Reinforcement Learning and Monte Carlo Tree Search for Games

University of Limerick

DRAFT

Adam Swayne

March 26, 2020

Contents

Misc

Not sure where this goes, but I would like to keep the information for future reference

0.0.1 Search Space Complexity

Games such as Tic-Tac-Toe can be solved quite easily using a trivial brute-force search algorithm such as Minimax; but, Minimax begins to grind to a halt the more complex the game is.

The problem arises however when we move on to more complex games such as Connect4 or Chess. In Connect4, the search-space size for a regular six row by seven column grid results in 3^{42} 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; so we would need at least that many positions stored to do a brute-force search of the game (Lijing and Gymrek; 2010). Similar calculations can be done for Chess and Go, leading to a search space complexity of 10×10^{50} and 10×10^{170} respectively. This exponential explosion in complexity means that traditional brute-forcing algorithms means....

Chapter 1

Introduction - 5 pages

1.1 Overview - 2 pages

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).

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) as the search policy for the game Connect4. I am also hoping to generalise the temporal difference learning process through the use of a deep convoluted neural network.

1.2 Objectives - 1/2

1.2.1 Research Objective

The intent of my research is to quantify the efficacy of deep reinforcement learning and Monte Carlo Tree Search for the game Connect4. The reason I chose Connect4 was because the game is of reasonable complexity as opposed to much more complex games such as Chess or Go (see section 0.0.1), which both require a considerable amount more of computation power in order to solve effectively. For example, the MCTS algorithm in question 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).

As part of my research objective I will also attempt to assess the impact that hyper-paramaters have on the learning process, these hyper-paramaters include the learning rate (α), the discount factor (γ) and the exploration factor (ϵ). All these paramaters are usually in the range [0,1], and depending on their value; can seriously change the behaviour of a reinforcement learning agent. I like to think of them as little knobs that you can tweak to your choosing to effect how the agent learns and explores the state space.

1.3 Methodology - 1/2

1.4 Overview of the Report - 1/2

1.5 Motivation - 1/2

I am hoping to have multiple versions of this project, the first version should incorporate Monte Carlo Tree Search with Reinforcement Learning using Temporal Difference learning. The second version should then hopefully incorporate a deep neural network in order to generalise learning (i.e. Be able to infer from previously unseen states).

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.

Chapter 2

Background- 16 pages

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). 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). 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 Connect4).

The idea behind Reinforcement Learning is based on an idea from pyschology called classical conditioning, 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 and 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 and 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 (see section 0.0.1) 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 2.5.3)

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 and Barto; 2018).

An MDP consists of:

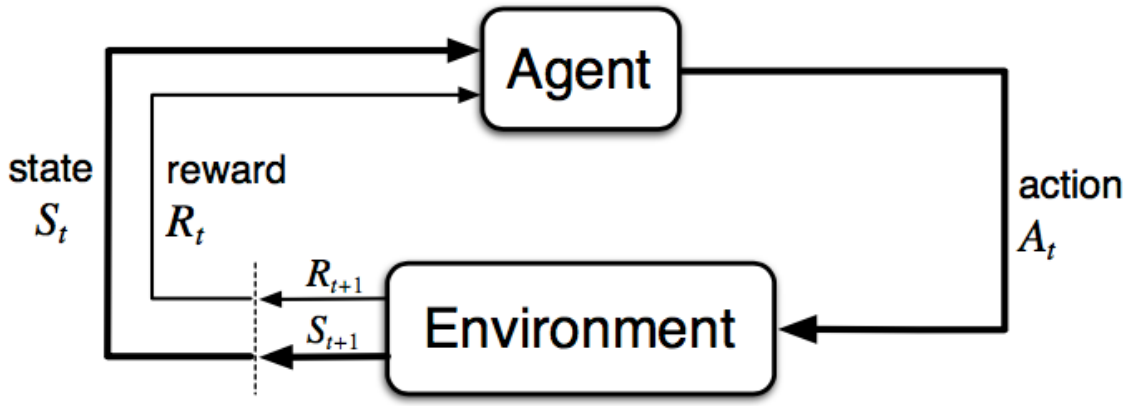

Figure 2.1: A typical model of an MDP, (Sutton and Barto; 2018)

· 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 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 - 1page

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 and Barto; 2018).* 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') + γ 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 and 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, 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 and Barto (2018), 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 and Barto (2018), 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 Reinforcement Learning - 1 page

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, Neu-

rogammon, which used supervised learning techniques to achieve a high level of play.

