Learning to Play Leduc Hold'em Using Reinforcement Learning

UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

Department of CSIS

Bachelor of Science in Computer Systems

Author: Jamie Mac Manus

ID: 15147312

Supervisor: J.J Collins

Abstract

In recent years the area of machine learning has gained a lot of ground in a broad range of areas. A particularly interesting problem pertaining to machine learning is how we can develop useful AIs in a more hands off fashion. This problem is addressed by a machine learning paradigm named reinforcement learning. Reinforcement learning allows us to set up an agent in an environment after which the agent can explore the environment and begin to learn which actions that it should take in the different scenarios it can find itself in. This avenue of machine learning is suited to a broad range of problems but in this project we will explore the possibility of applying it to the game of texas hold'em.

Acknowledgements

First I would like to thank my FYP supervisor, Mr J.J Collins for his constant support, feedback and guidance on the project. I came to him with the broad aim of applying reinforcement learning to poker and through his expertise and engagement he was able to help me find a path for the project that tread the line of ambition and feasibility. Throughout the year J.J and I organised weekly meetings in which he provided invaluable feedback that led the project forward. I also greatly appreciate his patience as I worked to gain familiarity with a complex new area computer science.

I would also like to thank my family for their support, and to my friends with whom many fun times were had throughout my time in college.

I would like to thank Rory Egan, Dan Noonan and Kevin Moynihan with whom I collaborated on a number of projects during the four year course. Their effort, talent and insights made project work a painless task throughout the years.

I would like to thank Dr Jim Buckley for his work as FYP coordinator.

Finally I would like to thank the CSIS faculty and staff who have all contributed to a very positive four years of learning in this department. This is especially the case for all the lectures and TAs that I have had over the years who helped me develop a foundational CS knowledge that stood me in good stead for this project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Since the inception of machine learning, games have been a key problem area that has seen a lot of focus from top academics. For decades games have been used as a platform to test and develop algorithms that have gone on to provide invaluable services that are used in peoples everyday lives. The ability to contribute to this great history was a large motivator when it came to choosing this project.

Although this report will, to an extent, discuss machine learning and how it applies to games in general, the primary focus will be on machine learning techniques when applied to texas hold'em, or variations of it. In the past, methods such as Counterfactual Regret Minimization (CFR) have been used to develop agents that can play texas hold'em to a superhuman level. CFR is an algorithm that allows computation of a strategy through self-play. The metric used to update these strategies is called regret, which measures the difference between the game's outcome and the outcome that could have been achieved if some other action was taken. A large number of simulated games are played, with regret being calculated each time and subsequently being used to compute a strategy that is optimal. One example of CFR being used was the 2018 Annual Computer Poker Competition (ACPC) champion,

slumbot(Jackson 2013).

When applied to large imperfect information games such as texas hold'em CFR generally relies on the creation of abstractions of the game. The CFR algorithm is applied to these abstractions, generating a strategy. The generated strategy is then applied to the full version of the game and, if the original abstraction was accurate, our strategy will work well. This obviously requires a high degree of knowledge of the game in order to create an accurate abstraction.

There have also been attempts to tackle texas hold'em using a machine learning paradigm called reinforcement learning (RL). The great appeal of these approaches is that they may not require game abstractions or the associated domain knowledge.

RL is a way of programming agents by reward and punishment without needing to specify how the task is to be achieved(Kaelbling et al. 1996). RL problems consist of an agent in an environment. The environment consists of a number of states and rewards. The agent is allowed to take certain actions, in certain states. The overall goal is to learn a strategy that will maximise the cumulative reward. On the surface this seems like a perfectly reasonable methodology for solving games such as texas hold'em. However, texas hold'em is an imperfect information game. This means that we do not know the entirety of the state information at any given time i.e we do not know the values of the opponents cards. Thus, from a RL perspective, we do not know the actual state from which we are choosing actions. This makes pure reinforcement learning strategies impractical for these types of games.

However, there has been some success when more custom reinforcement learning methods have been implemented. In one case linear programming techniques and RL were used in order to tackle a simplified version of the game(Dahl 2001). RL was combined with techniques inspired by game theory(Heinrich et al. 2015). An RL related search algorithm called Monte Carlo Tree Search (MCTS) was also applied to a number of poker variations(Heinrich 2017). This latter approach will be the basis for our leduc

hold'em agent as we attempt to replicate and build upon the results outlined in this paper.

The different approaches will be discussed in greater detail in the background section.

1.2 Objectives

1.2.1 Primary Objectives

Although this project will be largely research based, the primary goal is to create a texas hold'em playing agent. Due to the fact that texas hold'em has an extremely large state space, combined with the fact that it is an imperfect information game, the initial goal will be to tackle a simplified version of the game. Specifically Leduc Hold'em will be used for this version of the project. This version of hold'em consists usually of a six card deck and only one private card, compared to two in texas hold'em.

In(Heinrich & Silver 2015) a metric called exploitability was used. This is a measure of how the agent's strategy fares against the best responses to that strategy. In other words, if the opponent knows our strategy, and can take the best possible action in every state in order to maximise their potential gain in reward, exploitability is the average amount that they would gain from doing so. For Leduc Hold'em, with a 6 card deck and 500,000,000 iterations the initial exploitability was 2 and converged to .0223. As such the success criteria for our initial iteration of the game is to replicate these results, with an allowance for hardware differences that may impact computational speed.

If the success criteria for this simplified version of the game are met, we will then proceed to tackle a more complex version of the game. In this second iteration of the project we will tackle limit texas hold'em with the end goal of recreating the results shown by Heinrich in his PhD thesis(Heinrich 2017). In this experiment, rather than using exploitability as an evaluation method, the agent was simply compared against a number of hold'em agents from the ACPC of the previous year. Thus win rate was used as the evaluation

metric. More specifically the measure used was mbb/h or milli big-blinds per hand. This is a measure of the number of big blinds won or lost per thousand hands. Note that the big blind is the larger of the two mandatory bets required at the beginning of each hand. Against the top five competitors in the ACPC, the fully trained agent achieved a win rate of between -25 and 28 mbb/h. This score is consistent with that of a player that is competitive with the best poker agents from that year's competition. As such, if the second iteration of our project is completed the goal will be to have a win rate of -100mbb/h or better against these same agents. Again we give an allowance for the difference in hardware used to train the agent as well our limited time.

It is also my goal to create a product that will be fun and useful for the general public. As such another objective will be to create an application that will allow users to play heads-up against the final product.

1.2.2 Secondary Objectives

As this project is very specific and academic, one of the larger challenges will be to gain a strong knowledge of the domain. This means learning the history of RL, the types of problems that it has been used to solve and the specific details of different RL algorithms.

A successful project will require a high degree of knowledge from the broader domain of RL. However, it is also the case that I must become closely familiar with the existing academic literature in the area of RL with respect to imperfect information games. This will allow me to avoid taking approaches that have previously shown to fail as well as allow me to add value to the existing literature whether that be through literature review or through my own experimental findings.

1.3 Contribution

1.4 Motivation

For the last number of years I have played poker recreationally with friends or online. It became more of an interest of mine as I started to explore the mathematical basis for the game and how players could use their knowledge, intelligence and temperament in order to gain an advantage in a game that, on the face of it, seemed to be largely based on chance. I spent some of my free time researching different aspects of the game. This included gaining some basic knowledge like the probability of making drawing hands as well as learning more technical aspects of the game such as how to calculate expected value of hands, or how to narrow down one's opponent's range of possible hands.

Concurrent with the development of this interest I was also becoming more and more interested in the area of machine learning. Machine learning and the development of artificial intelligence is possibly the most glamourized area of computer science. However, this is probably for good reason because there is something intrinsically interesting about machines that can learn to solve a problem on their own, without direct instructions from a human. The fact that machine learning has made so many strides in recent years was another cause of interest in this area of computer science, especially as the practical viability of ML as means of tackling a wide array of problems in industry continues to increase.

As a result the merging of these two interests as the basis for my final year project seemed like an obvious choice.

Chapter 2

Background

The aim of this chapter is to give the reader background information on certain areas of machine learning and game theory in order for them to understand the rest of the report. There will also be a in-depth discussion of existing literature that relates to machine learning in texas hold'em agents.

Machine learning is an area of computer science that tackles how we construct computer programs that improve with experience(Mitchell et al. 1997). The term was coined by Arthur Samuel in a paper in which he discussed machine learning methods using checkers(Samuel 1959). Since then there has been a great deal of advancement in the field. Some of the notable early contributions being the discovery of recurrent neural networks in 1982, the advancement of reinforcement learning by the introduction of Q-Learning in 1989 and the development of a backgammon-playing agent using neural networks and temporal difference learning(Tesauro 1995). Recently we have seen some of this early academic work culminate in more practical achievements such as Facebook's DeepFace system which, in 2014, was shown to be able to recognise faces at a rate of 97.35% accuracy, a rate that is comparable to that of humans. Another example of recent achievement is Google's AlphaGo program which, in 2016, became the first program to beat a professional human player.

It should be becoming clear that machine learning can be a solution to

a wide array of problems and as both hardware and software continue to improve it's reach will only continue to grow. We are starting to see machine learning systems become a key component of many companies business model. Since certain machine learning techniques are great at prediction, machine learning has been widely used for content discovery by companies such as Google and Pinterest. Other business applications include the use of chatbots as a part of customer service, self-driving cars and even in the field of medical diagnostics.

2.1 Reinforcement Learning

The early research for this project yielded reinforcement learning as the most suitable machine learning paradigm for the problem of texas hold'em. However, in order to understand both reinforcement learning and how it would apply to the chosen problem, in depth research was required. This research included a Udemy course(LazyProgrammerInc. 2018) as well as reading in part the book Reinforcement Learning: An Introduction(Sutton et al. 1998).

