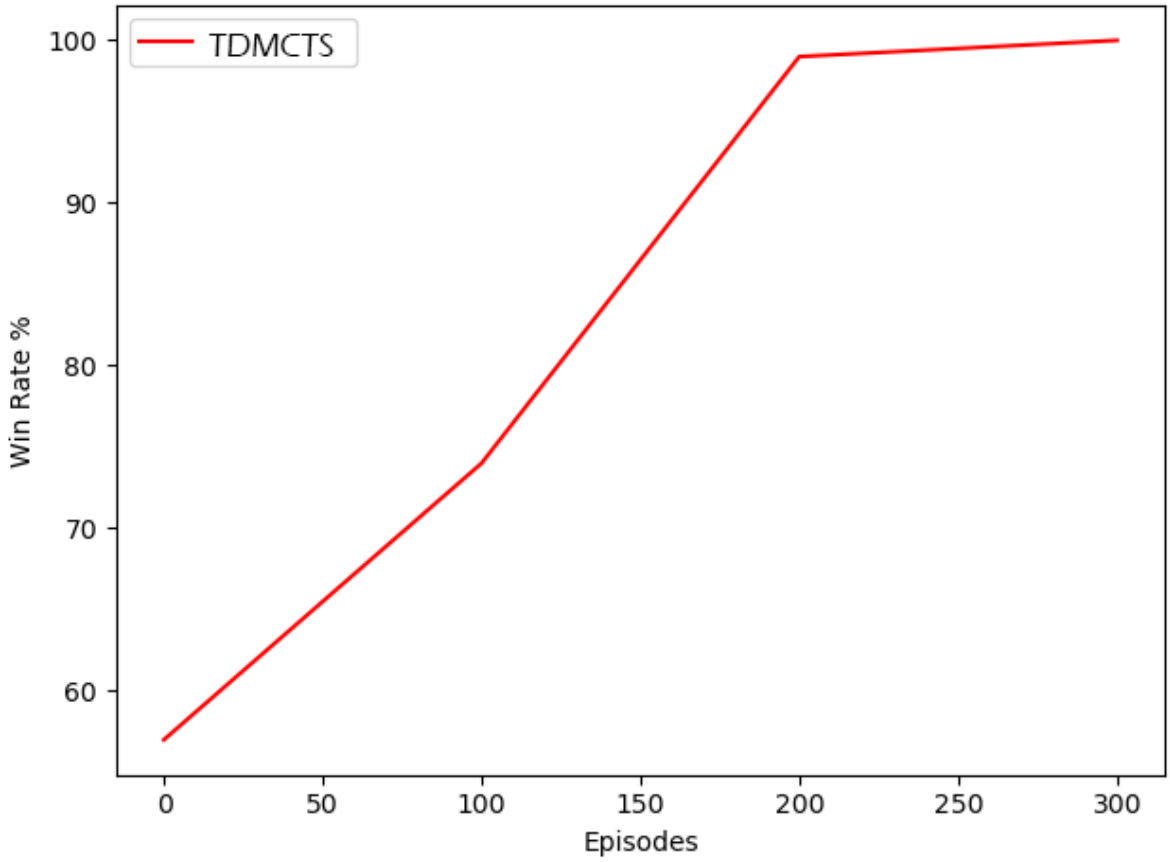

Figure 4.3: TDMCTS, n=1000 exploiting the heuristic used for Minimax depth 4 in Connect4

4.2.3 TDMCTS vs AlphaBeta

After changing the heuristic for Minimax, as well as implementing Alpha Beta pruning to the algorithm, I noticed a huge increase in performance for the algorithm. I then pitted TDMCTS against this new Alpha beta algorithm in the game of Tic-Tac-Toe; in order to see what changes in hyperparamaters do to both the training process, as well as the tournament games. In figures 4.4 and 4.1, you can see the difference that the exploration factor has on the processes.

Remember that the exploration rate paired with UCT tries to balance exploration with exploitation, as a result, in figure 4.4 you can see when the exploration rate is set to a high number (i.e. 1) it tries its best to explore all the state space. This leads it to exploiting its learned knowledge for the first 2500 episodes; then because "sub-optimal" moves haven't been checked enough, it goes through a period of trying out these moves, leading to a sudden drop in win rate. But it quickly corrects itself and after 3000 episodes it is back up to a 80% winrate. You can also see in figure 4.1 that it doesn't have the sudden dip in performance at the 2500 episode mark. It seems to be a much more consistent graph, but what I noticed was at the cost of two things: less of the state space is searched (after 5,000 episodes, e=0.5 explored about 60% of the space that e=1.0 did), and a lower win rate (e=1 was able to reach a peak winrate of 95%, while e=0.5 only reached 85%)