·**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)

(Bostrom; 2014)

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

2.5.1 History

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that was introduced in 2006 by Rémi Coulom. 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)

2.5.2 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)

2.5.3 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 and Barto; 2018). The algorithm for use in my MCTS prototype as described by (Kocsis et al.; 2006) is UCB Applied to Trees (A.K.A UCT). It is defined as follows:

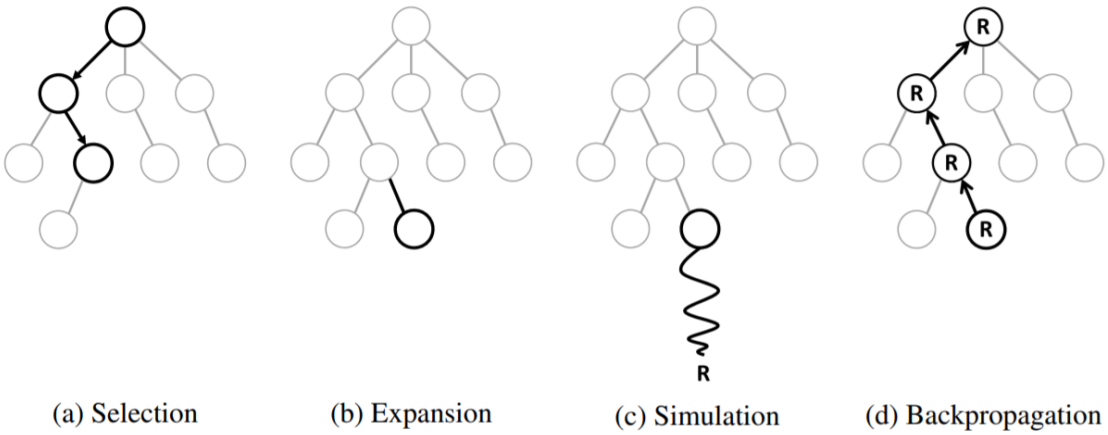

Figure 2.2: Simplified MCTS tree, (James et al.; 2017)

$\frac{w_i}{n_i} + \epsilon\sqrt{\frac{lnN_i}{n_i}}$, 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.

2.5.4 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.

2.5.5 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 and Monte Carlo Tree Search in Games - 1 page

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.

2.6.1 AlphaGO - 3-4 pages

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.

There were a few iterations of AlphaGO, the most recent being AlphaZero which was able to beat the original AlphaGO (100-0). The striking thing about AlphaZero 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)

2.6.2 TDGammoin

2.7 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 and Gymrek; 2010)

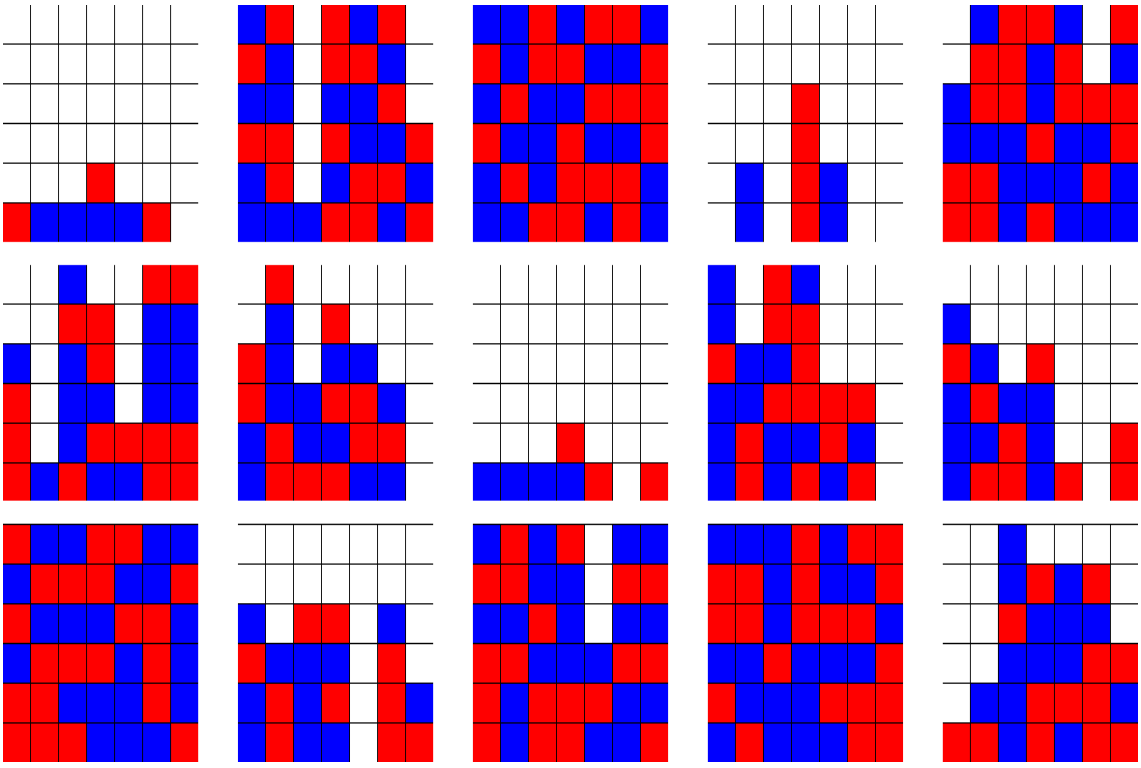

Figure 2.3: Some example games of Connect4

2.8 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.