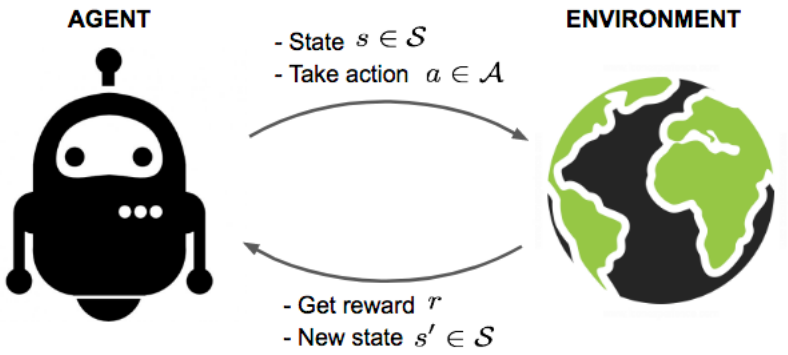

Figure 2.1: Reinforcement Learning

As mentioned in the introduction, reinforcement learning is a method of programming agents by reward and punishment without needing to specify how the task is to be achieved. As such the primary components of a reinforcement learning problem are an agent which exists in an environment.

From a simplified perspective we can think of the environment as a set of states, actions and rewards. The objective for the agent is to maximise cumulative reward. This is done by developing a policy that will dictate which actions should be taken in each state.

2.1.1 Explore-Exploit Dilemma

When it comes to reinforcement learning one of the first questions that we have to ask is how we explore the state space. An example that is often used to conceptualize this problem is the multi armed bandit problem. Let's say an agent is in a room with a number of gambling machines. Each of these machines has an arm that, when pulled will return a reward of 0 or 1 based on some underlying probability(Kaelbling et al. 1996). The agent has a limited number of total pulls. So the question becomes how do we distributed these pulls in order to maximise return? Well, first we have to ensure that we explore enough that we find the machine with the best reward probability and second, we must then exploit this machine to the best of our abilities.

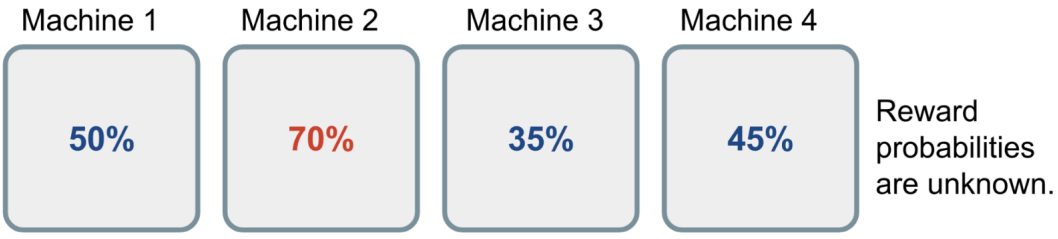

Figure 2.2: Multi Armed Bandit

There are a number of approaches that can be taken to solve this problem, we will now briefly discuss two of these methods.

ϵ-Greedy Solutions

The first approach that we will discuss is the ϵ-greedy strategy. This approach was first proposed in(Watkins 1989) and is a very simple and widely used method. The ϵ-greedy strategy involves choosing a random lever some

proportion ϵ of the time, and choosing the lever that has been established to give the highest reward the rest of the time.

There are a number of variations of this method, the first being the ϵ-first strategy. With this strategy we take all of our random choices first, allowing us to establish the best bandit, after which we exploit this bandit. However, as stated in(Vermorel & Mohri 2005) this simple approach is sub-optimal because asymptotically, the constant factor prevents the strategy from getting arbitrarily close to the optimal lever.

This is where the ϵ-decreasing strategy becomes useful. Here, the proportion of random lever pulls decreases with time. Generally if our initial epsilon value is ϵ_0 then our epsilon value at time t will be $\epsilon_t = \frac{\epsilon_0}{t}$.

Upper Confidence Bounds

Another approach that can be used is called the upper confidence bound(UCB) method. With this method we initially give an optimistic estimate of the reward to each bandit within a certain confidence interval. Then we simply take a greedy approach to our exploration. Less explored bandits will have a artificially higher reward estimate and thus they will be greedily chosen, thus allowing us to evaluate each of the bandits.

In the context of reinforcement learning, state space exploration through the ϵ-greedy approach is generally sufficient.

2.1.2 Markov Decision Processes

In the last section we have established some methods that can be used to explore environments. We will now discuss in more detail how reinforcement learning environments, and their interaction with reinforcement learning agents, are modelled. Generally finite Markov decision processes (finite MDPs) are used. Markov decision processes provide a formal mathematical framework for sequential decision making, where actions influence immediate rewards as well as subsequent situations(Sutton et al. 1998). MDPs allow us

to create an idealized model of reinforcement learning problems and thus we can make precise theoretical statements.

MDP Dynamics

As mentioned earlier, reinforcement learning problems consist of an agent and an environment interacting. Markov decision processes can be looked at in a similar way. However, there are a number of additional factors that we must consider in order to paint a complete picture.

We can think of the problem as consisting of a set of discrete time steps. At each time step the environment supplies the agent some information about the state, S_t. Using this information the agent chooses an action, A_t. Then, as a result of the action, the environment will supply the agent with a reward, R_t, as well as a new state. As such the process of interaction between the agent and the environment can be seen as a trajectory of states, actions and rewards(Sutton et al. 1998):

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2 \dots \tag{2.1}$$

Thus far we understand that states, actions and rewards are related, however questions still exist as to the exact workings of this relationship. The answer is that finite MDPs contain a discrete probability distribution that determines the likelihood that we will reach the state s' and receive reward r at time t based on the previous action a and state s:

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.2}$$

In simplified terms this means that for a given state-action pair (s, a), the probability of us reaching some new state s' and receiving reward r is determined by the MDPs probability function Pr.

This four argument probability function completely characterizes the dynamics of the MDP and from it anything else we want to know about the environment(Sutton et al. 1998).

MDPs and Learning

The goal of the agent in an MDP is to learn how to maximise the cumulative reward received when traversing the environment. In some cases we will traverse the MDP until we reach some terminal state, T. This type of MDP reflects episodic tasks that will always terminate. In this case we can calculate the cumulative reward,G_t as follows:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.3}$$

However, in other cases we will model continuous tasks. The problem here is that, if we use the same method of calculating G_t as we do for episodic tasks then in this case G_t will always eventually sum to infinity, regardless of whether we are taking good or bad actions. As such we must introduce the a new concept called discounting. With this approach the aim is to maximise the sum of future discounted rewards. Thus γ, a parameter with a value between 0 and 1, is introduced. As such, for continuous tasks modelled as MDPs our cumulative reward is as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \tag{2.4}$$

Based on our specified value of γ we can alter the weighting of future rewards. For example if we have a low value for γ (eg .5) then the value of rewards more than a few time steps in the future will be very low.

Partially Observable Markov Decision Processes

A partially observable Markov decision Process (POMDP) is used to model environments that are not fully observable(Kaelbling et al. 1996). POMDPs extend MDPs by including a set of observations, O. There is also an observation function, $P(O_t = o|S_t = s)$, that determines the probability of making a certain observation in a certain state. Hold'em can be modelled as a POMDP due to the fact that is an imperfect information game.

However, it is possible that an agent in a POMDP environment can remember the sequence of observations and actions that lead to the current

state. This is a sufficient statistic of it's experience and can thus define a complete information state(Heinrich 2017). As such we can reduce the POMDP to it's underlying MDP by using these full history information states and also extending the relevant transition and reward functions.

2.1.3 Policy Evaluation and Policy Improvement

As mentioned above the primary focus of reinforcement learning is to find a policy (denoted by π) that allows the agent to take actions in states that lead to the maximum possible reward. There are two primary problems that we must solve in order to do so.

The first is called the prediction problem, or policy evaluation. This involves computing the values of states given some arbitrary policy(Sutton et al. 1998). For example a state would have a high value if the reward for reaching that state was high. A state would also have a high value if we were only one action away, according to the supplied policy, from a state that renders a high reward. However a state would have a low value if, according to the policy, there was no path to a state that would return a positive reward in the foreseeable future.

The second problem is known as the control problem, or policy improvement. This involves changing the policy in order to improve our cumulative reward. The policy improvement process can only occur when the we have performed policy evaluation. Let's say, after our evaluation step, we know the value of some state s. Note that this value is calculated with the condition that we take some action a in state s. But, if we take some other action a' would this render a higher value for s? If the answer is yes then we update the policy.

These two operations can be seen as the core of reinforcement learning. In the next section we will discuss different reinforcement learning methodologies. Some of the main differences are in how method each approaches the prediction and control problems.

2.1.4 Dynamic Programming

Dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process(Sutton et al. 1998). DP is not widely used in practical reinforcement learning applications due to its assumption of a perfect MDP and it's high computational requirements. Despite this it is very important from a theoretical standpoint as it serves as an introduction to a number of important reinforcement learning concepts. Furthermore, it provides a basis for many algorithms that are used in practical reinforcement learning applications.

Policy Evaluation in Dynamic Programming

When discussion policy evaluation we talk about a state-value function or a value function. This is simply the mapping of states to their corresponding values and is denoted by v.

Since the environments's dynamics are completely known we can apply an iterative solution to finding the value function. If we consider a series of approximate value functions $v_0, v_1, v_2, ...$ The initial value function, v_0 is chosen arbitrarily and each successive generation is obtained by using the Bellman equation for v_π as an update rule(Sutton et al. 1998):

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \tag{2.5}$$

In order to produce each successive approximation of v_{k+1} from v_k we apply the operation outlined to each state s. As shown above our new value for s is based on a combination of the expected immediate rewards(R_{t+1}), and the expected values of each of the states that we can transition to(S_{t+1}) given policy π. It can be shown that as $k \to \infty$ v_k will converge to v_π, the correct value function for policy π. This algorithm is called *iterative policy evaluation*(Sutton et al. 1998).

Policy Improvement in Dynamic Programming

Since we have now determined how good it is to follow v_π we can use this information to determine how we should modify this policy in order to improve it's value. If we assume that π is a deterministic policy then $\pi(s)$ will return some action that we must take. Now the question becomes what if we take some other action $a \neq \pi(s)$? Well we must consider whether or not choosing this action, and then continuing to use the existing policy will improve the value of the policy. If it does then we will choose this new action.

The logical extension of this approach is to apply it to each state and each possible action. As such we will select what appears to be the best action at each state. We can thus denote our new greedy policy π' as:

$$\pi'(s) = argmax\, \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \tag{2.6}$$

Essentially here we are determining the value of each available action in the current state, using the same operation as outlined in the policy evaluation phase. Then the argmax function will select the action with the highest value. Finally we assign this action to be the one we will choose in state s, according to the new policy π'.