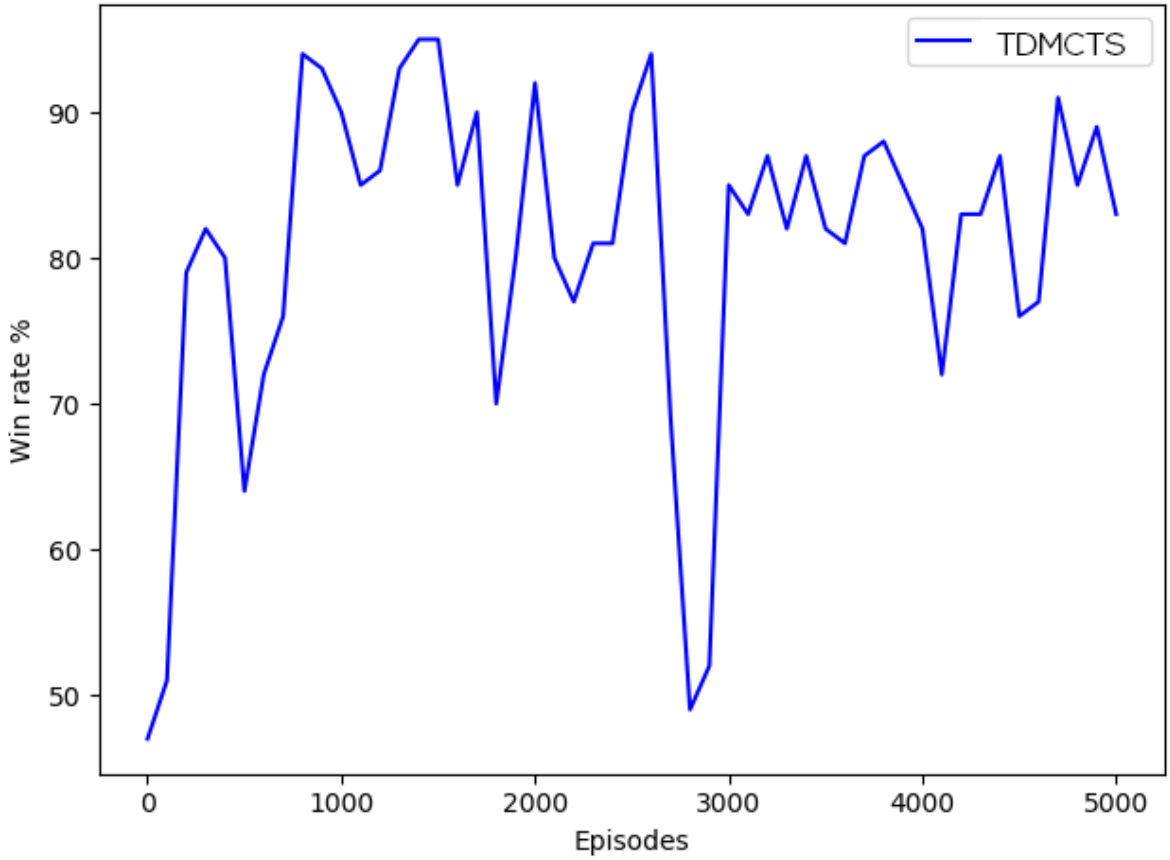

Figure 4.4: TDMCTS, n=200, e = 1, l = 0.9 against Minimax depth 6, for the game Tic-Tac-Toe

4.2.4 TDMCTS vs MCTS

As a result of the setbacks with the Minimax heuristic evaluation function, I decided to try and put TDMCTS against regular MCTS. The outcomes from this process can be seen when assessing the hyper-paramaters of the algorithm as seen in figures 4.8 and 4.10.

4.3 State Space Search

As seen in TDGammon 2.6.1, a neural network is trained using temporal difference learning on the actual states that occur during the training process. By doing that way with Connect4, as seen in fig 4.5, TDMCTS observed 3,580 states after 100 episodes . The amount of states visited can be drastically improved if the results from the tree search are not thrown out after every move. By implementing temporal difference learning in the backpropagation phase, TDMCTS (fig 4.6) sampled over 1,400,000 states.