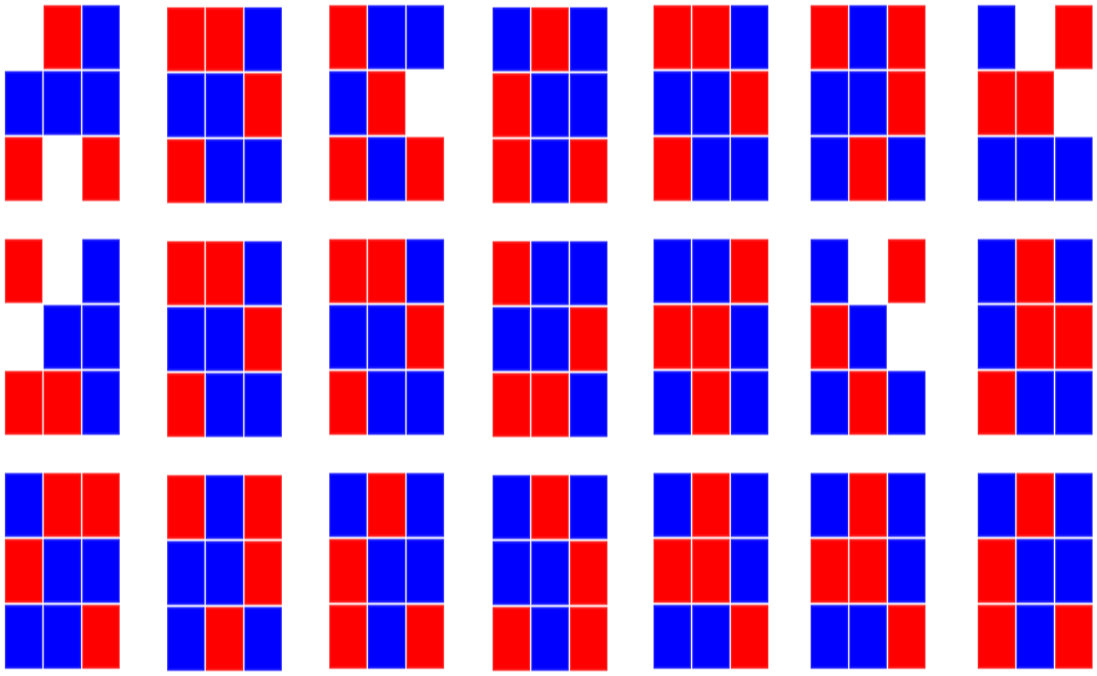

Figure 2.4: Some example games of Tic-Tac-Toe, (Note that X's and O's can be represented as colours instead)

Chapter 3

Prototype - 10 pages

3.1 Overview

In the following chapter I will outline my MCTS implementation, as well as a slightly modified version I call TDUCT. I will then outline the reason for using TDUCT 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.

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 3.1 and 3.2 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 3.2 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 can actually only require a 16bit number to represent the state space.

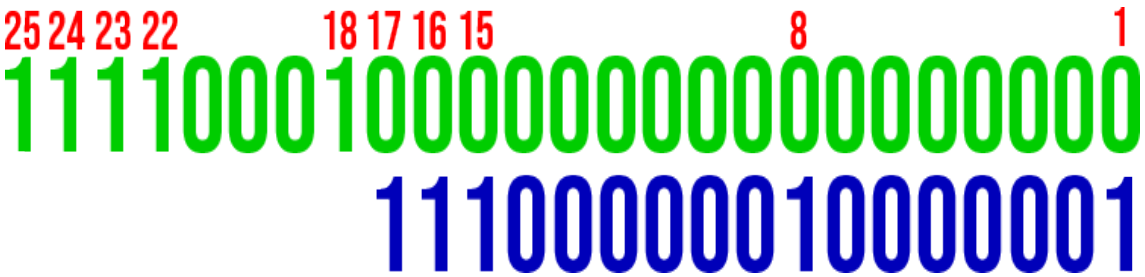

Figure 3.1: How each board is represented in memory, the red numbers represent the bit that is turned on. Note the correlation between the bits in Fig 3.2

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

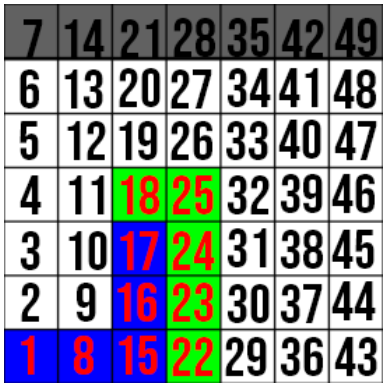

Figure 3.2: A visual Representation of a Bitboard, note the "extra" bits at the top

In our case, n is set to one, as we only need one hidden row.

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

```
def set_bit(self, player, bit, turn_on=True):
    if turn_on:
        self.boards[player] |= (1 << bit)
    else:
        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 3.2, 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.

```
    #Bitboard black magic
    def check_win(self):
        ##Vertical |, horizontal -, diagonal \\, diagonal /
        directions = [1, self.rows + 1, self.rows, self.rows + 2]

        #Only need to worry about the player who placed last
        player_to_check = self.get_player_turn(prev=True)

        board = self.boards[player_to_check]

        for direction in directions:
            m = board

            #Loop for however many discs we need in a row
            for i in range(self.win_span):

                #Shift the board i amount of columns/rows
                m = m & (board >> (direction * i))

            #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 3.3 and fig 3.4, 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 3.3 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 3.4, 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 TicTacToe.

