Learning to Play Texas 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 the more 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 one interesting area is that of imperfect information games such as texas holdem.

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 peoples's everyday life. 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. Finally we can apply the generated strategy 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 advantage of these approaches is that they do 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). In another RL was combined with techniques inspired by game theory(Heinrich et al. 2015). This latter approach will be the basis for our texas hold'em agent as we attempt to replicate and build upon the results outlined in this paper.

These 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 et al. 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. For Leduc Hold'em, with a 6 card deck and 300 seconds of training time the initial exploitability was slightly over 1.2 and descended and converged to under .5. When the deck size increased to 60 cards the exploitability began at over 5.5 and converged to roughly 1.2 in the same period of time. 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 second paper on the matter(Heinrich & Silver 2016). In this paper, rather than using exploitability as an evaluation method, the agent was simply compared against the best hold'em agents from the ACPC of the previous year. Thus win rate was used as a the evaluation mechanism. 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 three competitors in the ACPC, the fully trained agent achieved a win rate of between -50 and -15 mbb/h. This score is consistent with that of a player that is slightly inferior but still competitive with these superhuman poker agents. As such, if the second iteration of our project is completed the goal will be to have a win rate of -200mbb/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 a website 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.

The same can be said

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 Methodology

1.5 Motivation

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).

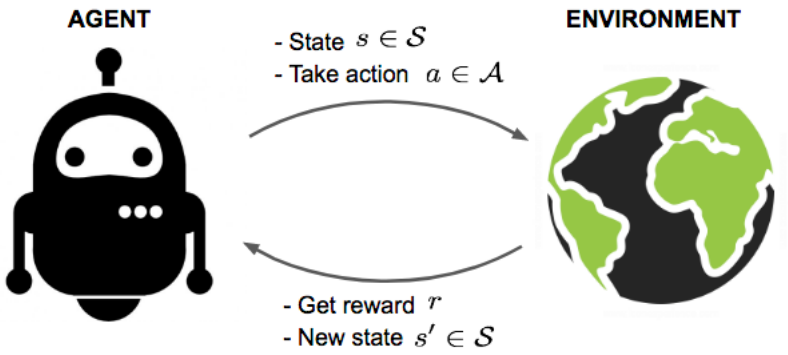

Figure 2.1: Reinforcement Learning

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.

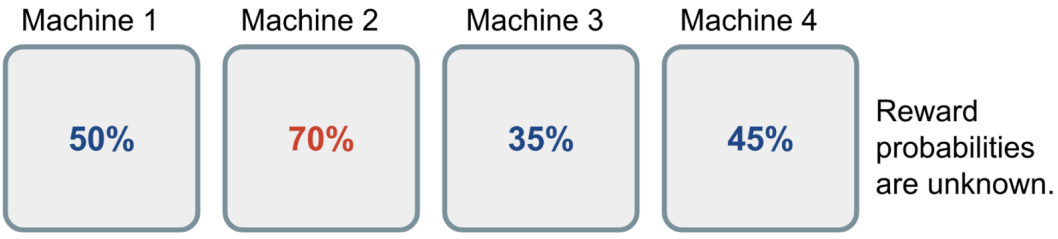

Figure 2.2: Multi Armed Bandit

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 $\epsilon_t = \frac{\epsilon_0}{t}$.

Interval Estimation Strategy

Another approach that can be used is called the interval estimation strategy. 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.

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, also known as 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, also known as 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

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). 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. As such, 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-value pairs are generally evaluated. This mapping of state-action pairs to values is referred to as the q 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.

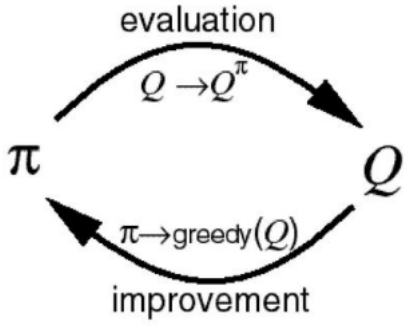

Figure 2.3: Monte Carlo Policy Improvement

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 that state. 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.

TD Learning Policy Improvement - Q-Learning

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. Certain game theoretic concepts were pervasive in the texas hold'em implementations and as such a brief study of these concepts was conducted.

2.2.1 Extensive-form Games

2.2.2 Nash Equilibria

2.2.3 Fictitious Play

2.3 Supervised Learning

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. As such this learning task is a generalization problem. The agent must be able to identify general features of the input data and how they map to the output. Common examples of

2.4 Texas Hold'em

Texas hold'em is one variant of the 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 cards are dealt. The third round is called the turn where one additional public card is dealt. Finally in the fourth round another 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

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 Texas Hold'em Using Counterfactual Regret Minimization

2.4.5 Texas Hold'em Using Reinforcement Learning

2.4.6 Texas Hold'em Using Neural Fictitious Self-Play

Chapter 3

Application Development (10)

3.1 Requirements

3.2 Design

3.3 Backend API

3.4 Frontend Website

3.5 Testing

3.6 Issues

Chapter 4

Empirical Studies

4.1 Experiment 1

4.2 Experiment 2

4.3 Experiment 3

4.4 Experiment 4

4.5 Experiment 5

4.6 Experiment 6

4.7 Experiment 7

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., 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. (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'.

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

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

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

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.

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.

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