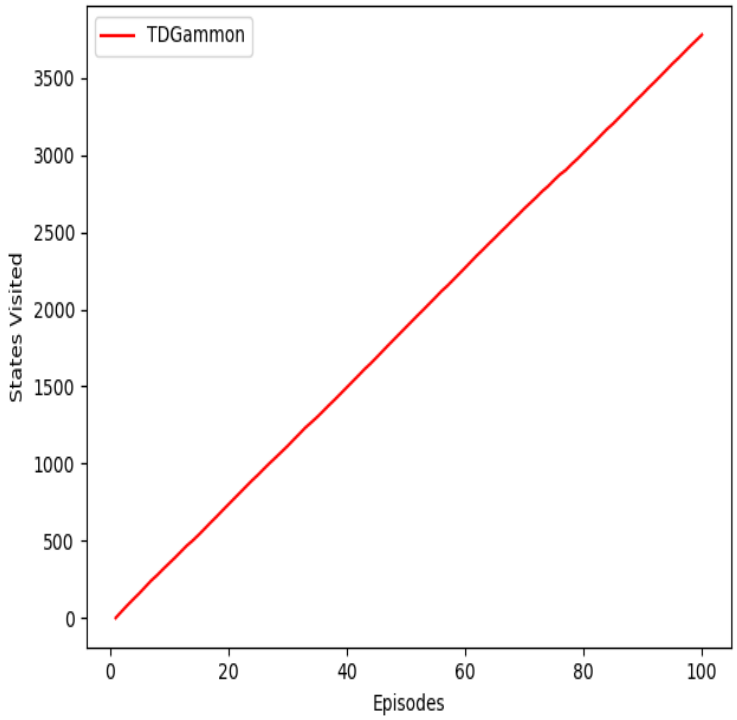

Figure 4.5: TDMCTS learning from actual states that have occurred in training (As seen in Tesauro's approach)

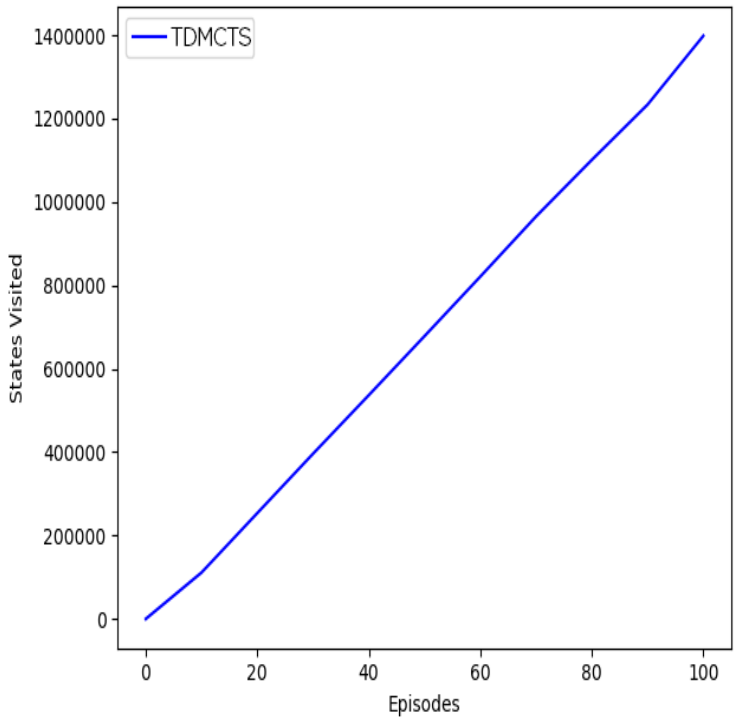

Figure 4.6: TDMCTS, n=1000 learning from states found through the use of tree search

4.4 Impact of Hyper-Paramaters on the Training and Tournament Process

I have shown previously, that the algorithm relies on certain paramaters, which affects its behaviour. In this section I am going to try my best to explain the difference that these paramaters made on the training and tournament process. Many hours were spent trying to tweak these paramaters, as a small change could cause a big change in the behaviour of the algorithm.

4.4.1 Exploration Rate

In this section I am going to detail the difference that the exploration rate paramater ϵ had on with regards to the algorithm's behaviour. As you can

see from the graph (fig 4.8, e = 0.25 seemed to have the highest winrate over all from all the different values, but didn't seem to explore as much of the state space (fig: 4.7. e=0.5 was the next closest to e = 0.25, but seemed to vary much more frequently. While e=10.0 had the lowest winrate overall, which I think is to be expected as it means that it is constantly exploring and never exploiting. Yet, varying values above 0.25 seem to have similar state space exploration to each other, despite having a lower winrate. This leads me to believe that it's more about the quality of the states visited, and not the quantity of the states visited. e = 0.25 also seems to be very conservative in exploring the state space, but eventually, as is evident at the 3,800 episode mark, it finds a move in the training process that is better than it previously thought. As a result, this new move leads to a steady increase in the state space search as it does more and more simulations and expansions for this new move.