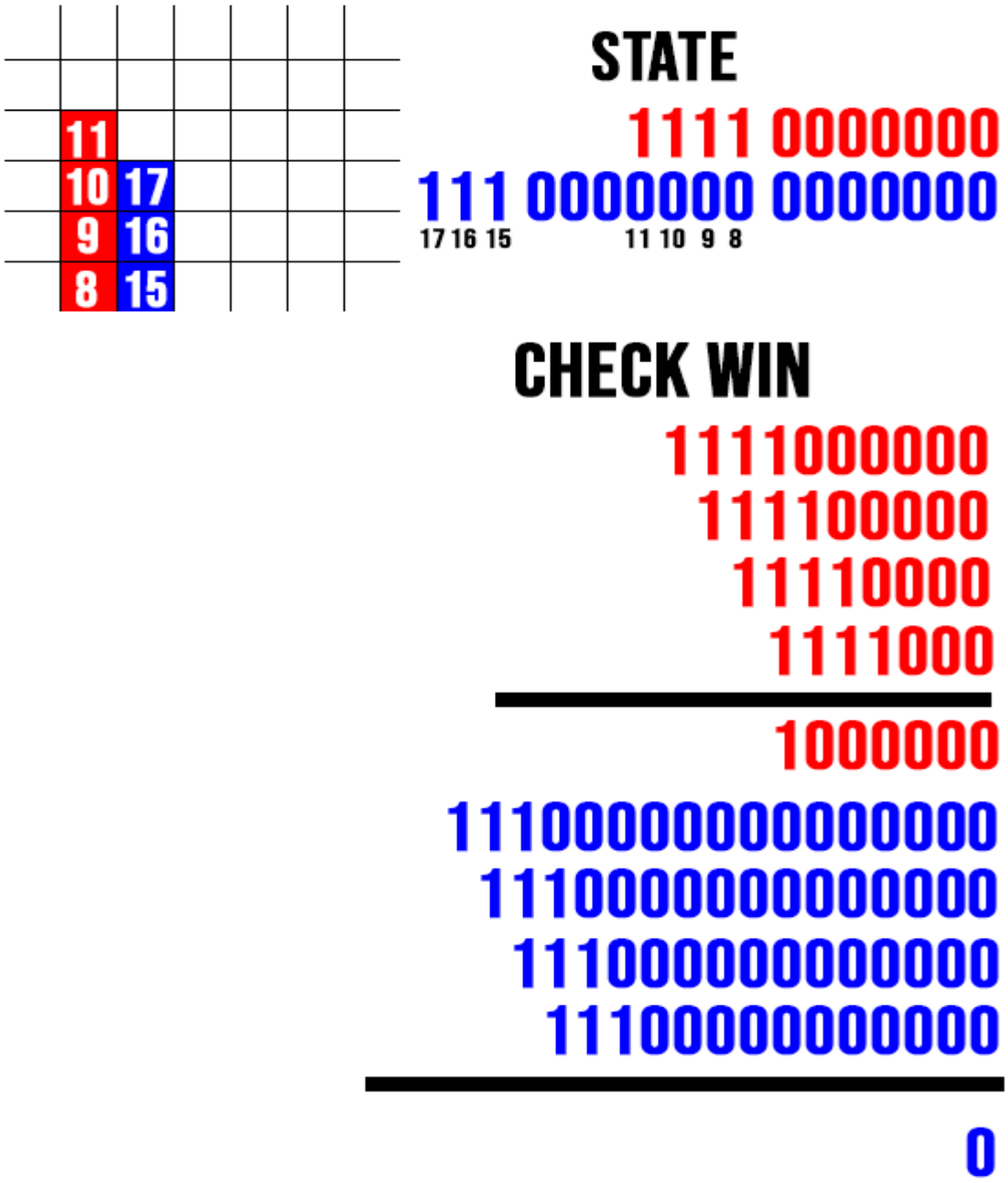

Figure 3.3: Bitwise checking of state win for a vertical four-in-a-row

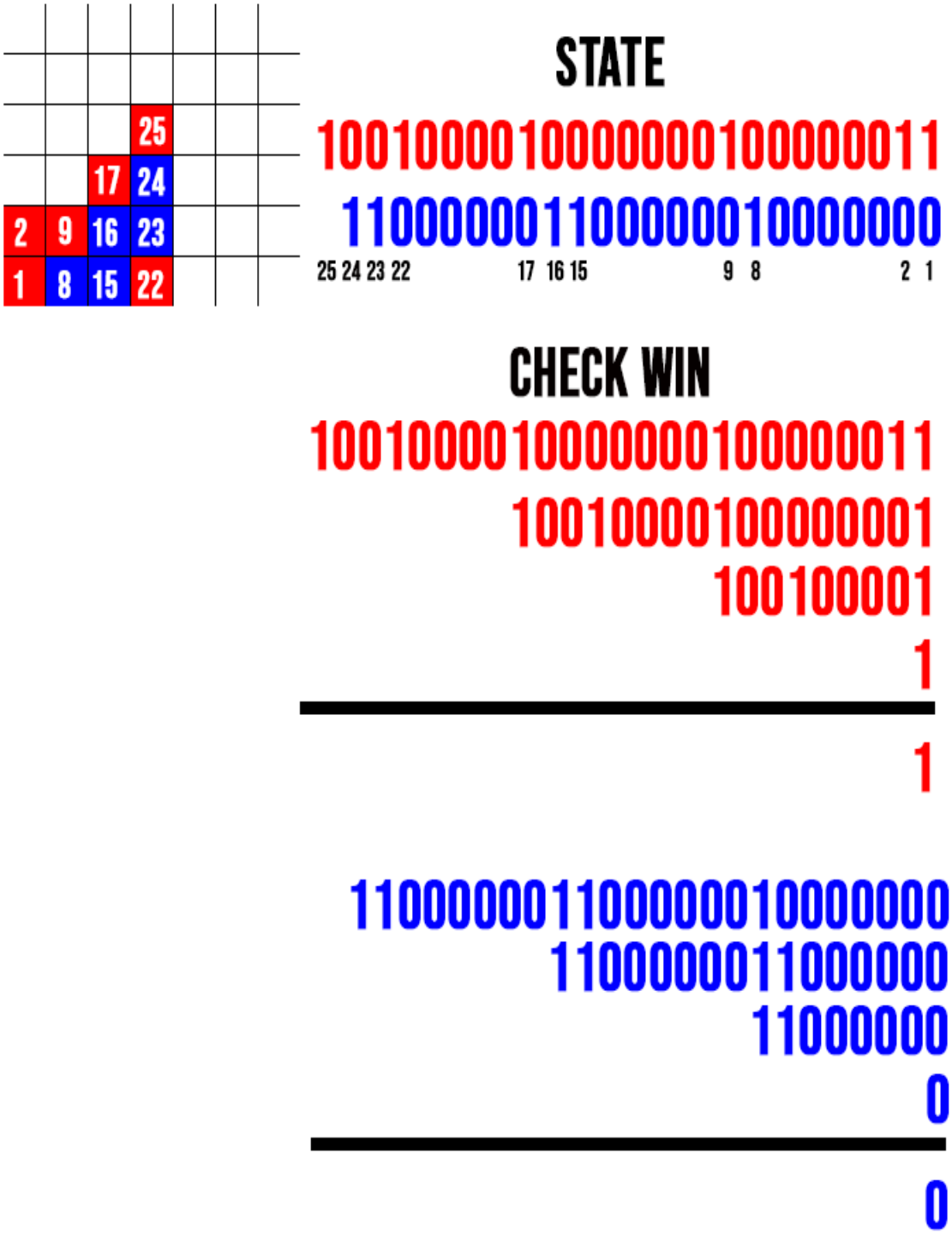

Figure 3.4: Bitwise checking of state win for a diagonal four-in-a-row

3.3 Monte Carlo Tree Search Implementation

3.4 Suggestions and Possible Improvements

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

3.4.1 Upgrades to BitBoards

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.

3.4.2 Memory Replay

Memory replay has seen success in the literature (Ruishan and James; 2017) (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.

3.4.3 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 - 15 pages

4.1 Training

I incorporated the use of Google Cloud Compute Engine in order to do all my training as the hardware that I have 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.

4.1.1 Training Paramaters

4.1.2 Problems Faced While Training

There were many different iterations to my TDUCT algorithm. The original was based off of (Vodopivec and Ster; 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, eventually I had to limit the learning rate of the backpropagation phase. Originally, the learning rate scaled based on the visit count of the state in question. So if a state had 5,000 visits, the next time its selected and a reward is observed, the value of that state is changed by $\frac{1}{5000}$ toward