Note that in this section we have outlined an algorithm with respect to deterministic policies, however a lot of times in reinforcement learning we deal with stochastic policies. This means that we take actions in states according to some probability distribution rather than always choosing the same action in a particular state. This is not a problem as the ideas mentioned apply equally well to stochastic policies.

2.1.5 Monte Carlo

Monte Carlo(MC) methods are a wide range of algorithms that rely on random sampling in order to obtain results. We could apply such a method in the case of having an array of 1 trillion elements that could be either 1 or 0. If we wished to calculate the number of 1s in such an array through

iterating the array it would be very computationally costly. However, we could randomly generate indices and maintain a count of both the number of 1s found and the number of indices generated. Based on these figures it would be possible to estimate the total number of 1s in the array. Over time as we sampled more and more our estimate would converge towards the true value.

Monte Carlo based reinforcement learning techniques apply this method in the policy evaluation step. In Monte Carlo, unlike dynamic programming, we do not assume complete knowledge of the environment. Monte Carlo methods require only experience. We sample sequences of states, actions, and rewards from interaction with the environment(Sutton et al. 1998). These sequences are called episodes. Monte carlo evaluation is an episodic process this means that we only update our action values after an episode has completed.

Monte Carlo Policy Evaluation

In Monte Carlo methods we take a fundamentally different approach to policy evaluation. As mentioned this method is focused on using episodic experience. In order to evaluate a state we can simply average the rewards returned after visiting that state. As we observe more returns the average will converge to the actual expected value of the state.

It is worth noting that a state s could be visited more than once in an episode. As such we can either average the returns following the first visit to s or we could average the returns after each visit to s. These two methods are called *first-visit* and *every-visit* respectively.

Monte Carlo Action Values

In Monte Carlo methods, the lack of a model means that we cannot use only state values in order to obtain a policy. Rather state-action pairs are generally evaluated. This mapping of state-action pairs to values is referred to as the q function or the action value function. In this case the evaluation

problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π(Sutton et al. 1998).

The method for policy evaluation using state-action pairs is almost identical to that outlined above. The only difference being that instead of averaging rewards for each state, we average rewards for each action taken when a state is visited. There is one problem with this approach in the context of deterministic policies. The problem being that in following a deterministic policy we will only receive returns for a single action, thus only one action value estimate will be improved. In order to negate this problem we can specify that every episode begin in a state-action pair, with the probability of starting in each state-action pair being non-zero. This is called the *exploring-starts* method. Another approach would be to ensure that we are using a stochastic policy with the probability of selecting each action being non-zero.

Monte Carlo Policy Improvement

In Monte Carlo methods, the overall policy improvement algorithm is the same as outlined in the dynamic programming section. That is, we alternate between modifying the value function to more closely approximate the current policy, and using the value function to improve the policy.

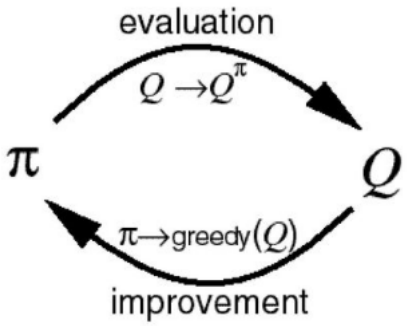

Figure 2.3: Monte Carlo Policy Improvement

2.1.6 Temporal Difference Learning

The final reinforcement learning method we will discuss is the Temporal Difference(TD) learning method. TD learning combines dynamic programming(DP) and and Monte Carlo(MC) ideas(Sutton et al. 1998). Like MC, we can learn directly from experience, without a model of the environment's dynamics. However, like DP we update state value estimates based on other learned estimates, without needing to wait for an episode to complete and the return of some final outcome.

The selective use of different aspects of these reinforcement learning methodologies by TD learning has a number of advantages. Obviously the fact that a model of the environment is not needed makes it easier to implement TD methods compared to DP . TD methods are also conducive to solving problems with long episodes or even continuous tasks with no episodes at all due to the fully online nature of this learning algorithm.

TD Learning Policy Evaluation

Unlike MC, with TD learning we need only wait until the next time step in order to update the value function. This is exemplified by the *TD(0)* or *one-step TD* method in which we make the update immediately on transition from state S_t to state S_{t+1}. The more general case of this method is the *TD(λ)* or *n-step TD*. With *TD(0)* the update rule is as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.7}$$

As such the new value for some state S_t is dependant on the previous value of the state($V(S_t)$), along with the reward(R_{t+1}) gained from transitioning to state S_{t+1} plus the discounted(γ) estimated value of that state($V(S_{t+1})$). The sum of the latter is multiplied by α which is a small positive fraction that influences the learning rate.

This rule is applied for each state visited in an episode and for each episode.

TD Learning Policy Improvement - SARSA

At this point it is worth noting that there are two distinct methods of handling policy improvement. The first is on-policy and the second is off-policy. On-policy reinforcement learning is when the policy being evaluated or improved is the same policy that is used to make decisions. In off-policy reinforcement learning the policy being used to generate behavior is not the same as the policy being evaluated or improved.

SARSA is an example of an on-policy algorithm. The policy improvement mechanism is the same here as outlined in the previous sections. With SARSA, like in Monte Carlo we utilise the action-value function($q_\pi(s, a)$) rather than the state-value function. Thus the policy evaluation step is slightly modified as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (2.8)$$

This update utilises the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, which is where the name SARSA originates.

TD Learning Policy Improvement - Q-Learning

Q-Learning is an off-policy control algorithm. This was one of the early breakthroughs in reinforcement learning as it allows the direct approximation of the optimal action value function independent of the policy being followed. The update rule is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max Q(S_{t+1}, a) - Q(S_t, A_t)]. \quad (2.9)$$

Here our update rule is similar to that of SARSA apart from the fact that we use the value of the best action available ($maxQ(S_{t+1}, a)$) when updating our action value.

2.1.7 Monte Carlo Tree Search

In this section we will discuss Monte Carlo Tree Search(MCTS), a decision time planning algorithm. This algorithm uses tree search combined with

value estimations from MC simulations in order to efficiently solve very large MDPs. One application of this algorithm was in DeepMind's AlphaGo(Silver et al. 2016) which became the first computer program to beat a highly ranked Go champion.

MCTS is an episodic process and there are four key steps involved in each of these episodes. The first step is **selection**. We begin at the root of the tree and use the tree policy, based on action values associated with the edges of the tree, in order to select a child node at each level(Sutton et al. 1998). The second step is **expansion**. When we reach a leaf node in the tree we may add one or more child nodes based on the actions available at that point. The third step is **simulation** When we have exited the tree we then switch to using the rollout policy (eg a random policy) in order to reach a terminal node in the tree from which we can obtain a reward. The final step is **backup**. This step is uses the reward obtained to update the values of the tree nodes that were visited during the episode.

In summary this algorithm uses the simulation step in order to efficiently sample rewards which are then propagated back to nodes of the tree. In later iterations and based on these values the tree policy selects which areas of the tree to explore and expand.

2.2 Game Theory

After taking a deep dive on reinforcement learning and the papers surrounding RL in texas hold'em it became clear that a pure reinforcement learning approach would not be feasible. The main reason for this was the fact that texas hold'em is an imperfect information game. As outlined by(Dahl 2001):

> Note that the concept of game state values, which is the key to solving perfect information games, does not apply to imperfect information games, because the players do not know from which game states they are choosing.

As such it became apparent that some other techniques would have to be incorporated in order to create a competent texas hold'em agent. After discovering(Heinrich & Silver 2016) and(Heinrich 2017) the initial goal was to utilise his Neural Fictitious Self-Play(NFSP) method as a basis for our agent. Eventually, it was decided that MCTS would be used as the basis for our agent implementation. NFSP is based heavily on the game theoretic concepts that are outlined in this section, however, MCTS also utilises a number of these concepts so it was decided that this research would be fully documented.

2.2.1 Modelling Games

When we attempt to solve a game, we must first understand how we are to model that game. We must also carefully consider how we define a form or structure of the model(Myerson 2013). In this section we will discuss three important forms that games can take and the varying utility of each in different scenarios.

Extensive Form Games

Extensive form games are a model of players sequential interaction, with explicit representation of a number of key aspects of the game. They were formally defined in(Kuhn 1953) as consisting of the following components:

- N - a set of players.
- S - a set of states that represent the nodes of a rooted game tree.
- $A(s)$ - a set of actions for each state representing the edges to the following states.
- U - a set of information states (one set for each player).
- *Player Function* - determines who is to act at a given state.

- *Information Function* - determines which states are indistinguishable for the player by mapping them to the same information state.
- *Return Function* - maps terminal states to each player's return/payoff.

Extensive form games allow us to richly describe game situations. This allows us to uncover characteristic differences between games and the structural features which determine these differences(Kuhn 1953).

We can also define behavioral strategies for players, which consist of a probability distribution over actions given information states(Heinrich 2017). This is denoted by $\pi^i(a|u^i)$. If we have a collection of strategies for all players in the game then this is called a strategy profile, π. π^{-i} is the set of strategies in π not including π^i. This will be discussed later when we discuss Heinrich's implementation in more depth.

Normal Form Games

Normal form games are represented by way of a matrix. Although some information is lost in comparison to extensive form games, normal forms are better suited to the derivation of generalized theorems(Kuhn 1953) and thus have their own utility.

An extensive-form game can induce an equivalent normal form of the same game. This can be done through the generation of a set of deterministic behavioral strategies for each player called pure strategies. Each pure strategy is a full game plan that will determine an action for each situation the player may encounter. We can also create mixed strategies which define a probability distribution over the players pure strategies. We can denote a mixed strategy for player i as Π^i. When we restrict the extensive-form return function to normal-form we yield an expected return function. The expected return function for some mixed strategy profile Π is $R^i(\Pi)$(Heinrich 2017).

Sequence Form Games

In order to compute a Nash equilibrium for an extensive form game we can convert the game to a normal form, however normal forms tend to produce very large game trees. The sequence form is an efficient method of representing extensive form games(Koller et al. 1996). This representation is described as a linear-sized strategic description of the game. It decomposes players strategies into sequences of actions and probabilities of realizing those sequences.