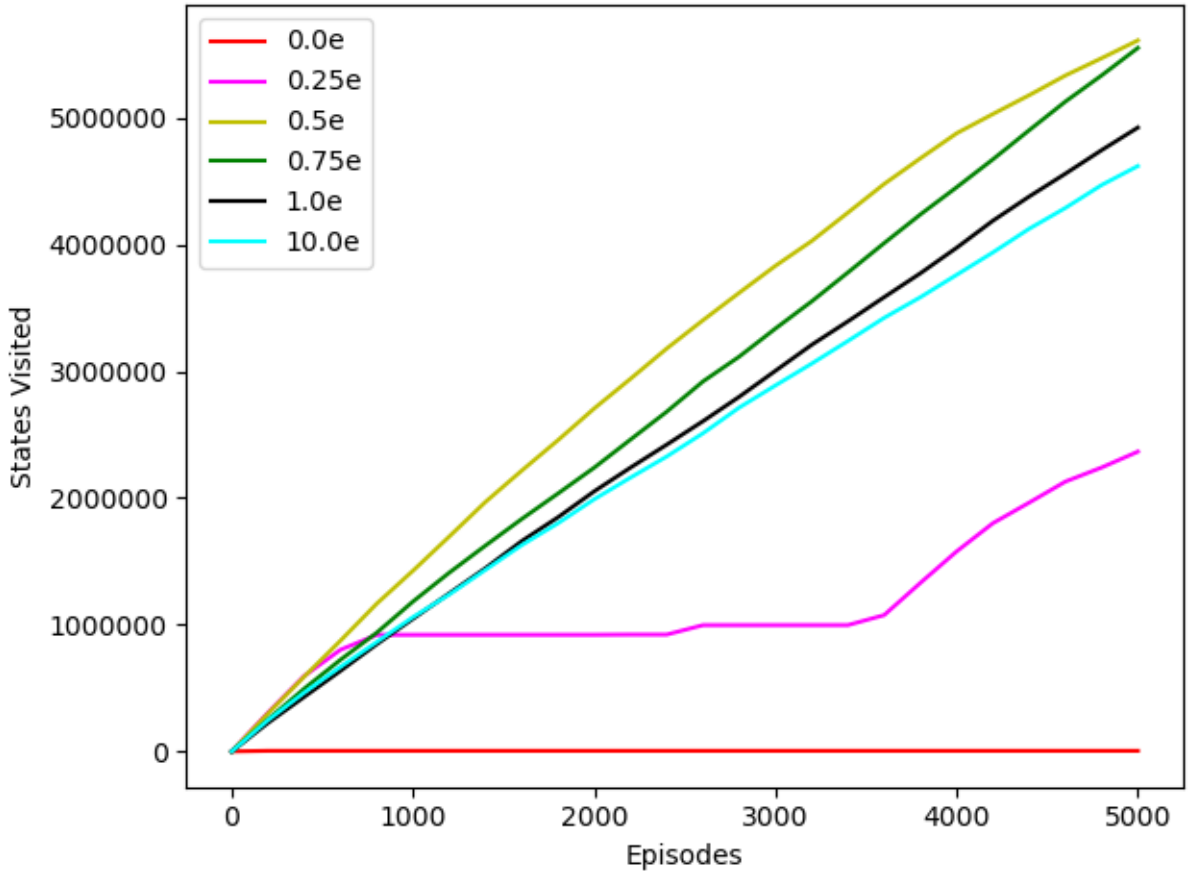

Figure 4.7: The amount of states visited with TDMCTS, n=100, with varying exploration rates