the new observed value. What I found out was that because of the nature of Monte Carlo Tree Search, whereby it samples the most promising moves at a time, the first thousand or so moves could leave the node with a high value, then another player could find a different path down the tree

from that state which leads to a loss. This leads to the state value being artificially inflated and because the weakness is found too late, the learning rate is now too low to make any significant impact on the node's value. As a result, I observed that in the training phase, a TDUCT agent would keep doing the same sequence of moves as it thought they were good moves to make, while its opponent would exploit these artificially inflated states and end up winning. This led to essentially the same game being played multiple times per training iteration. A solution to this was to let the learning rate decrease gradually until a set number. The number I came to was $\alpha = 0.005$ through trial and error with different learning rates.

4.2 State Space Search

As seen in TDGammon, 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 4.1, TDUCT 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, TDUCT (fig 4.2) sampled over 1,400,000 states.

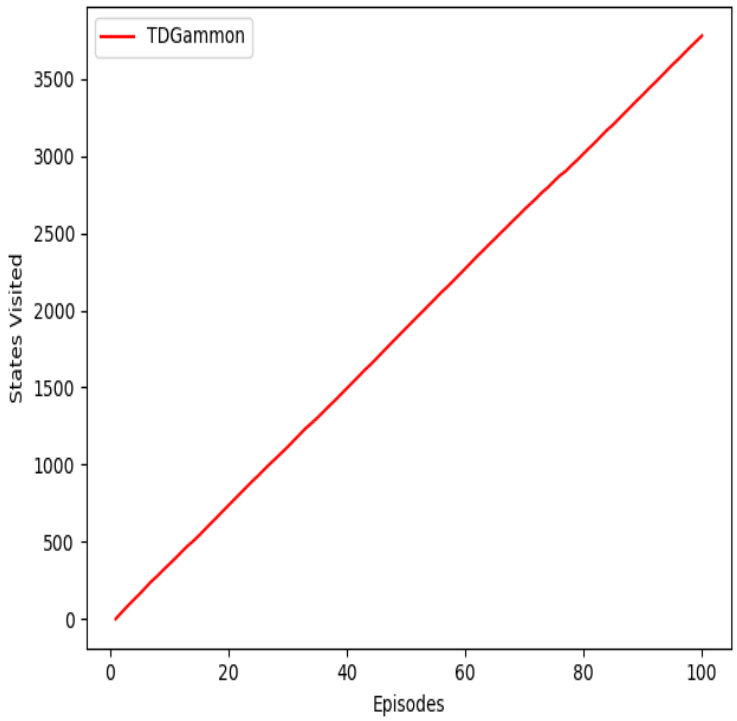

Figure 4.1: TDUCT learning from actual states that have occurred in training (As seen in Tesauro's approach)