In sequence form games, for every player $i \in N$, each of their information states $u^i \in U^i$ uniquely define a **sequence** σ_{u^i} of actions that a player must take in order to reach that information state. These sequences are then mapped to realization probabilities through what is called a realization plan, denoted by x. When two or more strategies have the same realization plan we consider these strategies to be realization equivalent(Von Stengel 1996). This can apply across different types of strategies for example extensive-form, behavioral strategies and normal-form, mixed strategies(Kuhn 1953).

2.2.2 Nash Equilibria

A Nash's equilibrium is a state in which each player in a game has chosen a strategy and none of the players can benefit from changing their strategies, if the other player's strategies remain unchanged. As such, if we reach a strategy that induces a Nash's equilibrium our strategy can no longer be exploited.

In the context of extensive form games the concept of best responses is related to that of Nash's equilibria. If the opponent's strategies are denoted by π^{-i} then the set of best responses are denoted by $BR^i(\pi^{-i})$. Note that if we have a strategy profile π such that $\pi^i \in BR^i(\pi^{-i})$ for every $i \in N$ (i.e for every player) then that game constitutes a Nash equilibrium(Heinrich 2017). Heinrich also discusses the concept of ε-best responses. This is the set of strategies that are within a certain tolerance ε of the best responses. As

such we can define an ε-Nash equilibrium as a strategy profile π such that $\pi^i \in BR^i_\varepsilon(\pi^{-i})$ for all $i \in N$.

2.2.3 Fictitious Play

Fictitious play(FP) is a game-theoretic model of learning through self-play. At each iteration players choose the best responses to their opponents average strategies(Heinrich 2017). These strategies converge to Nash equilibria in certain classes of games, including two-player, zero sum games.

Generalised weakened fictitious self-play (GWFSP) is a method that is built on FP but allows for approximations in players strategies(Leslie & Collins 2006). Thus it is more suitable for machine learning. GWFSP allows for a certain error at each iteration of the algorithm and relies on the fact that this error rate will tend towards zero as time progresses. In the the research done by Leslie and Collins normal form games were studied. There has also been research done into the applicability of FP to extensive form games however, before(Heinrich & Silver 2016) there was no method shown to converge for imperfect information games such as poker. In later sections we will discuss how Heinrich utilised GWFSP as a basis for neural fictitious self-ply, a method that has shown success in games of imperfect information.

2.3 Supervised Learning

Supervised learning was also a common theme in the RL based texas hold'em implementations so once again a brief study of this area was conducted.

Supervised learning involves an agent which observes some example input-output pairs and learns a function that maps from input to output.(Russell & Norvig 2016). This learned function can then be used on new input data, that wasn't used to train the agent and the agent should be able to give an accurate output. The agent must be able to identify general features of the input data and how they map to the output. Common applications of supervised learning include computer vision, speech and pattern recognition

as well as spam detection.

2.4 Texas Hold'em

Texas hold'em is one variant in a family of games called poker. Poker is a group of card games that combine gambling, strategy and skill. All poker variants have three core similarities. There is betting involved, there is imperfect information (ie cards remain hidden until the end of a hand) and the winner is determined by combinations of cards.

2.4.1 Game Structure

Texas hold'em consists of four betting rounds. Initially each player is dealt two private cards. These remain face down and only the person who received these cards may view them. In the next three rounds five public cards are dealt face up on the table. The second round of dealing is called the flop, where three public cards are dealt. The third round is called the turn where one additional public card is dealt. Finally in the fourth round another public card is dealt which is called the river.

At each round, after the cards are dealt, the players are given the opportunity to take a number of betting related actions. We will discuss the permitted actions in the next section.

In order for players to be incentivized to continue playing in a wider array of situations, blinds are required. Blinds are a mandatory bet that must be posted by two of the players present at the game. These two bets are called the big blind and the small blind, the big blind being twice that of the small blind. As hands are played the big and small blinds are posted by different players in order to distribute the cost fairly.

The big and small blind are the first two bets that contribute to what's called the pot. The pot is the collection of all of the current chips bet by the players. When a player wins a hand then what they receive in return is the pot.

The final structural component of the game is player stacks. Each player will start the game with a certain amount of chips. If a player wins a pot then all of the chips in the pot are transferred to the winners stack.

2.4.2 Actions

As mentioned in the previous section, after cards are dealt players are permitted to take a number of actions. If a player is the first to act the they may either check or bet a chosen amount. If a player is not first to act and the previous player has made a bet then they may choose to either fold and forfeit the pot, call the bet by adding the same amount to the pot, or raise by adding the amount previously bet plus some additional chips. Players can go back and forth with bets until they run out of chips in which case they are considered to be "all in".

2.4.3 Hand Values

In poker the best 5 cards available to the player can be played. This means any combination of his own private cards and the public cards can be used. There are 10 major poker hands. These are listed below in ascending order of value:

1. **High Card:** None of the higher hand values achieved, highest card plays.
2. **Pair:** Any two cards of the same rank.
3. **Two Pair:** Two different pairs.
4. **Three of a kind:** Three cards of the same rank.
5. **Straight:** Five cards in a sequence.
6. **Flush:** Five cards of the same suit.
7. **Full House:** Three of a kind with a pair.

8. **Straight Flush:** Five cards in sequence, all of the same suit.

9. **Royal Flush:** A, K, Q, J, 10 - all of the same suit.

2.4.4 Leduc Hold'em

Leduc Hold'em is a simplified version of Hold'em that was first introduced in(Southey et al. 2012). In Leduc Hold'em the deck is reduced to six cards with two suits and three ranks in each suit. Rather than four rounds there are only two. In the first round a single private card is dealt to each player. In the second round a single board card is revealed. In the first round both players have a mandatory bet of one and a raise of two is allowed. In the second round players can raise by four. Each round allows for at most two bets.

2.5 Variations of MCTS applied to poker

As mentioned MCTS has been shown to be a very powerful method for solving large perfect-information games such as Go. However if we wish to apply this algorithm to imperfect information games like poker then a number of modifications must be applied. Such an approach was outlined in(Heinrich 2017) in which a variation of MCTS called smooth UCT was implemented to tackle both leduc hold'em and limit hold'em.

One of the subtleties of poker is that player information is asymmetric. In other words each player has access to their own private card information but not to their opponents private cards. This means that we cannot represent the search tree as a single, collective entity(Heinrich 2017), rather two search trees must be available to accommodate self-play. The method used by Heinrich to accomplish this goal was the creation of an information function $I^i(s)$. This function will return the information state u^i of the current player (i) given the current state s from the overall game tree. Note that the overall game tree will have separate nodes for any variation in either players

cards. However, player 1's game tree will not have separate nodes where the only differentiating factor is player 2's private cards(Johanson et al. 2011) due to the fact that this information is not available. This means that a number of nodes that are separate in the overall game tree will be grouped in either players individual game trees.

In figure 2.4 we have shown Heinrich's algorithm for extensive form MCTS .

In order to understand smooth UCT we must first explain the meaning of UCT . The abbreviation UCT refers to upper confidence bound(UCB) applied to trees. This means that in the action selection portion of the algorithm a UCB approach is taken, where less explored nodes in the tree are given a positive bias in value. This means that unexplored nodes will be explored which ensures that we discover the best actions at each position in the tree. The value of an action is denoted as follows:

$$Q(u^i, a) + c\sqrt{\log N(u^i)/N(u^i, a)} \tag{2.10}$$

In this expression the Q function denotes the current value estimates for each action a in information state u^i. $N(u^i)$ and $N(u^i, a)$ denote the current visitation count of the information state u^i and the subsequent information state after action a has been taken.

This extension of the extensive form MCTS algorithm is shown in figure 2.5.

Smooth UCT is a modification of UCT that is inspired by fictitious play(Heinrich 2017). As described in the last section the action selection in UCT is purely deterministic. Smooth UCT changes this by utilizing the average strategy a certain proportion of the time. In order to calculate the average strategy for a particular state we create a probability distribution based on the visitation counts of the actions available at that state. For example we could have an information state u^i that has been visited 100 times and has three available actions, a_1, a_2, a_3. Then it would be possible that a_1 has been visited 50 times, a_2 35 times and a_3 15 times. According to the

Algorithm 1 Extensive-Form MCTS

function SEARCH
  while within computational budget **do**
    Sample initial game state s_0
    SIMULATE(s_0)
  end while
  return π_{tree}
end function

function ROLLOUT(s)
  $a \sim \pi_{rollout}(s)$
  $s' \sim \mathscr{G}(s,a)$
  return SIMULATE(s')
end function

function SIMULATE(s)
  if ISTERMINAL(s) **then**
    return $r \sim \mathscr{R}(s)$
  end if
  $i =$ PLAYER(s)
  if OUT-OF-TREE(i) **then**
    return ROLLOUT(s)
  end if
  $u^i = I^i(s)$
  if $u^i \notin T^i$ **then**
    EXPANDTREE(T^i, u^i)
    $a \sim \pi_{rollout}(s)$
    OUT-OF-TREE(i) $\leftarrow$ **true**
  else
    $a =$ SELECT(u^i)
  end if
  $s' \sim \mathscr{G}(s,a)$
  $r \leftarrow$ SIMULATE(s')
  UPDATE(u^i, a, r^i)
  return r
end function

Figure 2.4: Extensive Form MCTS - Johannes Heinrich 2017 PhD Thesis

average strategy we would then select a_1 50% of the time, a_2 35% of the time and a_3 15% of the time. In order to determine when the average strategy should be used compared to the UCB approach Heinrich describes a sequence η_k. This sequence decays to 0 as $\lim_{k\to\infty}$ and is expressed as follows:

Algorithm 2 UCT

SEARCH(Γ), SIMULATE(s) and ROLLOUT(s) as in Algorithm 1

function SELECT(u^i)
  return $\arg\max_a Q(u^i,a) + c\sqrt{\frac{\log N(u^i)}{N(u^i,a)}}$
end function

function UPDATE(u^i, a, r^i)
  $N(u^i) \leftarrow N(u^i) + 1$
  $N(u^i,a) \leftarrow N(u^i,a) + 1$
  $Q(u^i,a) \leftarrow Q(u^i,a) + \frac{r^i - Q(u^i,a)}{N(u^i,a)}$
end function

Figure 2.5: UCT - Johannes Heinrich 2017 PhD Thesis

$$\eta_k = max(\gamma, (1 + d * \sqrt{N_k})^{-}1) \tag{2.11}$$

Here N_k is the total number of plays and γ is a lower limit on η_k and d is a constant that parameterises the rate of decay.