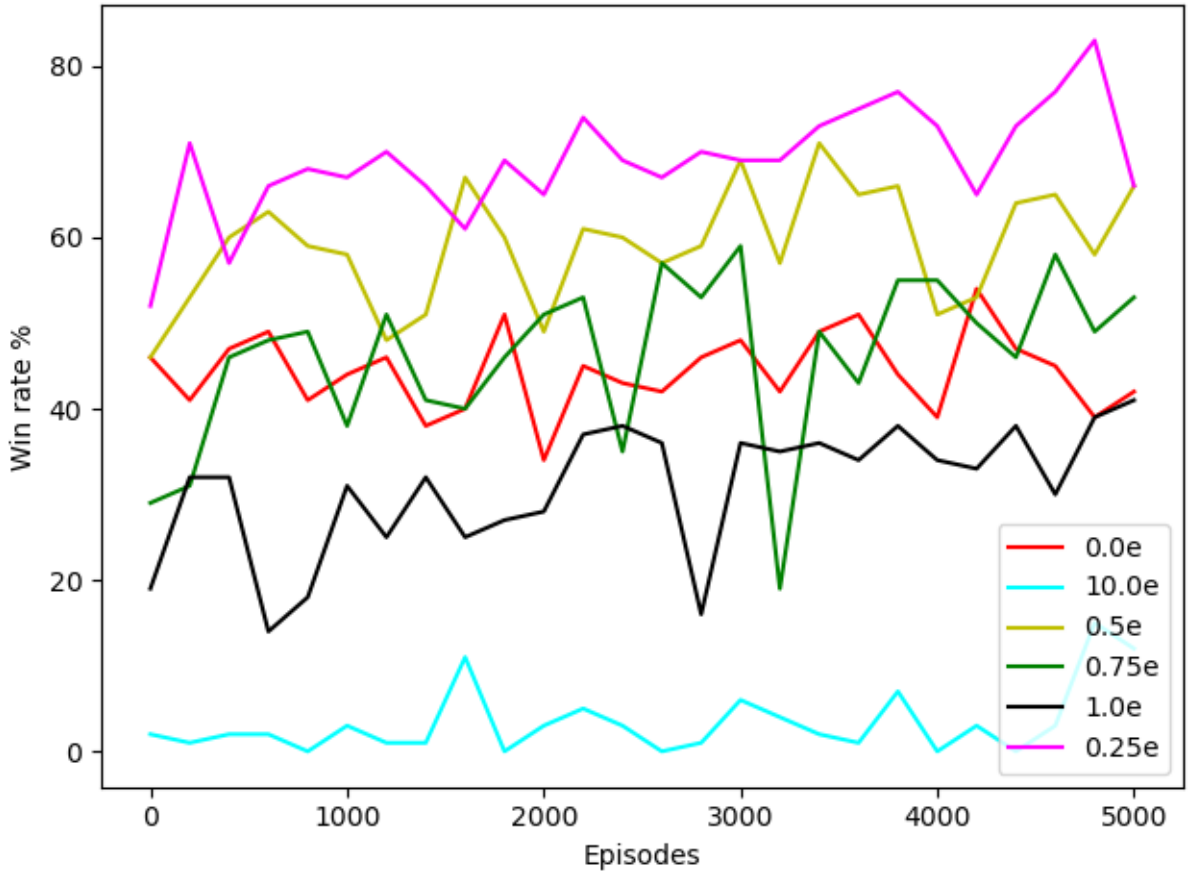

Figure 4.8: The winrates of TDMCTS, n=100, with varying exploration rates, versus standard MCTS, n=100

4.4.2 Learning Rate

In the backpropagation phase of the algorithm, the new value of each node is linearly interpolated between the old value, and the reward obtained. This step size is based on how many times the node was visited. For example, if it is the first time the node is visited, it is stepped 100% towards the value, the second time is then 50% etc., down to a certain decimal figure. Figure 4.9 shows at what value the learning rate was capped at, and how it affected the amount of states visited. Fig 4.10 show the win rates with varying levels of the cap on the exploration rate. As you can see, lower values of the learning

rate definitely perform better, but to a certain point. With a learning rate cap of 0.0005 I was able to achieve around a 75% winrate against standard MCTS.

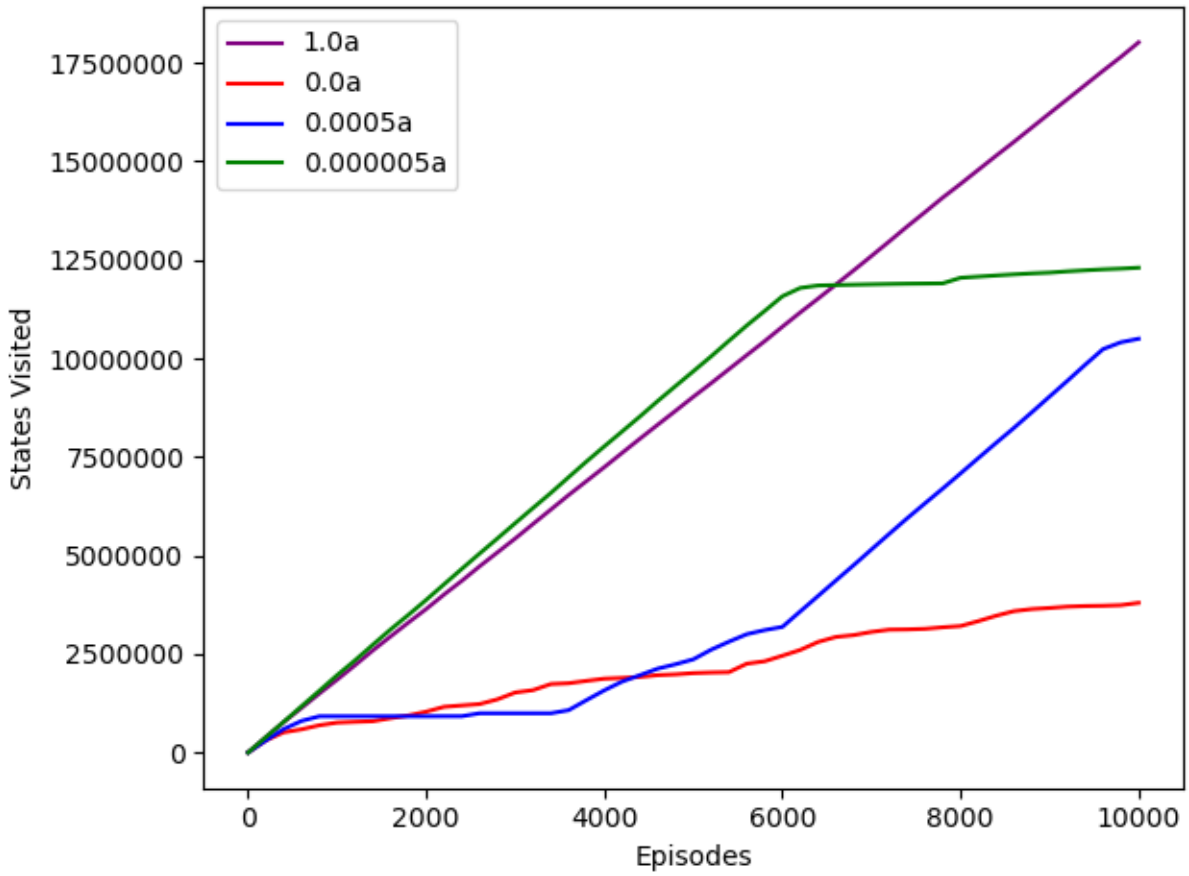

Figure 4.9: The amount of states visited with TDMCTS, n=100, with varying learning rates