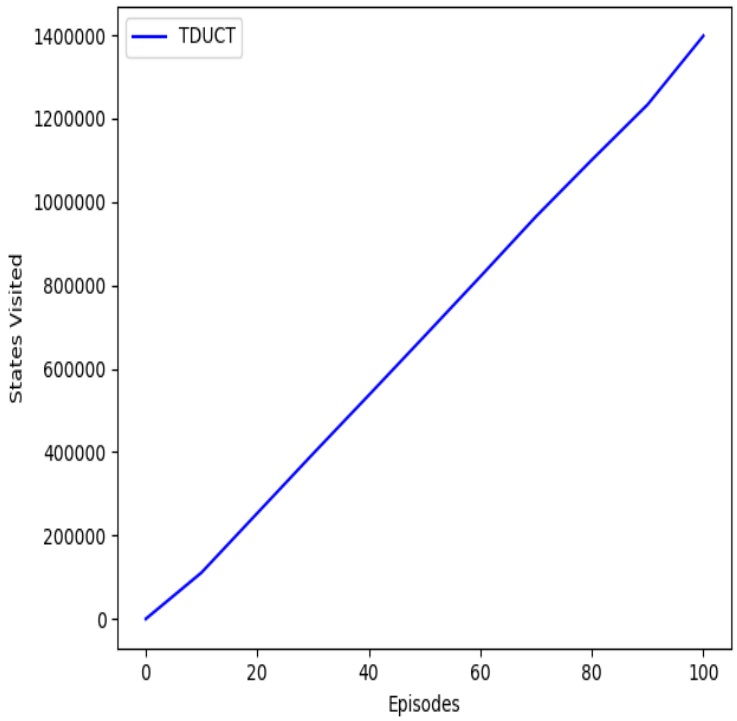

Figure 4.2: TDUCT, n=1000 learning from states found through the use of tree search

4.3 TDUCT vs Random Player

Originally I decided to play TDUCT vs a completely random opponent. What I found out was that with n=100, e=1 and l=0.9, TDUCT was able to achieve a 100% winrate with **no** learning whatsoever. This is a good indication that TDUCT 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 random player.

4.4 TDUCT vs Minimax

Fig 4.3 depicts TDUCT being pit against Minimax after each round of training. I observed that the Minimax heuristic weighted states the same way