This final modification of the algorithm is shown in figure 2.6.

Algorithm 3 Smooth UCT

SEARCH(Γ), SIMULATE(s) and ROLLOUT(s) as in Algorithm 1
UPDATE(u^i, a, r^i) as in Algorithm 2

function SELECT(u^i)
  $\eta \leftarrow \max\left(\gamma, \eta_0 \left(1 + d\sqrt{N(u^i)}\right)^{-1}\right)$ as in Equation 3.1
  $z \sim U[0,1]$
  if $z < \eta$ **then**
    return $\arg\max_a Q(u^i,a) + c\sqrt{\frac{\log N(u^i)}{N(u^i,a)}}$
  else
    $\forall a \in A(u^i) : p(a) \leftarrow \frac{N(u^i,a)}{N(u^i)}$
    return $a \sim p$
  end if
end function

Figure 2.6: Smooth UCT - Johannes Heinrich 2017 PhD Thesis

2.6 Other Approaches

Although we have already mentioned that we will be focusing on MCTS it's worth discussing some other notable methods for creating poker agents before we continue. Currently the premier method for tackling full-scale texas hold'em is counterfactual regret minimization. This is the method that has dominated the Annual Computer Poker Competition(ACPC) for the last number of years, however recently some new methods have been outlined which look promising and thus we will discuss these methods as well.

2.6.1 Counterfactual Regret Minimization

Counterfactual regret minimization(CFR) is a method for finding approximate Nash equilibria in imperfect information games and was first outlined in(Zinkevich et al. 2008). Regret is a measure of the difference in utility between following some strategy σ compared to another strategy. Zinkevich introduced counterfactual regret which is regret applied to a single information set ie a single extensive form game state. It was found that by calculating and minimizing regret on an individual state basis that there were performance benefits as well as an improvement in accuracy of the calculated approximate Nash equilibria.

One implementation of CFR is outlined in(Jackson 2013). This paper outlines the implementation used for Slumbot, the 2018 ACPC champion in no-limit hold'em. Jackson describes his use of CFR in order to generate a static strategy that approximates a Nash's equilibrium. At each iteration a strategy would be computed and when the process had ended an average of these strategies would be used. Jackson also mentions using an abstraction of the game in order to reduce no-limit's massive state space (roughly 10^{164} with stack sizes of 100 big blinds)(Johanson 2013). This is done through techniques like grouping similar hand values into buckets and treating them as strategically equal or splitting the game up into it's individual rounds and solving the rounds separately. These techniques allow for a dramatic

reduction in state space and if handled correctly, should allow for a strategy that can transfer to the full version of the game successfully.

2.6.2 Neural Fictitious Self-Play

As mentioned earlier Fictitious Play(FP) is a game theoretic model that allows learning through self-play. This model was then extended a number of times until generalised weakened fictitious play(GWFP) was developed as a method of self-play that was applicable to machine learning techniques. Heinrich then utilised this method to develop Fictitious Self-Play(FSP)Heinrich et al. (2015). This is a machine learning framework that implements fictitious play. This method was shown to successfully generate approximate Nash's equilibria in imperfect information games such as Leduc Hold'em.

Neural Fictitious Self-Play(NFSP) develops upon these methods through the use of neural networks in order to facilitate the larger state spaces of games such as limit texas hold'em. This method was outlined inHeinrich & Silver (2016).

Algorithm

NSFP agents learn through self play ie interaction with other instances of the agent. As this process is happening the agents store experience of their play in two memories named M_{RL} and M_{SL}. These memories are subsequently used to for as data sources for the reinforcement learning and supervised classification portions of the algorithm respectively.

This algorithm trains two neural networks which are in-turn used to generate two strategies. The first neural network $Q(s, a|\theta^Q)$ is trained to predict action values from data in M_{RL} using off-policy reinforcement learning. The second neural network $\Pi(s, a|\theta^\Pi)$ is used to imitate it's own past best-response behavior using supervised classification on the data gathered in M_{SL}. The resultant strategies are the best-response strategy $\beta = \epsilon - greedy(Q)$, which uses the previously mentioned ϵ-greedy method for balancing exploration and exploitation, and the average strategy $\pi = \Pi$.

Algorithm 1 Neural Fictitious Self-Play (NFSP) with fitted Q-learning

Initialize game Γ and execute an agent via RUNAGENT for each player in the game
function RUNAGENT(Γ)
  Initialize replay memories $\mathcal{M}_{RL}$ (circular buffer) and $\mathcal{M}_{SL}$ (reservoir)
  Initialize average-policy network $\Pi(s, a \,|\, \theta^{\Pi})$ with random parameters θ^{Π}
  Initialize action-value network $Q(s, a \,|\, \theta^{Q})$ with random parameters θ^{Q}
  Initialize target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
  Initialize anticipatory parameter η
  for each episode **do**
    Set policy $\sigma \leftarrow \begin{cases} \epsilon\text{-greedy}\,(Q)\,, & \text{with probability } \eta \\ \Pi, & \text{with probability } 1-\eta \end{cases}$
    Observe initial information state s_1 and reward r_1
    for $t = 1, T$ **do**
      Sample action a_t from policy σ
      Execute action a_t in game and observe reward r_{t+1} and next information state s_{t+1}
      Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in reinforcement learning memory $\mathcal{M}_{RL}$
      if agent follows best response policy $\sigma = \epsilon$-greedy (Q) **then**
        Store behaviour tuple (s_t, a_t) in supervised learning memory $\mathcal{M}_{SL}$
      end if
      Update θ^{Π} with stochastic gradient descent on loss
        $\mathcal{L}(\theta^{\Pi}) = \mathbb{E}_{(s,a)\sim\mathcal{M}_{SL}}\left[-\log \Pi(s, a \,|\, \theta^{\Pi})\right]$
      Update θ^{Q} with stochastic gradient descent on loss
        $\mathcal{L}\left(\theta^{Q}\right) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{M}_{RL}}\left[\left(r + \max_{a'} Q(s', a' \,|\, \theta^{Q'}) - Q(s, a \,|\, \theta^{Q})\right)^2\right]$
      Periodically update target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
    end for
  end for
end function

Figure 2.7: Neural Fictitious Self-Play(Heinrich & Silver 2016)

As shown in figure 2.7 at each iteration the policy is selected first. Next an action a_t is sampled from that policy and by taking that action we generate a reward and subsequent state. This information is stored in M_{RL} and conditionally in M_{SL} based on the policy selected. After this stochastic gradient descent is used to update θ^{Π} and θ^{Q} who's new values are periodically applied to our neural networks. Over time the action value and average-policy networks will improve their accuracy as each player gains experience

and learns. This culminates in an approximate Nash equilibrium as players develop better and better strategies.

Convergence and Empirical Results

This algorithm was applied to both Leduc Hold'em and Limit Texas Hold'em with variation in a number of parameters such as number of hidden layers and neurons in the neural networks.

The best results achieved for Leduc Hold'em used a single hidden layer neural network with 128 hidden neurons. This instantiation of the algorithm developed a strategy reached exploitability levels of less that .1, with a starting value of 1.5.

The Limit Texas Hold'em instantiation of the algorithm was tested against the top 3 competitors of the 2014 ACPC using win rate as the evaluation metric in milli big-blinds per hand (mbb/h). This saw a trained win rate of between -13 and -52mbb/h against these agents with the initial win rate being roughly -700mbb/h.

Chapter 3

Implementation

In this chapter the details of the product developed will be discussed. This includes the implementation of the MCTS algorithm, our metric computation and the development of our prototype game for demonstration purposes.

3.1 UCT Algorithm

The core component of our implementation

3.1.1 Tree Representation

3.1.2 Selection

```
def select(self, history):
    player = util.player(history)
    player_history = util.information_function(history, player)

    if player in {-1, 1}:
        tree = get_tree(player)
        eta_sub_expression = math.pow(1 + .1 *
            math.sqrt(tree[player_history].visitation_count), -1)
```

```
8            eta = max((GAMMA, ETA_ZERO * eta_sub_expression))
9            z = random.uniform(0, 1)
10           if z < eta:
11               return self.get_best_action_ucb(history, player, tree)
12           else:
13               return self.get_best_action_avg_strategy(player_history,
                     tree)
14       else:
15           return random.choice(util.get_available_actions(history))

1    def get_best_action_avg_strategy(self, player_history, tree):
2        total_child_visits = 0
3        actions = []
4        probabilities = []
5
6        for child in tree[player_history].children:
7            total_child_visits += tree[child].visitation_count
8
9        for child in tree[player_history].children:
10           child_prob = tree[child].visitation_count / total_child_visits
11           actions.append(child.replace(player_history, ""))
12           probabilities.append(child_prob)
13
14       return rand.choice(actions, p=probabilities)

1    def get_best_action_ucb(self, history, player, tree):
2        player_history = util.information_function(history, player)
3        best_value = float('-inf')
4        best_action = None
5
6        for action in util.get_available_actions(history):
```

```
            node_val = self.calculate_next_node_value(tree, player_history,
                action, player)
            if node_val > best_value:
                best_action = action
                best_value = node_val

    return best_action
```

3.1.3 Expansion

```
def expand(tree, history, player):
    player_history = util.information_function(history, player)

    if player_history not in tree:
        tree[player_history] = potree.PoNode()

    for action in util.get_available_actions(player_history,
        player=player):
        new_history = player_history + action
        if new_history not in tree:
            tree[new_history] = potree.PoNode()
        tree[player_history].children.add(new_history)
```

3.1.4 Simulation

```
def simulate(self, history):
    if util.is_terminal(history):
        return self.handle_terminal_state(history)

    player = util.player(history)
    if self.out_of_tree[player]:
        return self.rollout(history)
```

```
8
9       player_history = util.information_function(history, player)
10      player_tree = get_tree(player)
11      if player_history in player_tree and
            player_tree[player_history].children:
12          action = self.select(history)
13      else:
14          expand(player_one_tree, history, 1)
15          expand(player_two_tree, history, -1)
16          action = random.choice(util.get_available_actions(history))
17      if player != 0:
18          self.out_of_tree[1] = True
19          self.out_of_tree[-1] = True
20
21      new_history = history + action
22      running_reward = evaluator.calculate_reward_full_info(history) + \
23                                  self.discount_factor *
                                        self.simulate(new_history)
24      update_player_tree(history, action, 1, running_reward)
25      update_player_tree(history, action, -1, running_reward)
26
27      return running_reward
```

```
1   def rollout(self, history):
2       action = random.choice(util.get_available_actions(history))
3       new_history = history + action
4       return self.simulate(new_history)
```

3.1.5 Tree Update

```
1   def update(tree, history, new_history, running_reward):
2       tree[history].visitation_count += 1
```

```
3       tree[new_history].visitation_count += 1
4       tree[new_history].value += (running_reward -
            tree[new_history].value) / tree[new_history].visitation_count
```

3.2 Best Response Computation

In order to evaluate the performance of our agent exploitability was utilised as the primary metric. Exploitability is a measure of how well our agent would fare against an opponent responding optimally to our strategy. Exploitability is related to the concept of Nash equilibria in that a Nash equilibrium is induced by a strategy that cannot be exploited.