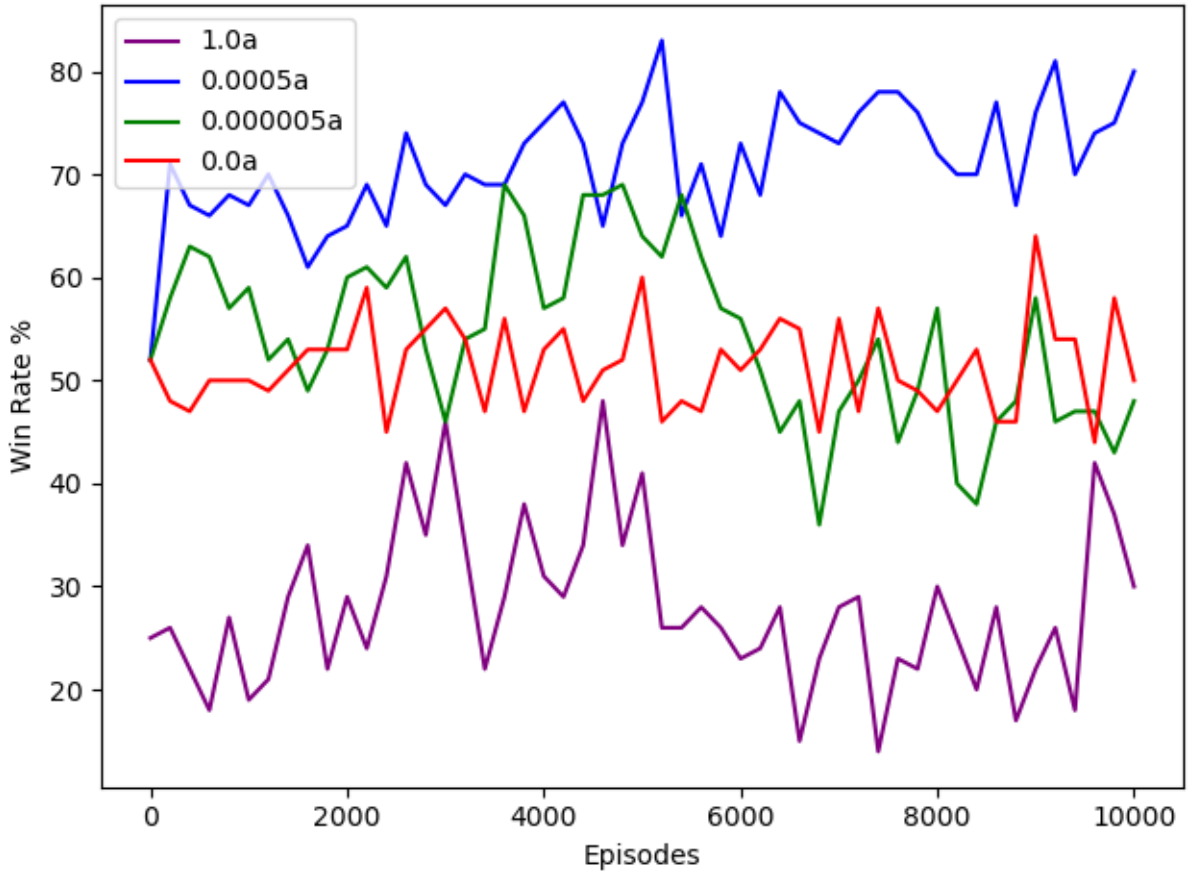

Figure 4.10: The winrate of TDMCTS, n=100, with varying learning rates

4.5 Evaluation

I personally believe that the algorithm developed here is quite an improvement over the regular MCTS algorithm. The only down-sides I can see are the tuning of hyper-paramaters. Although I was able to achieve a high of over 70% win rate against Monte Carlo tree Search as well as Minimax, I think further tuning of the hyper-paramaters can increase this win-rate even more. Finding the "sweet spots" for these hyper-paramaters was very tough and time-consuming, being off by one decimal place can change the algorithm drastically, going from a 20% to a 70% winrate based on what value was there

for the exploration, or the learning rate. Furthermore, it seemed that if the cap for the learning rate was too high, the value of each state would be very volatile, and therefore it would change drastically fairly often. This volality means that the value for the state fails to converge to a value. While a too low of a cap on the learning rate meant that it seemed to get pidgeon-holed into a set of moves that it thought were good, and once the learning rate gets too small, it ends up playing the same games over and over again, with the value for the state only changing very very slightly. This can lead to a waste of training games as it ends up playing over a thousand episodes of virtually the same moves, while a learning rate just a slight bit bigger would have caused it to skip those 1000 episodes entirely, and maybe get to that value after 200 episodes.