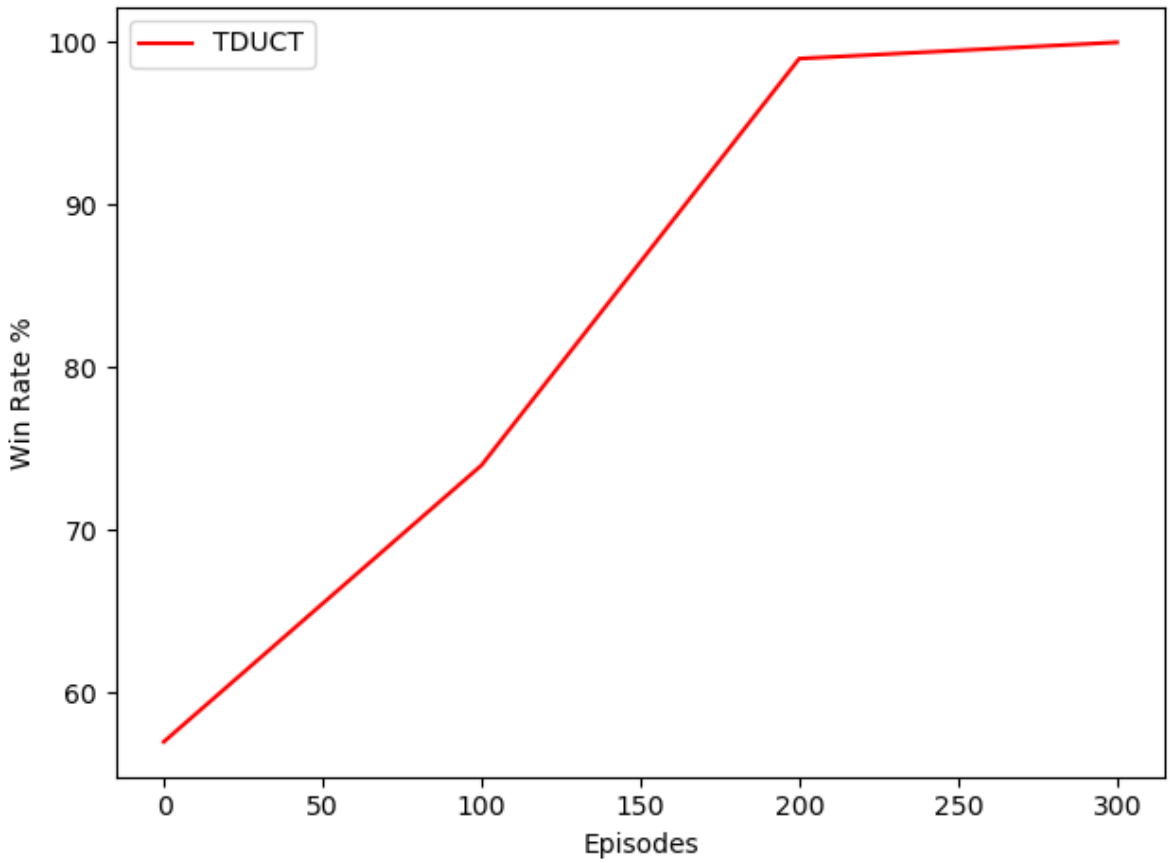

Figure 4.3: TDUCT, n=1000 exploiting the heuristic used for Minimax depth 4

every time (the heuristic used involved counting the amount of tokens in a row minus the opponents amount in a row), therefore TDUCT was able to learn a set of moves in which it could win every single time. As a result, TDUCT reached a 100% winrate after 300 episodes. As a result I had to revise my Minimax implementation and incorporate a newer improved heuristic.

4.5 TDUCT 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 TDUCT against this new Alpha beta algorithm in the game of TicTacToe; 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 4.5, 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 4.5 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%)

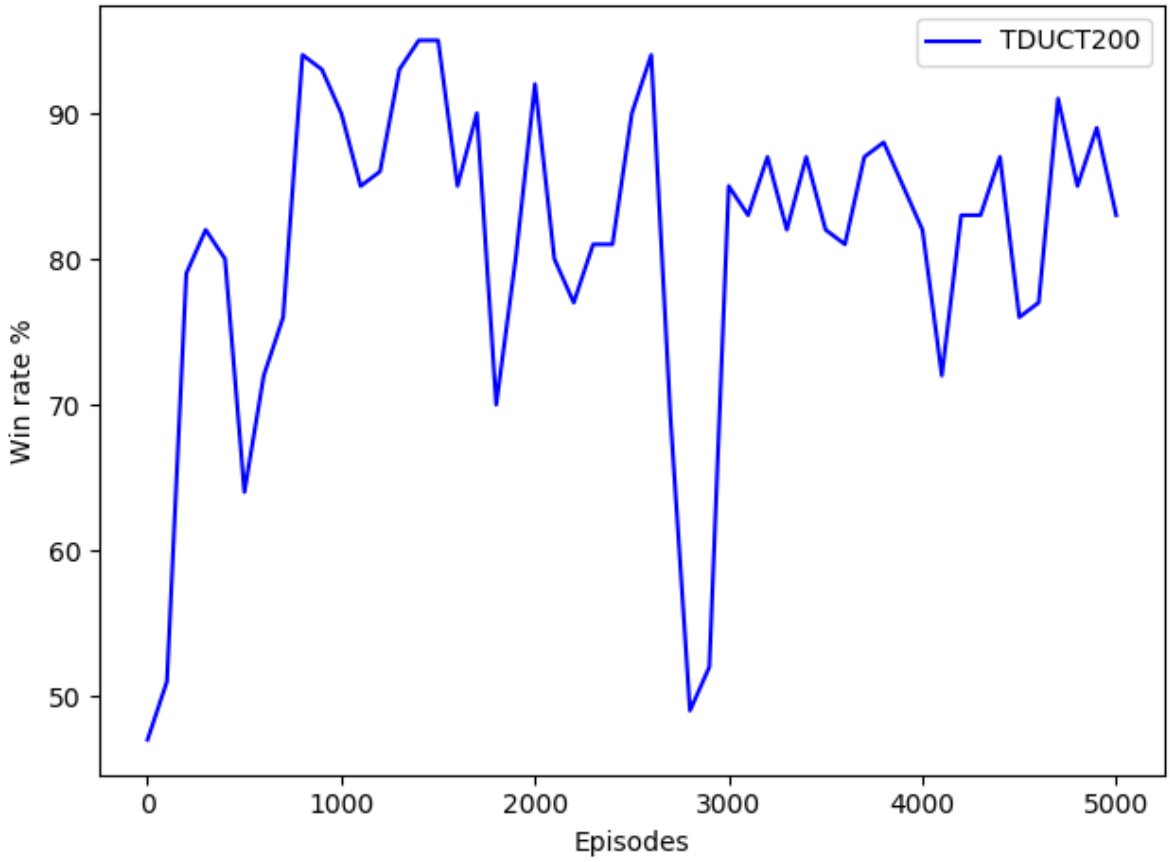

Figure 4.4: TDUCT, n=200, e = 1, l = 0.9 against Minimax depth 6, for the game TicTacToe