To calculate the exploitability of a strategy the best responses to that strategy must be determined. The first step in calculating the best response strategy involves taking our agent's action selections and inserting them into the game tree(Heinrich 2017). In other words, wherever the agent must take an action in the game tree, the best action is chosen based on our MCTS estimations. As such the resultant tree will consist of the MCTS player's decision nodes which will have a single child node along with the second player's decision nodes and chance nodes which will both have multiple children. Terminal states can then be evaluated followed by propagation of values back up the tree, with the highest child value being propagated when a decision node for player two is reached. The average child value is propagated for chance nodes. This process will now be explained in detail with coding examples.

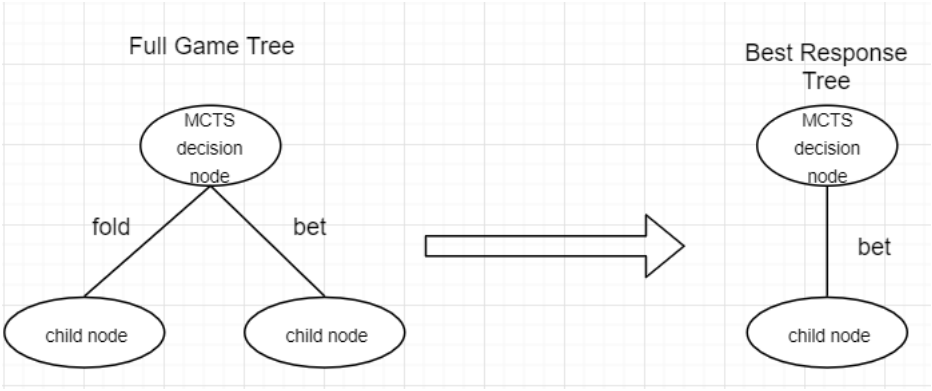

Figure 3.1: Generation of Best Response Tree

3.2.1 Generating Best Response Tree

3.2.2 Propagating Terminal Values

3.2.3 Exploitability

3.3 Prototype Application

In order to give visual evidence of the work done and the agent developed, a user interface(UI) was created that allowed human interaction with the trained bot. In order to facilitate the development of this interface in a short period of time the Qt framework was used. Qt is a open-source widget toolkit for creating graphical user interfaces and applications. Specifically Qt Designer was used in order to generate a UI template. Qt Designer is a what-you-see-is-what-you-get (WYSIWYG) UI generation tool that accompanies Qt. PyQt5 was then used to generate python code from this UI template and connect the functional component of the game. In this section the process involved in creating this prototype application will be outlined and some key code snippets will be highlighted.

In order to elicit requirements for this application I applied brainstorming as well as analysing a number of online UIs that served a similar purpose to mine. This process rendered the following functional requirements:

- Display list of available actions.
- Allow user to take action.
- Display relevant cards to player.
- Annotate the sequence of events that occur in the game.
- Display the current pot size.
- Allow player to play multiple rounds.
- Keep track of cumulative winnings across multiple rounds.

3.3.1 UI Screen

The first step towards achieving these functional requirements was to generate a UI screen. This was achieved through Qt designer using drag and drop with the end result being as shown below. Due to the fact that a number of images had to be displayed for the available cards in the game a Qt resource file was also created to store and locate those images.

Qt designer generates a .ui file that stores the UI template as well as a .qrc resource file. The following commands were used to convert these two files into python format in order for them to be compatible with the rest of the program:

```
$ pyuic4 -x ui.ui -o screen.py
$ pyrcc4 -o cards_resource.py cards/cards_resource.qrc
```

In order to create a visible screen the following code was used:

```
1 def setup_screen(self):
2     self.main_window = QMainWindow()
3     self.ui_screen = screen.Ui_MainWindow()
4     self.ui_screen.setupUi(self.main_window)
5     self.main_window.show()
6     sys.exit(self.application.exec_())
```

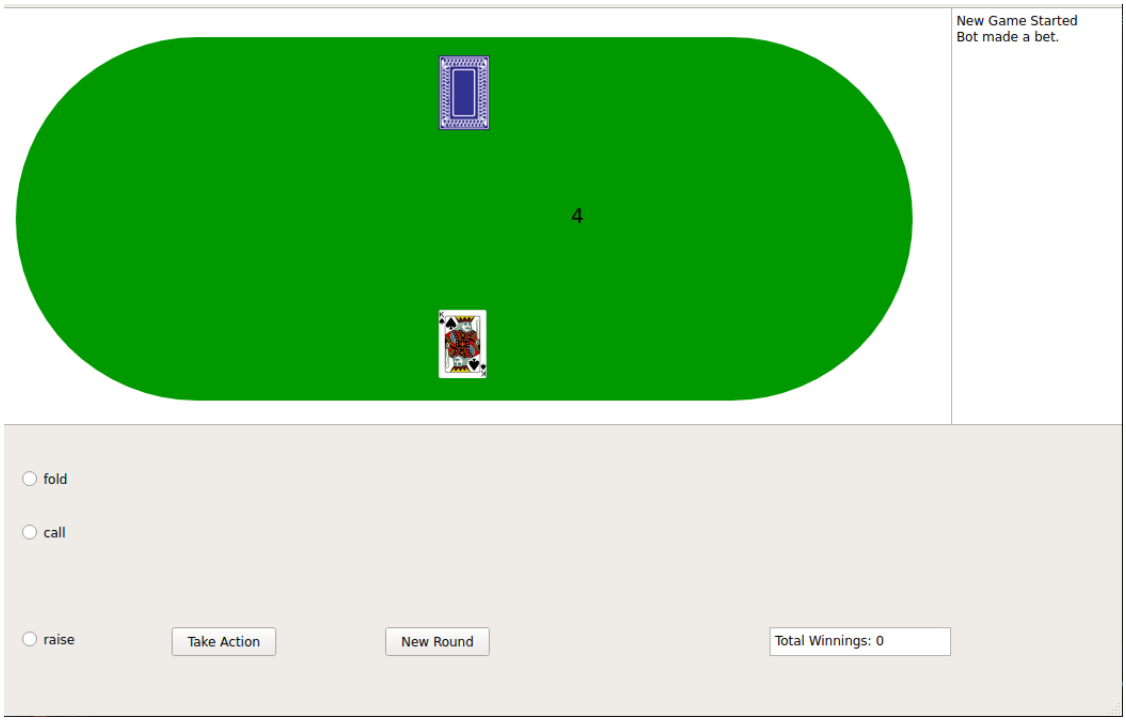

Figure 3.2: Game UI

3.3.2 Event Handling and UI Manipulation

When the UI screen had been implemented it was then time to attach the screen to the functional aspect of the game. This was done through the creation of a python class called Controller that extends the QWidget class from the PyQt framework.

In order to handle events callback methods were registered with the buttons that required their events to be handled. The code below demonstrated this process:

```
1 def setup_event_handlers(self):
2     self.ui_screen.take_action_button.clicked.connect(self.take_action)
3     self.ui_screen.new_game_button.clicked.connect(self.new_game)
```

Along with event handling the Controller class also had the responsibility of updating the UI based on the information that it gained from the game

model, which will be discussed in the next section. This primarily involved calling the setText() method of a number of the screen's widgets.

3.3.3 Game Model

The third component of our game was the game model. This was instantiated through a python class named Model that had the responsibility of feeding the correct data to the Controller class at any point in the game. Due to the fact there was already significant functionality built around the concept of histories, it was decided that the state of the game would be tracked through the use of histories. From the history the majority of the information about the game could be deciphered. This included the available actions, the cards in play and the pot. In order to fully satisfy our functional requirements the Model class also kept track of annotation messages that were generated based on game events, along with the cumulative winnings of the player over time. This data was returned to the Controller class in the form of a python dictionary.

The most significant method in this class was the update_game_state method. This method handled the player taking actions, the agent responding to those actions and the state being updated. Below the code for this method is listed.

```
1 def update_game_state(self, action, player):
2     self.history += action
3     self.display_text += PLAYER_NAMES[player] +
          ACTION_MESSAGES[action]
4
5     # Handle the game being over
6     if util.is_terminal(self.history):
7         winner = evaluator.get_winner(self.history)
8         winnings = -evaluator.calculate_reward_full_info(self.history)
9         self.total_winnings += winnings
```

```
10            self.display_text += "Game over. " +
                  PLAYER_NAMES[winner] + "won: " + str(abs(winnings))
11
12        # If the next player is the bot, allow it to take its action
13        elif util.player(self.history) == 1:
14            self.update_game_state(self.agent.get_action(self.history), 1)
15
16        # If the we need cards to be dealt, deal the cards.
17        elif util.player(self.history) == 0:
18            self.pub_card =
                  random.choice(util.get_available_cards(self.history))
19            self.update_game_state(self.pub_card, 0)
```

3.3.4 Agent Representation

In the case of this prototype game the agent is merely an instantiation of a pre-defined strategy. As such when we call agent.get_action(history) in the previous code block we are either stochastically or deterministically selecting an action based the aforementioned strategy.