I found that lower exploration rates were definitely better overall, but it comes at a cost of exploring the state space. I've come to realise that the values of 0.5, 0.75, and 1.0 all have a similar amount of states visited, but I noticed that the lower the exploration rate, the more "good" states were visited; that is, the amount of times that states with a positive value associated with them were visited a lot more times the lower the exploration rate became, but too low of an exploration rate led to the algorithm being pidgeon-holed into chosing the same opening move every time, and so would nearly refuse to explore other possible opening moves. As a result, these really low exploration rates meant that it depended heavily on its first initial simulations, depending on which move it would deem the best. I think that though over-time, given enough episodes of training, it would eventually explore and find better moves, but was computationally unfeasible in the experiments that I ran.

One final point I would like to make about exploration rate is that as the algorithm learns, I think the exploration rate should decay slightly as it explores the state-space. This should lead it to explore highly at the start, then refine itself into exploiting more often than it explores by the end of its training session. This is evident in figure 4.4 where it tends to weight lower viewed states disproportionally to already exploited higher-value states, and so goes through a period where it explores these "bad" moves more, until it realises that they're still bad moves, and so gets back on track and its performance sky-rockets again.

Chapter 5

Conclusions

To conclude, I successfully implemented a Reinforcement Learning algorithm, backed by Temporal Difference Learning in order for an AI to learn the games of Connect4 and Tic-Tac-Toe, which is not taught in the undergraduate syllabus. In doing so, I assessed how the algorithm I developed performed against benchmark algorithms such as Minimax, Alphabeta, and regular Monte Carlo Tree Search (which is the basis for the recent development in super-human game AI as seen in AlphaGO). I also assessed key hyperparamaters, which overall affected how the algorithm performed, and did a detailed analysis on the results of the changing of these hyperparamaters. I also discussed possible extensions to the algorithm which could be made in the future to improve the performance of the algorithm.

Overall, Monte Carlo Tree Search is a very powerful algorithm. The generality of it is so appealing, and seems intuitively based on how humans themselves learn things. There has been countless times where I see myself, and others, using some form of this algorithm when they themselves play games, through the use of selecting a move that seems good, and then playing out max depth simulations in their heads to try and see which move will bring them to the best board position to win. The inclusion of temporal difference learning and backpropagation is also thought to be how neurons in the human brain learn good behaviours from bad behaviours. This intuitiveness, paired along with the concept of hyper-paramaters; little knobs and dials which affect the behaviour of the algorithm, is really what artificial intelligence is about. I absolutely loved the way tweaks could be made to each individual paramater, greatly affecting the behaviour of the algorithm, and then trying to get into the "mind" of the algorithm, trying to figure out its motivations

for each certain move, then making further tweaks to try and improve its behaviour.

It's clear that there is still a lot of work to be done, and I think the general introduction of a neural network would greatly increase both the efficacy, as well as the efficiency of the algorithm. We have seen high enough win rates against Minimax and regular Monte Carlo Tree Search in the games of Tic-Tac-Toe and Connect4, but I am confident that this algorithm can be extended to many other games as it was designed in a way with generality in mind, as seen with AlphaGO.

Bibliography

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O. & Zaremba, W. (2017), Hindsight experience replay, *in* I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan & R. Garnett, eds, 'Advances in Neural Information Processing Systems 30', Curran Associates, Inc., pp. 5048–5058.

Bostrom, N. (2014), *Superintelligence: Paths, Dangers, Strategies*, Oxford University Press.

Bouzy, B. (2007), 'Old-fashioned computer go vs monte-carlo go', *EEE Symposium on Computational Intelligence and Games* .

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. & Colton, S. (2012), 'A survey of monte carlo tree search methods', *IEEE Transactions on Computational Intelligence and AI in Games* **4**(1), 1–43.

Calhoun, L. (2017), 'Google artificial intelligence 'alphago zero' just pressed reset on how to learn'. accessed: 2020-1-05.
URL: *"https://www.inc.com/lisa-calhoun/google-artificial-intelligence-alpha-go-zero-just-pressed-reset-on-how-we-learn.html"*

Cormen, Leiserson, Rivest & Stein (2016), Introduction to algorithms (2nd ed.), MIT Press and McGraw-Hill, p. 344.

Coulom, R. (2006), Efficient selectivity and backup operators in monte-carlo tree search, Vol. 4630.