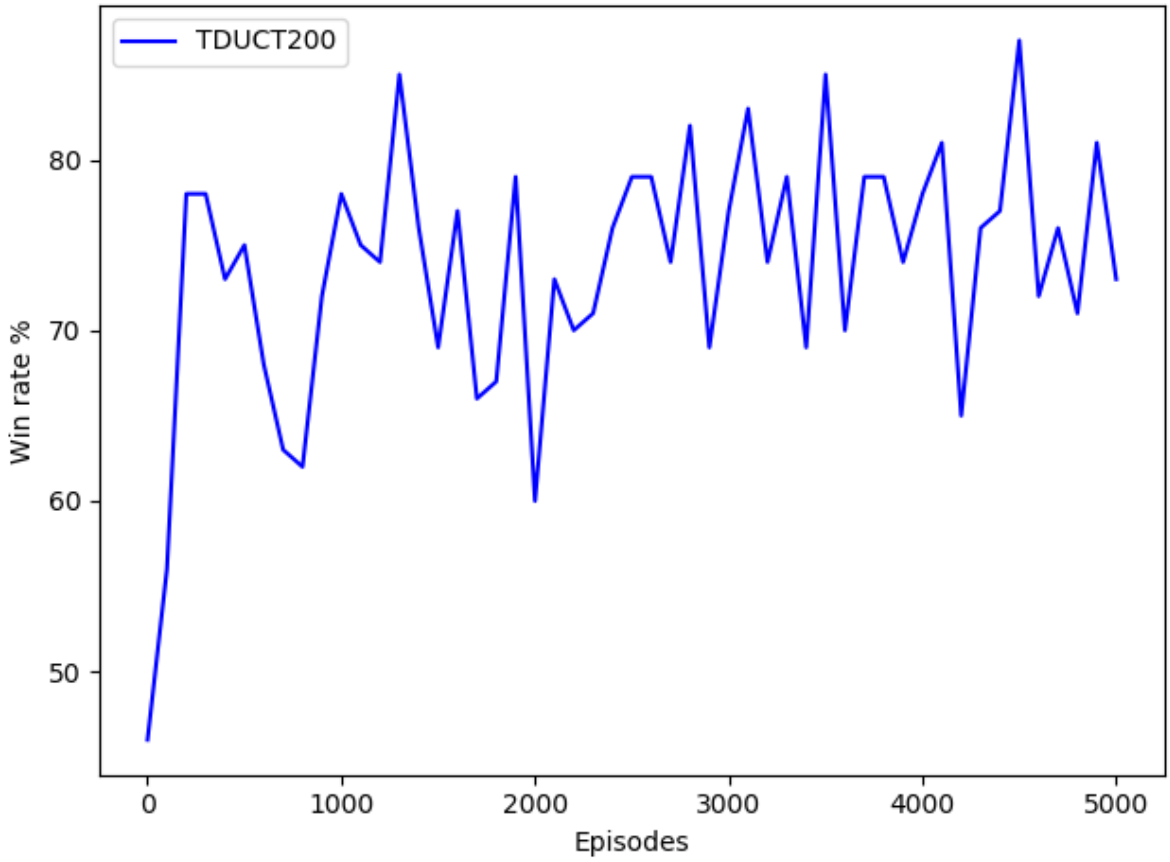

Figure 4.5: TDUCT, n=200, e = 0.5, l = 0.9 against Minimax depth 6, for the game TicTacToe

Chapter 5

Conclusions - 1 pages

Appendix A

Bibliography

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O. and Zaremba, W. (2017). Hindsight experience replay, *in* I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett (eds), *Advances in Neural Information Processing Systems 30*, Curran Associates, Inc., pp. 5048–5058.
URL: *http://papers.nips.cc/paper/7090-hindsight-experience-replay.pdf*

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

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S. (2012). A survey of monte carlo tree search methods, **4**.

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

Cormen, Leiserson, Rivest and Stein (2016). *Introduction to Algorithms (2nd ed.)*, MIT Press and McGraw-Hill.

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

Eskin, Arnold, Prerau, Portnoy and Stolfo (2002). A geometric framework for unsupervised anomaly detection, *Barbará D., Jajodia S. (eds) Applications of Data Mining in Computer Security. Advances in Information Security* **6**.

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

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

James, S., Konidaris, G. and Rosman, B. (2017). An analysis of monte carlo tree search, p. 2.

Kocsis, Levente, Szepesvári and Csaba (2006). Bandit based monte-carlo planning, pp. 282–293.

LeCun, Y., Cortes, C. and Burges, C. J. (n.d.). The mnist database of handwritten digits.
URL: *http://yann.lecun.com/exdb/mnist/*

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

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

Mihajlovic, I. (2019). Everytinhg 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.

Ruishan, L. and James, Z. (2017). The effects of memory replay in reinforcement learning.

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

Silver, Huang, Maddison, Guez, Sifre, Driessche, V. D., Schrittwieser, Antonoglou, Panneershelvam, lanctot, Dieleman, Grewe, Nham, Kalchbrenner, Sutskever, Lillicrap, Leach, Kavukcuoglu, Graepel and 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 and 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 and Kane (1994). The law of large numbers and the strength of insurance, **18**: 1.

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

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

Tesauro, G. (1995). Temporal difference learning and td-gammon, **38**(3).

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

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

Vodopivec, T. and Ster, B. (2014). Enhancing upper confidence bounds for trees with temporal difference values.

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