Chapter 4

Empirical Studies

4.1 Overview

In this chapter we will cover a number of experiments that were conducted in order to investigate the performance of the different algorithms implemented in order to tackle Leduc Hold'em. We will begin with a simplified version of MCTS for POMDPs and will incrementally add to this method in order to see how the performance of our agent evolves. In the table below we have listed the template that we will follow when conduction these experiments.

Section	**Rationale**
Objective	This section will contain an explanation of the purpose of the experiment along with how it was carried out
Algorithm and Coding	This section will go into the details of the algorithm used to produce results
Results	This section will detail the results acquired from the experiments conducted
Analysis	In this section we will examine our results and try to provide insight into the reasoning behind these results

4.2 Experiment 1 - UCT Versus Random Player

The first experiment conducted involved a simplified version of the algorithm outlined in(Heinrich 2017). We set an initial goal of using a random player as our benchmark opponent in order to demonstrate how this algorithm could exploit such a player's strategic inefficiencies.

4.2.1 Objective

The goal of this experiment is to implement MCTS for Leduc Hold'em. The MCTS agent will play against a random player and learn a strategy to exploit this player for maximal reward. Although we are interested in the results gained from playing against the random player we treat the outcome of this experiment as a baseline for our subsequent results. The reasoning for this is that the difficulty in finding a winning strategy against a random player is not high. In fact, at each point in the game that the random player can take an action it is just as likely to fold it's cards, regardless of their value, as it is to take any other action. This means that for our agent the intelligent strategy is to merely retain all but the very worst of its hands. Thus we can think of the strategy learned by this initial agent as a broad categorisation between very weak hands and all other hands. Thus we expect that the exploitability of the resultant agent will be relatively high. Our agent will not have learned all of the strategic intricacies of the game and it will not react strategically to the opponent's play. Rather, it will simply know how to beat a 'dumb' random player. This will give us a platform to build a more sophisticated agent through different mechanisms such as self-play in the subsequent experiments.

4.2.2 Algorithm and Coding

As mentioned we will be following Heinrich's implementation. The pseudocode for this algorithm can be seen in figures 2.4 and 2.5. This algorithm ticks the box of being applicable to POMDPs like poker. In our case we

are learning values associated with histories of actions and observations that occur in the game. However it is worth noting that during this first iteration of the algorithm the game tree for only one player had been implemented and thus did not have to deal with information asymmetry.

4.2.3 Results

The first metric that was used in order to analyse the results of this experiment is cumulative reward. This is simply the sum of the output of the reward function over time. This function directly corresponds to the size of the pot won or lost in the game, thus the reward can be either positive or negative. In figure 4.1 we see the reward over time increasing. In order to obtain these results the algorithm was run for 10,000 iterations and this process was repeated 100 times. Our cumulative rewards were then averaged at each iteration across these 100 repetitions to give the graph shown. Figure 4.2 shows the rate of increase of cumulative reward, or the average reward over time. In the case of figure 4.2 we applied the MCTS algorithm for 500,000 iterations and repeated this process 20 times, averaging the results.

The second metric used to produce results was exploitability. This is a measure of the reward that can be gained by playing a best response strategy against the agent. In figure 4.3 we see that the exploitability of our agent is quite high throughout, with an initial dip followed by a divergence towards 4.9.

4.2.4 Analysis

We will first analyse our results from the cumulative reward metric. As shown by figure 4.1 we see that initially there is a gradual increase in cumulative reward, with the slope growing until there is a constant rate of increase. This demonstrates that during the initial phase of the algorithm we have not yet uncovered the most beneficial action selection for all states and the exploration phase is still in effect. However, by visual inspection we can see

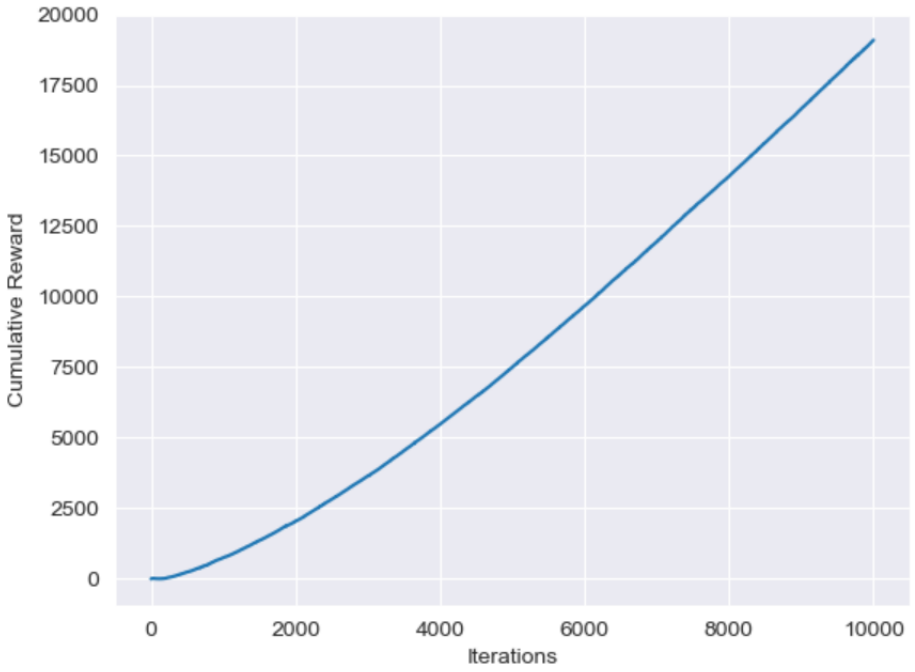

Figure 4.1: MCTS cumulative reward over time vs random player - 10000 Iterations

that over time the rate of increase in cumulative reward begins to stabilise which indicates that a concrete strategy that can exploit the random play has been established. This hypothesis is further supported by figure 4.2 as we see a dramatic upswing in the rate of increase of cumulative reward followed by a levelling of the graph.

As mentioned the exploitability value throughout this graph is relatively poor with an initial dip followed by a divergence. This is most likely explained by the fact that we are not playing against a rational player. As such our strategy is strong when it comes to maximally exploiting an irrational, random player but if we then substitute this irrational player for a rational player, the results will not be favourable.

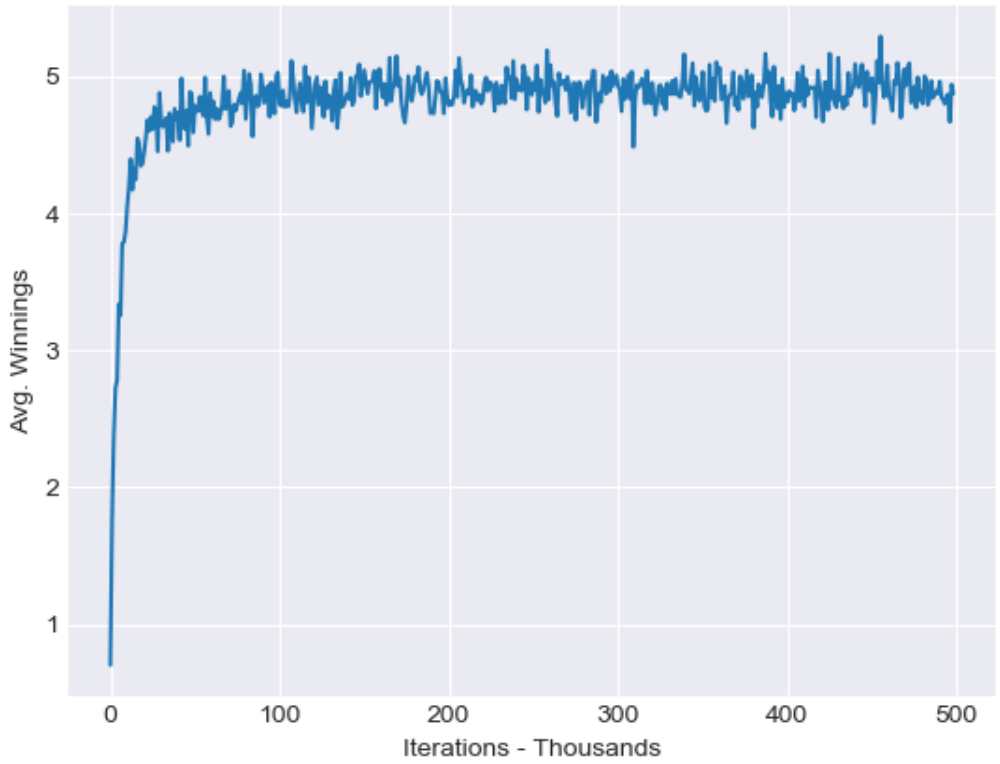

Figure 4.2: MCTS slope average reward over time vs random player - 500000 Iterations

4.3 Experiment 2 - UCT Self-Play

In our second experiment the agent was trained against itself. In other words, two instances of the agent were generated and allowed to develop strategies through self-play.