Eskin, E., Arnold, A., Prerau, M., Portnoy, L. & Stolfo, S. (2002), 'A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data', *Applications of Data Mining in Computer Security* **6**.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press.
URL: *http://www.deeplearningbook.org*

Gupta, A. (n.d.), 'Sarsa reinforcement learning'. accessed: 2020-01-06.
URL: *https://www.geeksforgeeks.org/sarsa-reinforcement-learning/*

James, S., Konidaris, G. & Rosman, B. (2017), An analysis of monte carlo tree search, *in* 'AAAI'.

K, A. V. (2019), 'Psychology in ai: How pavlov's dogs influenced reinforcement learning'. accessed: 2020-03-05.
URL: *https://medium.com/@anirudh246/psychology-in-ai-how-pavlovs-dogs-influenced-reinforcement-learning-58d69c0229fe*

Kocsis, L. & Szepesvári, C. (2006), Bandit based monte-carlo planning, *in* J. Fürnkranz, T. Scheffer & M. Spiliopoulou, eds, 'Machine Learning: ECML 2006', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 282–293.

LeCun, Y., Cortes, C. & Burges, C. J. (n.d.), 'The mnist database of handwritten digits'. accessed: 2019-09-05.
URL: *http://yann.lecun.com/exdb/mnist/*

Levinovitz, A. (2014), 'The mystery of go, the ancient game that computers still can't win'. accessed: 2020-02-05.
URL: *"https://www.wired.com/2014/05/the-world-of-computer-go/"*

Lijing & Gymrek (2010), 'Sp. 268 : The mathematics of toys and games'.
URL: *"http://web.mit.edu/sp.268/www/2010/connectFourSlides.pdf"*

Mihajlovic, I. (2019), 'Everything you ever wanted to know about computer vision'. accessed: 2019-10-23.
URL: *https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e*

Mitchell, T. M. (1997), Machine learning, McGra-Hill Science/Engineering/-Math, p. 2.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013), 'Playing atari with deep reinforcement learning'.

Nicholson, C. (2019), 'A beginner's guide to deep reinforcement learning'. accessed on: 2020-04-01.
URL: *https://pathmind.com/wiki/deep-reinforcement-learning*

Ruishan, L. & James, Z. (2018), 'The effects of memory replay in reinforcement learning', *56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)* .

Samuel, A. L. (1959), 'Some studies in machine learning using the game of checkers', **3**, 210–229.

Silver, D., Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T. & Lillicrap, T. (2019), 'Mastering atari, go, chess and shogi by planning with a learned model'.

Silver, Huang, Maddison, Guez, Sifre, Driessche, V. D., Schrittwieser, Antonoglou, Panneershelvam, lanctot, Dieleman, Grewe, Nham, Kalchbrenner, Sutskever, Lillicrap, Leach, Kavukcuoglu, Graepel & Hassabis (2016), 'Mastering the game of go with deep neural networks and tree search', *Nature* **529**, 484–489.

Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton, Chen, Lillicrap, Hui, Sifre, van den Driessche, Graepel & Hassabis (2018), 'Mastering the game of go without human knowledge', *Nature* **550**.
URL: *"https://doi.org/10.1038/nature24270"*

Sista (2016), 'Adversarial game playing using monte carlo tree search', *University of Cincinnati* p. 12.

Smith, M. L. & Kane, S. A. (1994), *The Law of Large Numbers and the Strength of Insurance*, Springer Netherlands, Dordrecht, pp. 1–27.

StanfordOnline (2019), 'Stanford cs234: Reinforcement learning — winter 2019 — lecture 2 - given a model of the world'. accessed: 2020-1-10.
URL: *https://www.youtube.com/watch?v=E3f2Camj0Is*

Sutton & Barto (2018), *Reinforcement Learning, An Introduction, Second Edition*, MIT Press.

Tesauro, G. (1989), 'Neurogammon wins computer olympiad', *Neural Computation* **1**(3), 321–323.

Tesauro, G. (1995), 'Temporal difference learning and td-gammon', *Commun. ACM* **38**(3), 58–68.

Vinayb21, Aradhya, A., Sing, S., jain, R. & 29AjayLumar (n.d.), 'Minimax algorithm in game theory (alpha-beta prunning)'. accessed: 2019-10-25.
URL: *https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/*

Violante, A. (2018), 'Simple reinforcement learning: Temporal difference learning'.
URL: *https://medium.com/@violante.andre/simple-reinforcement-learning-temporal-difference-learning-e883ea0d65b0*

Vodopivec, Samothrakis & Ster (2017), 'On monte carlo tree search and reinforcement learning', *Journal of Artificial Intelligence Research* **60**, 882–895.

Vodopivec, T. & Šter, B. (2014), 'Enhancing upper confidence bounds for trees with temporal difference values', pp. 1–8.

Wojciechowski, J. (2017), 'Speed up your game - playing ai with bitboards'.
URL: *https://spin.atomicobject.com/2017/07/08/game-playing-ai-bitboards/*

Wong, G. (1998), 'Solvability of backgammon'. accessed: 2019-10-15.
URL: *https://bkgm.com/rgb/rgb.cgi?view+550*