4.3.1 Objective

The objective of this experiment was to implement Heinrich's extensive form MCTS using UCB action selection (ie extensive form UCT). Here a significant improvement in the exploitability of the agent was expected. This was due to the fact that the opposing agent is rational, in that it learns through

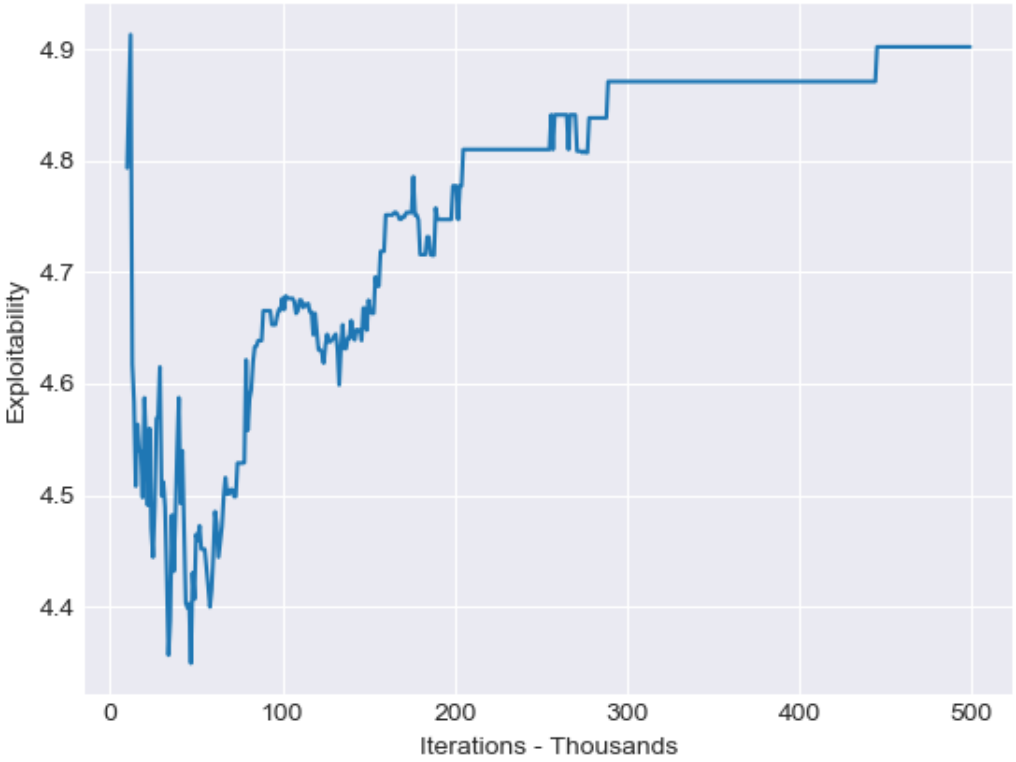

Figure 4.3: MCTS exploitability over time vs random player - 500000 Iterations

experience.

4.3.2 Algorithm and Coding

4.3.3 Results

In the case of self-play both agents develop intelligent strategies. This means that we do not see any significant trends in cumulative reward or average reward over time due to the fact that neither player has an advantage. As such these metrics were disregarded and exploitability became the sole metric. Once again we applied the algorithm for 500,000 iterations, repeating this process 10 times and averaging the results.

In figure 4.4 it can be seen that there is a notable improvement in exploitability, with lows of 3.5 initially but over time the values begin to diverge once again back towards 4.6.

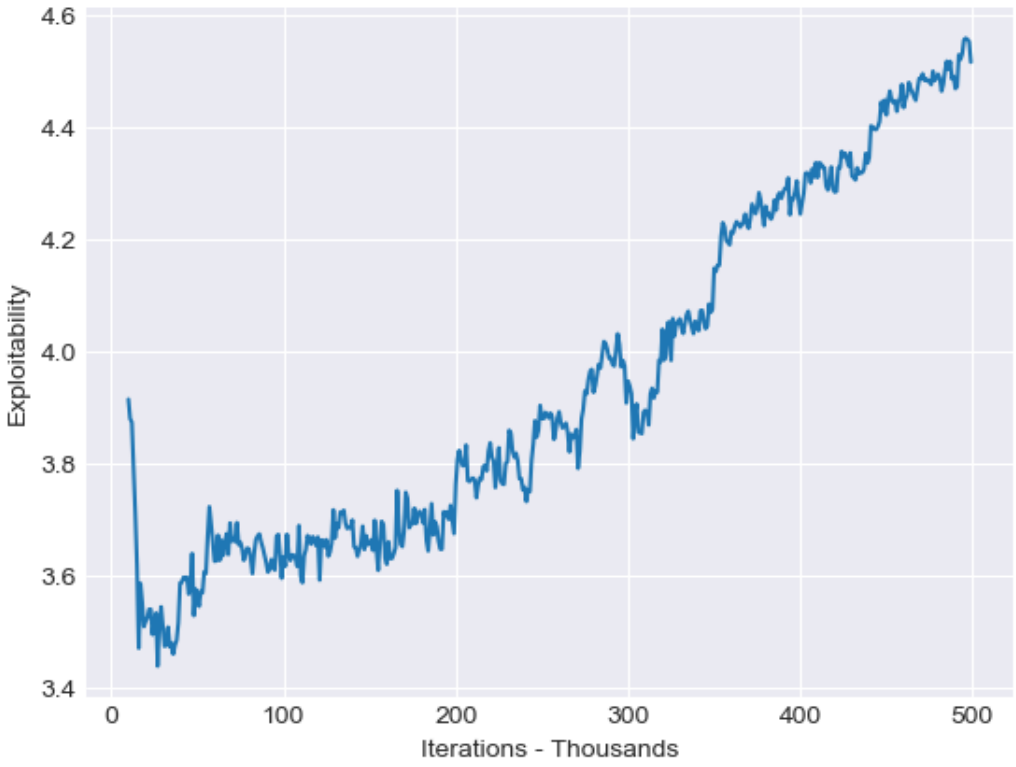

Figure 4.4: MCTS exploitability over time vs random player - 500000 Iterations

4.3.4 Analysis

The results gleaned from this experiment were unexpected to a degree. Heinrich had shown exploitability reaching lows of .8 and diverging to 1.5 in a similar experiment. Although a similar trend was shown here our exploitability values were significantly higher. This could potentially be explained by differences in the algorithm used to calculate exploitability or the implementation of UCT itself.

On further inspection of the search trees generated, along with the best response tree (see chapter 3) there appeared to be an over-fitting to the opponents strategy over time. Notably, both instances of the agent were playing more conservatively than expected. This meant that the best-response player could regularly force our agent to fold in situations where folding against a less conservative player would be illogical. In order to conceptualize this phenomenon the following example is given. Let's say player one is very conservative and they decide to only raise in the second round of the game when they have a pair of aces. This means that over time player two will only receive highly negative rewards for the states in which player one has raised in the second round. In fact, in this case it is more beneficial for player two to simply fold if player one raises in the second round of the game. When a new player is introduced to the system however, they can take advantage of this behavior by simply raising more frequently in the second round of the game. This type of strategic feedback loop is characteristic of deterministic strategies due to the fact that breaking such a loop becomes less and less likely the closer we get to purely greedy action selection.

4.4 Experiment 3 - Smooth UCT

4.4.1 Objective

4.4.2 Algorithm and Coding

4.4.3 Results

4.4.4 Analysis

Chapter 5

Conclusions

5.1 Summary

5.2 Reflections

5.3 Future Work

Bibliography

Dahl, F. A. (2001), A reinforcement learning algorithm applied to simplified two-player texas holdem poker, *in* 'European Conference on Machine Learning', Springer, pp. 85–96.

Heinrich, J. (2017), Reinforcement Learning from Self-Play in Imperfect-Information Games, PhD thesis, UCL (University College London).

Heinrich, J., Lanctot, M. & Silver, D. (2015), Fictitious self-play in extensive-form games, *in* 'International Conference on Machine Learning', pp. 805–813.

Heinrich, J. & Silver, D. (2015), Smooth uct search in computer poker, *in* 'Twenty-Fourth International Joint Conference on Artificial Intelligence'.

Heinrich, J. & Silver, D. (2016), 'Deep reinforcement learning from self-play in imperfect-information games', *arXiv preprint arXiv:1603.01121* .

Jackson, E. (2013), Slumbot nl: Solving large games with counterfactual regret minimization using sampling and distributed processing, *in* 'AAAI Workshop on Computer Poker and Incomplete Information'.

Johanson, M. (2013), 'Measuring the size of large no-limit poker games', *arXiv preprint arXiv:1302.7008* .

Johanson, M., Waugh, K., Bowling, M. & Zinkevich, M. (2011), Accelerating best response calculation in large extensive games, *in* 'Twenty-Second International Joint Conference on Artificial Intelligence'.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996), 'Reinforcement learning: A survey', *Journal of artificial intelligence research* **4**, 237–285.

Koller, D., Megiddo, N. & Von Stengel, B. (1996), 'Efficient computation of equilibria for extensive two-person games', *Games and economic behavior* **14**(2), 247–259.

Kuhn, H. (1953), 'Extensive games and the problem of information', *InH. KuhnandA. Tucker, editors, Contributions to the Theory of Games* pp. 193–216.

LazyProgrammerInc. (2018), 'Artificial intelligence: Reinforcement learning in python'.
URL: *https://www.udemy.com/artificial-intelligence-reinforcement-learning-in-python/*

Leslie, D. S. & Collins, E. J. (2006), 'Generalised weakened fictitious play', *Games and Economic Behavior* **56**(2), 285–298.

Mitchell, T. M. et al. (1997), 'Machine learning. 1997', *Burr Ridge, IL: McGraw Hill* **45**(37), 870–877.

Myerson, R. B. (2013), *Game theory*, Harvard university press.

Russell, S. J. & Norvig, P. (2016), *Artificial intelligence: a modern approach*, Malaysia; Pearson Education Limited,.

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

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. et al. (2016), 'Mastering the game of go with deep neural networks and tree search', *nature* **529**(7587), 484.

Southey, F., Bowling, M. P., Larson, B., Piccione, C., Burch, N., Billings, D. & Rayner, C. (2012), 'Bayes' bluff: Opponent modelling in poker', *arXiv preprint arXiv:1207.1411* .

Sutton, R. S., Barto, A. G., Bach, F. et al. (1998), *Reinforcement learning: An introduction*, MIT press.

Tesauro, G. (1995), Td-gammon: A self-teaching backgammon program, *in* 'Applications of Neural Networks', Springer, pp. 267–285.

Vermorel, J. & Mohri, M. (2005), Multi-armed bandit algorithms and empirical evaluation, *in* 'European conference on machine learning', Springer, pp. 437–448.

Von Stengel, B. (1996), 'Efficient computation of behavior strategies', *Games and Economic Behavior* **14**(2), 220–246.

Watkins, C. J. C. H. (1989), Learning from delayed rewards, PhD thesis, King's College, Cambridge.

Zinkevich, M., Johanson, M., Bowling, M. & Piccione, C. (2008), Regret minimization in games with incomplete information, *in* 'Advances in neural information processing systems', pp. 1729–1736.